Experiment-5 (OS LAB)
Name: Veeransh Shah
Reg ID: 221070063

Aim:

Implementing a Bankers algorithm for Deadlock avoidance.

Theory:

The Banker's algorithm is a classic method used in operating systems for deadlock avoidance, particularly in multi-process systems where resources must be allocated dynamically to processes. It operates on the principle of preventing deadlock by carefully managing resource allocation based on a process's maximum demands and the currently available resources. The algorithm maintains a state where it continually assesses whether the system can safely grant resource requests without risking deadlock. This is achieved by simulating resource allocation scenarios and ensuring that the system remains in a safe state, where at least one sequence of process executions can complete without resource contention leading to deadlock. The algorithm's core idea lies in its ability to avoid resource allocation scenarios that could potentially lead to a state where no process can progress further due to resource unavailability.

In practice, the Banker's algorithm maintains three main data structures: allocation, maximum demand, and available resources. These structures represent the current allocation of resources to processes, the maximum resources a process may need, and the resources available in the system, respectively. By carefully comparing process requests with available resources and the maximum demands of processes, the algorithm can grant or deny resource requests in a way that ensures the system's safety. This proactive approach to resource management significantly reduces the risk of deadlocks, thereby enhancing system stability and ensuring efficient resource utilization. Through its meticulous analysis of resource allocation scenarios, the Banker's algorithm provides a robust framework for managing resources in multi-process environments, contributing to the overall reliability and performance of operating systems.

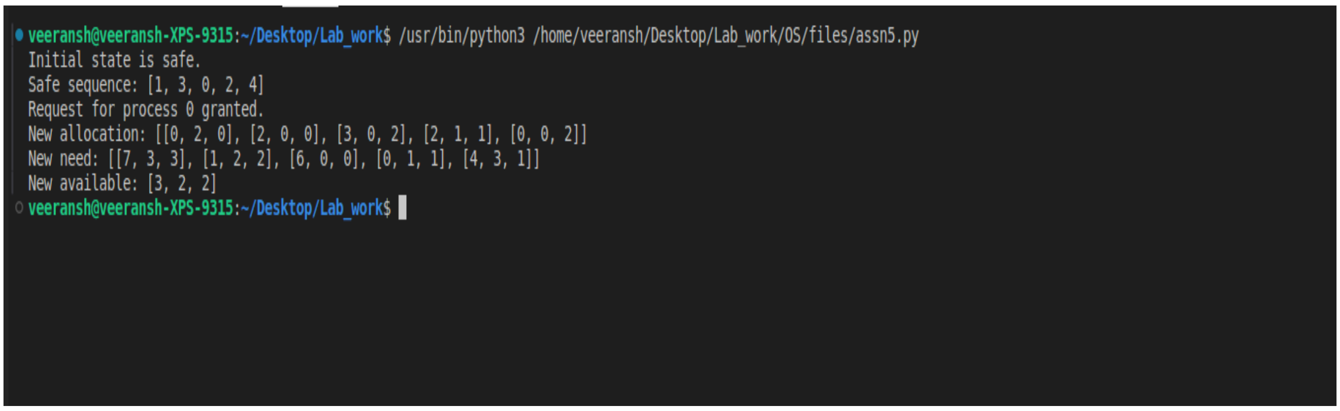
Code:

```
class BankersAlgorithm:
   def __init__(self, allocation, max_demand, available):
       self.allocation = allocation
       self.max_demand = max_demand
       self.available = available
       self.num_processes = len(allocation)
       self.num_resources = len(available)
```

```
        self.need = [[max_demand[i][j] - allocation[i][j] for j in
range(self.num_resources)] for i in range(self.num_processes)]
        self.safe_sequence = []

    def is_safe_state(self):
        work = self.available[:]
        finish = [False] * self.num_processes
        while True:
            found = False
            for i in range(self.num_processes):
                if not finish[i] and all(need <= work[j] for j, need in
enumerate(self.need[i])):
                    work = [work[j] + self.allocation[i][j] for j in
range(self.num_resources)]
                    self.safe_sequence.append(i)
                    finish[i] = True
                    found = True
                    break
            if not found:
                if all(finish):
                    return True
                else:
                    return False

    def request_resources(self, process_id, request):
        if all(req <= need for req, need in zip(request,
self.need[process_id])):
            if all(req <= available for req, available in zip(request,
self.available)):
                new_allocation = [a + b for a, b in
zip(self.allocation[process_id], request)]
                new_need = [n - r for n, r in zip(self.need[process_id],
request)]
                if BankersAlgorithm(self.allocation, self.max_demand,
self.available).is_safe_state():
                    self.allocation[process_id] = new_allocation
                    self.need[process_id] = new_need
                    self.available = [a - r for a, r in
zip(self.available, request)]
                    return True
        return False
```

```python
if __name__ == "__main__":
   allocation = [[0, 1, 0], [2, 0, 0], [3, 0, 2], [2, 1, 1], [0, 0, 2]]
   max_demand = [[7, 5, 3], [3, 2, 2], [9, 0, 2], [2, 2, 2], [4, 3, 3]]
   available = [3, 3, 2]

   banker = BankersAlgorithm(allocation, max_demand, available)
   if banker.is_safe_state():
       print("Initial state is safe.")
       print("Safe sequence:", banker.safe_sequence)
       process_id = 0
       request = [0, 1, 0]
       if banker.request_resources(process_id, request):
           print(f"Request for process {process_id} granted.")
           print("New allocation:", banker.allocation)
           print("New need:", banker.need)
           print("New available:", banker.available)
       else:
           print(f"Request for process {process_id} denied.")
   else:
       print("Initial state is not safe.")
```

Output:



```
veeransh@veeransh-XPS-9315:~/Desktop/Lab_work$ /usr/bin/python3 /home/veeransh/Desktop/Lab_work/OS/files/assn5.py
Initial state is safe.
Safe sequence: [1, 3, 0, 2, 4]
Request for process 0 granted.
New allocation: [[0, 2, 0], [2, 0, 0], [3, 0, 2], [2, 1, 1], [0, 0, 2]]
New need: [[7, 3, 3], [1, 2, 2], [6, 0, 0], [0, 1, 1], [4, 3, 1]]
New available: [3, 2, 2]
veeransh@veeransh-XPS-9315:~/Desktop/Lab_work$
```

Conclusion:

The code implements the Banker's algorithm for deadlock avoidance, allowing processes to request and release resources while ensuring that the system remains in a safe state to avoid deadlocks.