VIDHI ROHIRA

S.Y B.TECH

SEM III

COMPUTER ENGINEERING

OS LAB 7

231071052

BATCH – C

LABORATORY 7

AIM:- The aim of this lab session is to implement the Banker's Algorithm in C++ to ensure safe resource allocation in a system with multiple processes. This involves preventing deadlock by evaluating resource requests based on the system's safety conditions.

THEORY:-

Q] WHAT IS BANKER'S ALGORITHM?

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra. It tests whether granting a request for resources leaves the system in a safe state by checking available resources, allocated resources, and process needs. The algorithm works as follows:

1. Calculate the need matrix for each process, which is defined as Max - Allocation.

2. Check if the requested resources for a process can be fulfilled, i.e., if the request does not exceed its needs and the available resources.

3. Simulate the allocation of the requested resources and check if the system remains in a safe state.

4. If the system remains safe, grant the request; otherwise, deny it.

ALGORITHM:

1] Input:

- Input the number of processes and resources.
- Provide the Max, Allocation, and Available matrices.

2] Calculate the Need matrix:

- Compute the Need matrix using the formula:

$$\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$$

3] Check Resource Requests:

- For each resource request made by a process, check if the request can be granted by verifying:

$$\text{Request}[i][j] \leq \text{Need}[i][j]$$

$$\text{Request}[i][j] \leq \text{Available}[j]$$

4] Simulate Allocation:

- If the request can be granted, simulate the resource allocation and check if the system remains in a safe state using the safety algorithm.

5] Grant or Deny Request:

- If the system is in a safe state after the simulated allocation, proceed with the allocation.
- If the system is not safe, rollback the allocation and deny the resource request.

CODE:-

```
class BankersAlgorithm:
    def __init__(self, processes, resources, max_matrix, allocation_matrix,
available_matrix):
        self.processes = processes
        self.resources = resources
        self.max = max_matrix
        self.allocation = allocation_matrix
        self.available = available_matrix
        self.need = [[0] * resources for _ in range(processes)]
        self.calculate_need()

    # Calculate the Need matrix
    def calculate_need(self):
        for i in range(self.processes):
            for j in range(self.resources):
                self.need[i][j] = self.max[i][j] - self.allocation[i][j]

    # Check if the system is in a safe state
    def is_safe(self):
        work = self.available[:]
        finish = [False] * self.processes
        safe_sequence = []
        count = 0

        while count < self.processes:
            found = False
            for i in range(self.processes):
                if not finish[i]:
                    if all(self.need[i][j] <= work[j] for j in
range(self.resources)):
                        for j in range(self.resources):
                            work[j] += self.allocation[i][j]
                        safe_sequence.append(i)
                        finish[i] = True
                        found = True
                        count += 1
            if not found:
                print("System is not in a safe state.")
                return False

        print("System is in a safe state. Safe Sequence is:", safe_sequence)
        return True

    # Request resources for a specific process
    def request_resources(self, process_id, request):
```

```
        if any(request[i] > self.need[process_id][i] for i in
range(self.resources)):
            print("Error: Request exceeds maximum need.")
            return False

        if any(request[i] > self.available[i] for i in range(self.resources)):
            print("Error: Resources not available.")
            return False

        temp_available = self.available[:]
        temp_allocation = [row[:] for row in self.allocation]
        temp_need = [row[:] for row in self.need]

        for i in range(self.resources):
            self.available[i] -= request[i]
            self.allocation[process_id][i] += request[i]
            self.need[process_id][i] -= request[i]

        if self.is_safe():
            print("Request granted.")
            return True
        else:
            print("Request cannot be granted as it would make the system
unsafe.")
            self.available = temp_available
            self.allocation = temp_allocation
            self.need = temp_need
            return False


# Main function to test the Banker's Algorithm
if __name__ == "__main__":
    processes = 5
    resources = 3

    max_matrix = [
        [7, 5, 3],
        [3, 2, 2],
        [9, 0, 2],
        [2, 2, 2],
        [4, 3, 3]
    ]

    allocation_matrix = [
        [0, 1, 0],
        [2, 0, 0],
        [3, 0, 2],
        [2, 1, 1],
        [0, 0, 2]
```

```
    ]

    available_matrix = [3, 3, 2]

    bankers = BankersAlgorithm(processes, resources, max_matrix,
allocation_matrix, available_matrix)

    # Initial system state check
    print("Initial system state:")
    bankers.is_safe()

    # Test case 1: Process 1 requesting [1, 0, 2]
    request1 = [1, 0, 2]
    print("\nProcess 1 requesting [1, 0, 2]:")
    bankers.request_resources(1, request1)

    # Test case 2: Process 4 requesting [2, 2, 2]
    request2 = [2, 2, 2]
    print("\nProcess 4 requesting [2, 2, 2]:")
    bankers.request_resources(4, request2)

    # Test case 3: Process 3 requesting [0, 1, 0]
    request3 = [0, 1, 0]
    print("\nProcess 3 requesting [0, 1, 0]:")
    bankers.request_resources(3, request3)

    # Test case 4: Process 0 requesting [0, 2, 0]
    request4 = [0, 2, 0]
    print("\nProcess 0 requesting [0, 2, 0]:")
    bankers.request_resources(0, request4)

    # Test case 5: Process 2 requesting [1, 0, 1]
    request5 = [1, 0, 1]
    print("\nProcess 2 requesting [1, 0, 1]:")
    bankers.request_resources(2, request5)
```

OUTPUT

```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/OS LAB/lab7_bankers.py"
Initial system state:
System is in a safe state. Safe Sequence is: [1, 3, 4, 0, 2]

Process 1 requesting [1, 0, 2]:
System is in a safe state. Safe Sequence is: [1, 3, 4, 0, 2]
Request granted.

Process 4 requesting [2, 2, 2]:
Error: Request exceeds maximum need.

Process 3 requesting [0, 1, 0]:
System is in a safe state. Safe Sequence is: [1, 3, 4, 0, 2]
Request granted.

Process 0 requesting [0, 2, 0]:
System is not in a safe state.
Request cannot be granted as it would make the system unsafe.

Process 2 requesting [1, 0, 1]:
Error: Request exceeds maximum need.
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB>
```

CONCLUSION:-

Hence, in this lab session, we have learned and implemented the Banker's Algorithm to effectively manage resource allocation and prevent deadlocks in multi-process systems. We explored how the algorithm calculates the Need matrix, checks the system's safety state, and simulates resource requests. By running various test cases, we observed how the algorithm dynamically grants or denies resource requests based on system conditions, ensuring that the system remains in a safe state at all times. This hands-on experience reinforced the importance of safety in resource allocation, highlighting how algorithms like the Banker's Algorithm play a critical role in maintaining stability in concurrent and distributed computing environments. Additionally, we deepened our understanding of how deadlock prevention can be achieved through systematic resource management techniques.