**K. J. Somaiya College of Engineering, Mumbai-77**
(A Constituent College of Somaiya Vidyavihar University)
**Department of Computer Engineering**

| | |
|---|---|
| **Batch:** D-2 | **Roll No.:** 16010122151 |
| **Experiment No. 08** | |
| **Grade: AA / AB / BB / BC / CC / CD /DD** | |
| **Signature of the Staff In-charge with date** | |

**TITLE:** Implementation of Deadlock Avoidance Policy.

**AIM:** Implementation of Process synchronization algorithms using mutexes and semaphore – Dining Philosopher problem

**Expected Outcome of Experiment:**

**CO 3.** To understand the concepts of process synchronization and deadlock.

**Books/ Journals/ Websites referred:**

1. **Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate "Operating Systems", McGraw Hill Third Edition.**
3. **Sumitabha Das " UNIX Concepts & Applications", McGraw Hill Second Edition.**

**Pre Lab/ Prior Concepts:**

Knowledge of deadlocks and all deadlock avoidance methods.

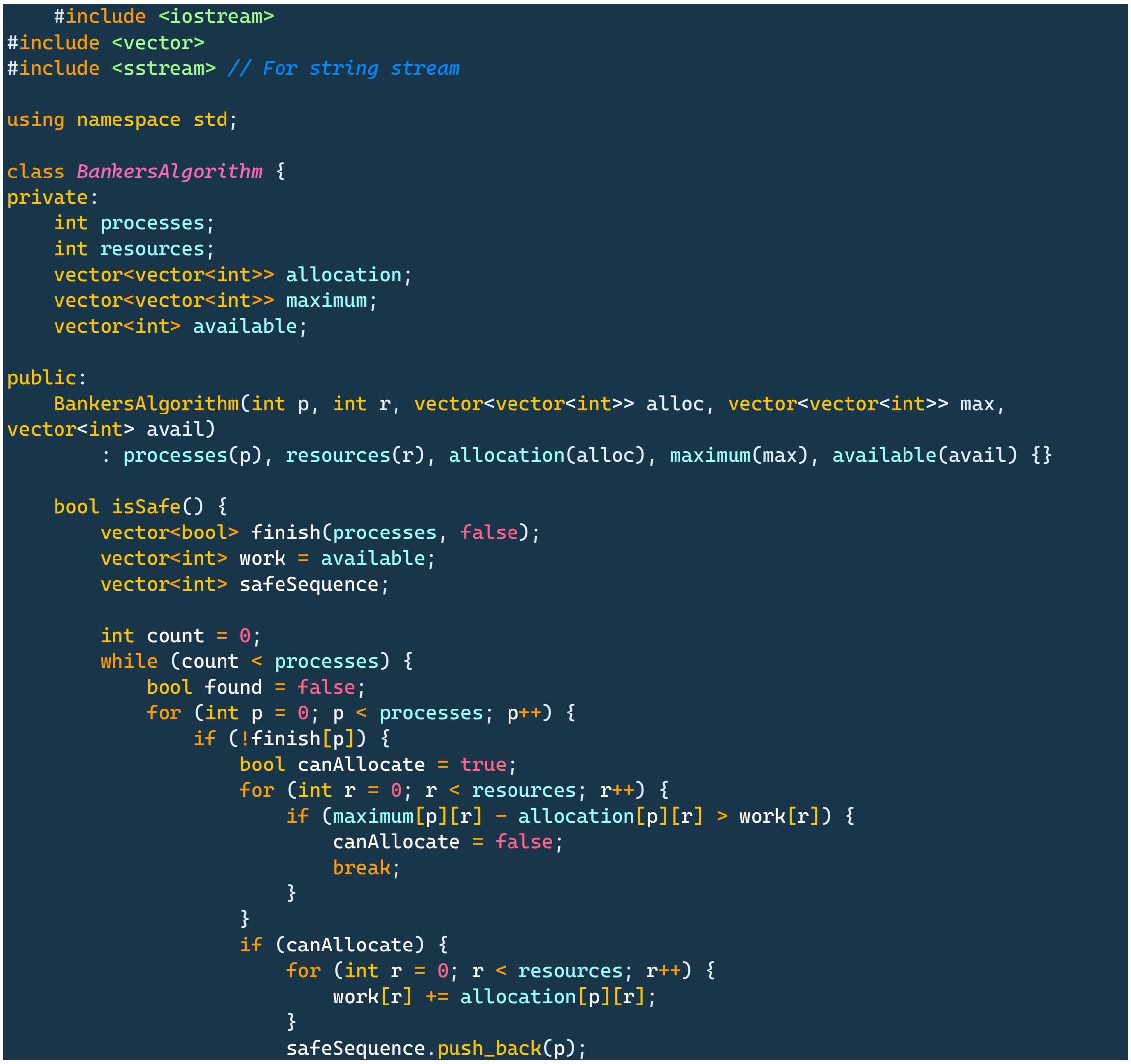
**Description of the application to be implemented**:

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra.

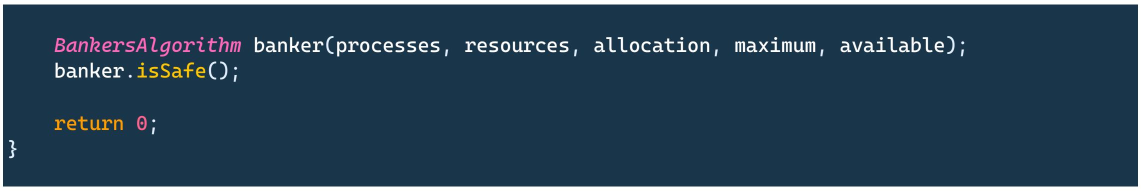
**DATA STRUCTURES**

(where *n* is the number of processes in the system and *m* is the number of resource types)

## Implementation details:

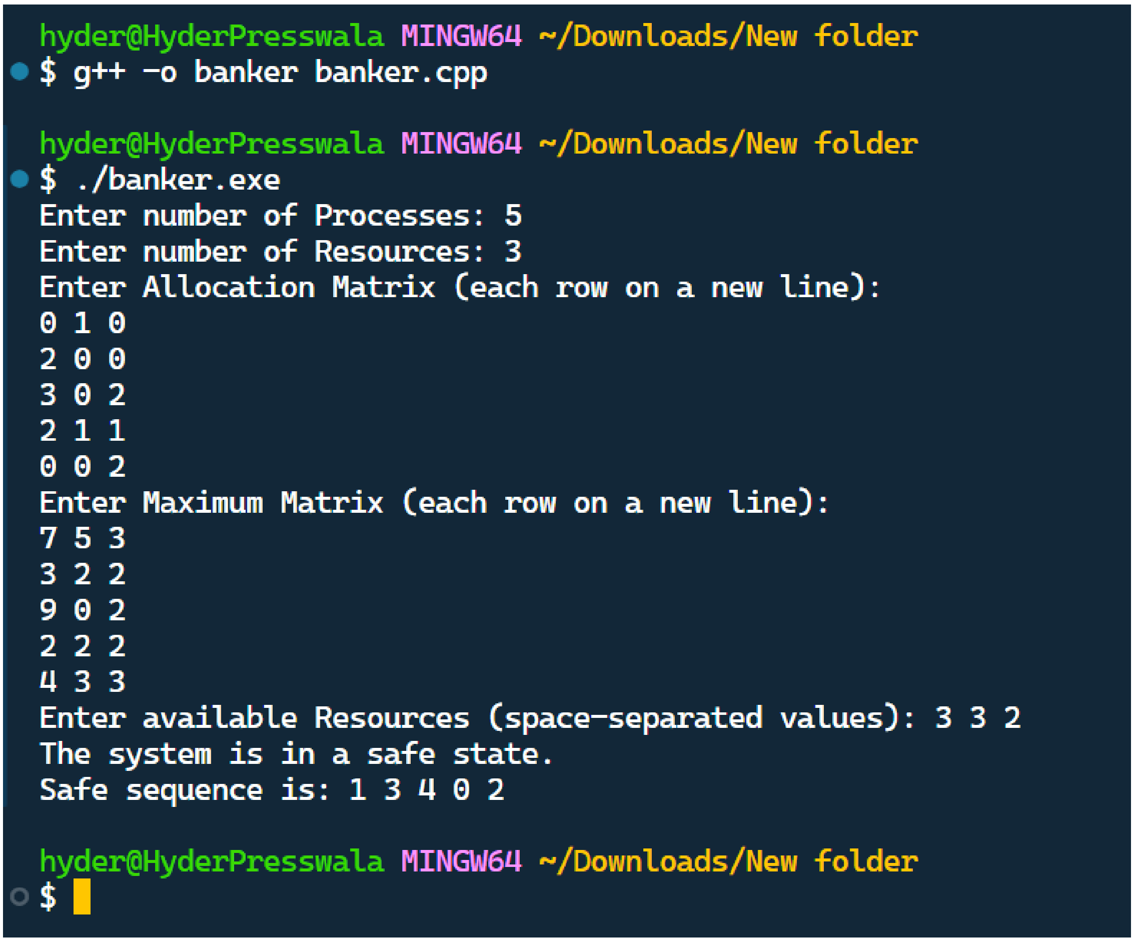


```
    #include <iostream>
#include <vector>
#include <sstream> // For string stream

using namespace std;

class BankersAlgorithm {
private:
    int processes;
    int resources;
    vector<vector<int>> allocation;
    vector<vector<int>> maximum;
    vector<int> available;

public:
    BankersAlgorithm(int p, int r, vector<vector<int>> alloc, vector<vector<int>> max,
vector<int> avail)
        : processes(p), resources(r), allocation(alloc), maximum(max), available(avail) {}

    bool isSafe() {
        vector<bool> finish(processes, false);
        vector<int> work = available;
        vector<int> safeSequence;

        int count = 0;
        while (count < processes) {
            bool found = false;
            for (int p = 0; p < processes; p++) {
                if (!finish[p]) {
                    bool canAllocate = true;
                    for (int r = 0; r < resources; r++) {
                        if (maximum[p][r] - allocation[p][r] > work[r]) {
                            canAllocate = false;
                            break;
                        }
                    }
                    if (canAllocate) {
                        for (int r = 0; r < resources; r++) {
                            work[r] += allocation[p][r];
                        }
                        safeSequence.push_back(p);
                        finish[p] = true;
                        found = true;
                        count++;
                    }
                }
            }
            if (!found) {
                cout << "The system is not in a safe state." << endl;
                return false;
            }
        }

        cout << "The system is in a safe state." << endl;
        cout << "Safe sequence is: ";
        for (int i : safeSequence) {
            cout << i << " ";
        }
        cout << endl;
        return true;
    }
};

int main() {
    int processes, resources;
    cout << "Enter number of Processes: ";
    cin >> processes;
    cout << "Enter number of Resources: ";
    cin >> resources;

    vector<vector<int>> allocation(processes, vector<int>(resources));
    cout << "Enter Allocation Matrix (each row on a new line):" << endl;

    // Read the allocation matrix
    cin.ignore(); // Clear the newline left in the buffer
    for (int i = 0; i < processes; i++) {
        string line;
        getline(cin, line); // Get the entire line
        stringstream ss(line);
        for (int j = 0; j < resources; j++) {
            ss >> allocation[i][j]; // Read each number
        }
    }

    vector<vector<int>> maximum(processes, vector<int>(resources));
    cout << "Enter Maximum Matrix (each row on a new line):" << endl;

    // Read the maximum matrix
    for (int i = 0; i < processes; i++) {
        string line;
        getline(cin, line); // Get the entire line
        stringstream ss(line);
        for (int j = 0; j < resources; j++) {
            ss >> maximum[i][j]; // Read each number
        }
    }

    vector<int> available(resources);
    cout << "Enter available Resources (space-separated values): ";
    for (int i = 0; i < resources; i++) {
        cin >> available[i];
    }
```



```
    BankersAlgorithm banker(processes, resources, allocation, maximum, available);
    banker.isSafe();

    return 0;
}
```

Output:-



```
hyder@HyderPresswala MINGW64 ~/Downloads/New folder
$ g++ -o banker banker.cpp

hyder@HyderPresswala MINGW64 ~/Downloads/New folder
$ ./banker.exe
Enter number of Processes: 5
Enter number of Resources: 3
Enter Allocation Matrix (each row on a new line):
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter Maximum Matrix (each row on a new line):
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter available Resources (space-separated values): 3 3 2
The system is in a safe state.
Safe sequence is: 1 3 4 0 2

hyder@HyderPresswala MINGW64 ~/Downloads/New folder
$ 
```

**Conclusion:** In this experiment we learned and successfully implemented banker's algorithm for resource allocation and deadlock avoidance.

## Post Lab Descriptive Questions

1. Explain the concept of a "safe state" in the context of the Banker's Algorithm. Why is it important for deadlock avoidance?

### Safe State in the Banker's Algorithm

The concept of a **safe state** in the context of the Banker's Algorithm is crucial for ensuring the system operates without entering a deadlock situation. Here's a detailed explanation:

### Definition of Safe State

A system is considered to be in a **safe state** if there exists at least one sequence of processes that can be executed to completion without causing a deadlock. In other words, if all the processes request their maximum resource needs at some point, the system can still satisfy those requests in a manner that allows all processes to finish.

### How the Banker's Algorithm Works

1. **Resource Allocation**: The Banker's Algorithm maintains a table that tracks:
   - **Allocation**: The current allocation of resources to each process.
   - **Maximum**: The maximum resources each process may need.
   - **Available**: The available resources in the system.
2. **Safety Check**: When a process requests resources, the algorithm performs a safety check to determine if granting the request keeps the system in a safe state:
   - It temporarily pretends to allocate the requested resources.
   - It then checks if there is a sequence of processes that can complete with the remaining resources.
   - If such a sequence exists, the state is safe; otherwise, it is unsafe.

### Importance of Safe State for Deadlock Avoidance

1. **Prevention of Deadlock**: By ensuring that the system only grants resource requests that keep it in a safe state, the Banker's Algorithm effectively prevents deadlocks. A deadlock occurs when processes are waiting for resources held by each other in a circular manner, and a safe state ensures that processes can always eventually proceed to completion.
2. **Resource Management**: The concept helps in managing resources efficiently. By analyzing maximum needs versus current allocations, the algorithm can dynamically adjust resource allocations based on current system states.

3. **Guaranteed Completion**: Safe states guarantee that every process can finish its execution if the system remains in a safe state. This is essential for systems requiring high reliability, such as operating systems in real-time applications.

2. Describe the role of the need matrix in the Banker's Algorithm. How is it calculated and used?

The **Need Matrix** is a crucial component of the Banker's Algorithm, serving as a key factor in determining whether resource requests can be safely granted to processes without leading to a deadlock. Here's a detailed explanation of its role, how it is calculated, and how it is used:

**Definition of the Need Matrix**

The Need Matrix represents the remaining resource needs of each process. It is defined as the difference between the Maximum Matrix (the maximum resources that each process could potentially need) and the Allocation Matrix (the resources that have already been allocated to each process).

**Calculation of the Need Matrix**

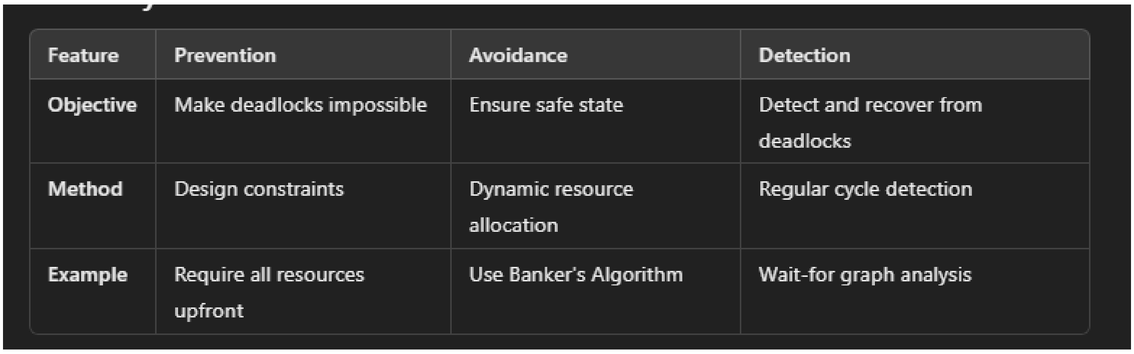
The Need Matrix is calculated using the formula:

Need[i][j]=Maximum[i][j]−Allocation[i][j]

Where:

- Need[i][j] is the number of resources of type j still needed by process i.
- Maximum[i][j] is the maximum resources of type j that process iii may need.
- Allocation[i][j] is the number of resources of type j currently allocated to process iii.

3. What are the differences between deadlock prevention, avoidance, and detection? Provide examples of each.



| Feature | Prevention | Avoidance | Detection |
|---|---|---|---|
| Objective | Make deadlocks impossible | Ensure safe state | Detect and recover from deadlocks |
| Method | Design constraints | Dynamic resource allocation | Regular cycle detection |
| Example | Require all resources upfront | Use Banker's Algorithm | Wait-for graph analysis |

4. How does the Resource Allocation Graph (RAG) help in detecting potential deadlocks?

The **Resource Allocation Graph (RAG)** is a graphical representation used in operating systems to depict the relationships between processes and the resources they require. It plays a critical role in detecting potential deadlocks by visually illustrating how resources are allocated and how processes are waiting for those resources.

5. In the Banker's Algorithm, what steps are taken if a resource request leads to an unsafe state?

In the Banker's Algorithm, if a resource request leads to an unsafe state, the algorithm takes specific steps to handle the situation. Here's a detailed explanation of these steps:

**Steps Taken When a Resource Request Leads to an Unsafe State**

1. **Revert the Request**:
    - The first step is to deny the resource request. Since granting the request would lead the system into an unsafe state, the system must maintain its safety by not fulfilling the request.
2. **Maintain the Current State**:
    - The algorithm preserves the current allocations and resource availability. It does not change the state of the system or the resources allocated to processes.
3. **Provide Feedback to the Process**:
    - Inform the requesting process that its request has been denied due to the potential for an unsafe state. This allows the process to understand why it cannot proceed with the requested resources.
4. **Allow the Process to Retry**:
    - The process may have the option to retry the request later. This can happen when the overall resource availability changes (e.g., other processes complete and release resources), potentially allowing the request to be satisfied safely in the future.
5. **Potentially Use Alternative Strategies**:
    - If requests continue to lead to unsafe states, the system may implement alternative strategies, such as:
        - **Resource preemption**: Temporarily take resources from processes that can be safely rolled back.
        - **Process termination**: In extreme cases, terminate one or more processes to free up resources.

6. Analyze the impact of resource allocation policies on system throughput and process starvation. How can these policies be optimized?

### Impact of Resource Allocation Policies on System Throughput and Process Starvation

Resource allocation policies are crucial in managing how resources are distributed among competing processes. Their design significantly impacts both system throughput and the likelihood of process starvation. Here's an analysis of these impacts and how policies can be optimized.

#### 1. System Throughput

**Definition**: System throughput is the number of processes that complete their execution in a given amount of time.

**Impact of Resource Allocation Policies**:

- **Fairness vs. Efficiency**: Some policies, like the First-Come, First-Served (FCFS), may lead to high waiting times for processes, reducing overall throughput. In contrast, policies like Shortest Job Next (SJN) can enhance throughput by prioritizing shorter processes.
- **Resource Utilization**: Efficient resource allocation can minimize idle times. For example, if resources are allocated based on priority, critical processes can complete faster, thereby improving overall throughput.
- **Dynamic vs. Static Allocation**: Dynamic policies that adapt to the current state of the system (e.g., Banker's Algorithm) can optimize resource usage, leading to better throughput compared to static policies.

#### 2. Process Starvation

**Definition**: Starvation occurs when a process is perpetually denied the resources it needs to proceed, often due to higher-priority processes continuously receiving resources.

**Impact of Resource Allocation Policies**:

- **Priority Scheduling**: While priority-based policies can enhance throughput, they may lead to starvation for lower-priority processes if higher-priority ones monopolize resources.
- **Ageing Techniques**: Implementing ageing (gradually increasing the priority of waiting processes) can help prevent starvation but may reduce the efficiency of the overall system if not managed properly.
- **Resource Quotas**: Policies that impose quotas on resource usage can prevent any single process from starving others but may lead to inefficient resource use.

### Optimizing Resource Allocation Policies

To achieve a balance between high throughput and minimal starvation, resource allocation policies can be optimized through the following strategies:

1. **Dynamic Resource Allocation**:
   - Implement algorithms that adjust resource allocations based on current system needs and priorities (e.g., Banker's Algorithm), which can adapt to changing conditions.

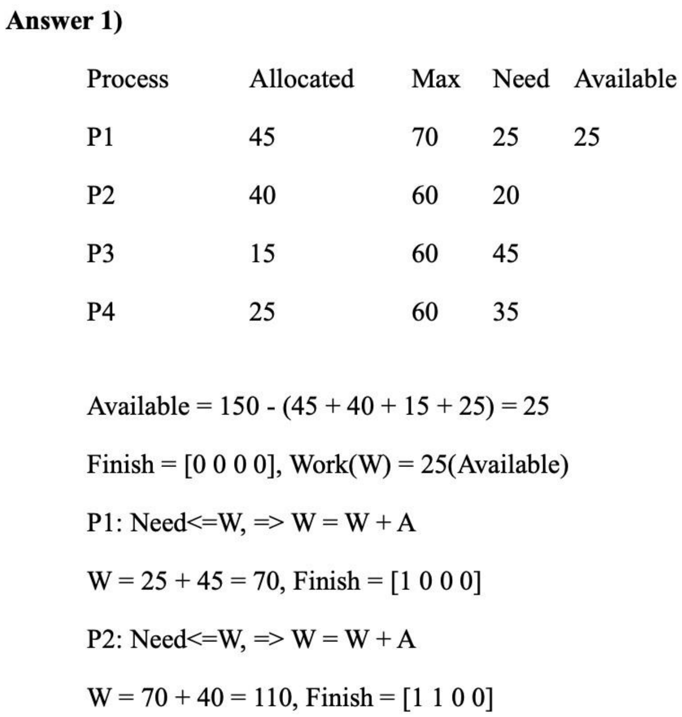
2. **Hybrid Scheduling**:
    - ○ Combine multiple scheduling strategies (e.g., round-robin for time-sharing combined with priority scheduling) to ensure fairness while maximizing throughput.
3. **Ageing**:
    - ○ Introduce ageing mechanisms to gradually increase the priority of waiting processes. This ensures that all processes eventually get the resources they need, reducing the likelihood of starvation.
4. **Resource Reservation**:
    - ○ Reserve resources for critical processes while allowing non-critical processes to access remaining resources. This ensures that essential tasks are completed promptly without completely starving other processes.
5. **Monitoring and Feedback**:
    - ○ Use monitoring tools to analyze system performance and resource usage patterns. Feedback mechanisms can help in adjusting allocation policies dynamically to maintain an optimal balance.
6. **Load Balancing**:
    - ○ Distribute workloads evenly across available resources to prevent bottlenecks. This can enhance throughput by ensuring that no single resource is overwhelmed while others remain underutilized.

7. Consider a system with total of 150 units of memory allocated to three processes as shown:

| **Process** | **Max** | **Hold** |
|---|---|---|
| P1 | 70 | 45 |
| P2 | 60 | 40 |
| P3 | 60 | 15 |

Apply Banker’s algorithm to determine whether it would be safe to grant each of the following request. If yes, indicate sequence of termination that could be possible.

1) The P4 process arrives with max need of 60 and initial need of 25 units.
2) The P4 process arrives with max need of 60 and initial need of 35 units.



**Answer 1)**

| Process | Allocated | Max | Need | Available |
|---|---|---|---|---|
| P1 | 45 | 70 | 25 | 25 |
| P2 | 40 | 60 | 20 | |
| P3 | 15 | 60 | 45 | |
| P4 | 25 | 60 | 35 | |

Available = 150 - (45 + 40 + 15 + 25) = 25

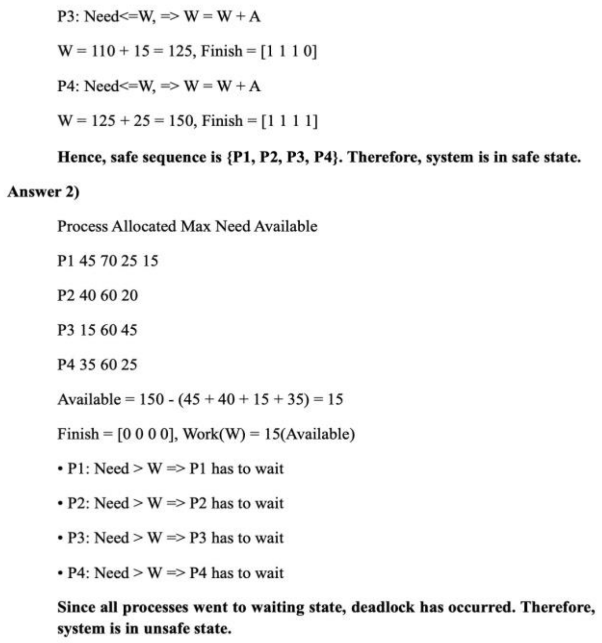
Finish = [0 0 0 0], Work(W) = 25(Available)

P1: Need<=W, => W = W + A

W = 25 + 45 = 70, Finish = [1 0 0 0]

P2: Need<=W, => W = W + A

W = 70 + 40 = 110, Finish = [1 1 0 0]



P3: Need<=W, => W = W + A

W = 110 + 15 = 125, Finish = [1 1 1 0]

P4: Need<=W, => W = W + A

W = 125 + 25 = 150, Finish = [1 1 1 1]

**Hence, safe sequence is {P1, P2, P3, P4}. Therefore, system is in safe state.**

**Answer 2)**

| Process | Allocated | Max | Need | Available |
|---|---|---|---|---|
| P1 | 45 | 70 | 25 | 15 |
| P2 | 40 | 60 | 20 | |
| P3 | 15 | 60 | 45 | |
| P4 | 35 | 60 | 25 | |

Available = 150 - (45 + 40 + 15 + 35) = 15

Finish = [0 0 0 0], Work(W) = 15(Available)

- P1: Need > W => P1 has to wait
- P2: Need > W => P2 has to wait
- P3: Need > W => P3 has to wait
- P4: Need > W => P4 has to wait

**Since all processes went to waiting state, deadlock has occurred. Therefore, system is in unsafe state.**

**Date:** 24-10-2024 **Signature of faculty in-charge**