Experiment -6 (OS LAB)
Name: Veeransh Shah
Reg ID: 221070063

Aim:

Implement an algorithm to detect the deadlock in given input resource allocation graph.

Theory:

The provided code implements the Banker's algorithm, which is a resource allocation and deadlock avoidance algorithm used in operating systems. In essence, it helps ensure that processes requesting resources do not enter a state where they are unable to proceed due to resource contention, thereby avoiding deadlock situations. The algorithm operates by maintaining data structures representing the current state of resource allocation and maximum resource needs for each process. It simulates resource allocation and checks if granting resources to a process would result in a safe state, where all processes can eventually complete their execution without causing deadlock. If the system is in a safe state, resources are allocated; otherwise, the system denies resource allocation to prevent deadlock.

The `BankerAlgorithm` class encapsulates the core functionalities of the Banker's algorithm. It includes methods to check if the system is in a safe state (`is_safe_state`) and to detect deadlock (`detect_deadlock`). The `is_safe_state` method simulates the allocation of resources and verifies if all processes can complete execution. If so, it returns `True`; otherwise, it returns `False`. The `detect_deadlock` method utilizes `is_safe_state` to determine if the system is in a deadlock state. The code exemplifies the algorithm's application through two examples, demonstrating its effectiveness in detecting deadlock scenarios and ensuring system stability by avoiding resource allocation that could lead to deadlock.

Code:

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

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

    def detect_deadlock(self):
        work = list(self.available)
        need = [[self.max_claim[i][j] - self.allocation[i][j] for j in
range(self.num_resources)] for i in range(self.num_processes)]
        finish = [False] * self.num_processes
        if self.is_safe_state(finish, work, need):
            return False
        return True


allocation1 = [[0, 1, 0], [2, 0, 0], [3, 0, 2], [2, 1, 1], [0, 0, 2]]
max_claim1 = [[7, 5, 3], [3, 2, 2], [9, 0, 2], [2, 2, 2], [4, 3, 3]]
available1 = [3, 3, 2]

allocation2 = [[1, 1, 0], [2, 0, 0], [3, 0, 2], [2, 1, 1], [0, 0, 2]]
max_claim2 = [[7, 5, 3], [3, 2, 2], [9, 0, 2], [2, 2, 2], [4, 3, 3]]
available2 = [3, 3, 2]

banker1 = BankerAlgorithm(allocation1, max_claim1, available1)
if banker1.detect_deadlock():
    print("Deadlock detected in Example 1.")
else:
    print("No deadlock detected in Example 1.")

banker2 = BankerAlgorithm(allocation2, max_claim2, available2)
if banker2.detect_deadlock():
    print("Deadlock detected in Example 2.")
else:
```

```
    print("No deadlock detected in Example 2.")
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    COMMENTS

veeransh@veeransh-XPS-9315:~/Desktop/Lab_work$ /usr/bin/python3 /home/veeransh/Desktop/Lab_work/OS/files/assn6.py
No deadlock detected in Example 1.
Deadlock detected in Example 2.
veeransh@veeransh-XPS-9315:~/Desktop/Lab_work$
```

Conclusion:

In conclusion, the Banker's algorithm presented in the code serves as a critical tool in operating systems for managing resource allocation and avoiding deadlock situations. By maintaining data structures representing resource allocation and process needs, the algorithm ensures that resources are allocated in a manner that prevents deadlock. Through the implementation of methods to check for a safe system state and to detect deadlock, the algorithm provides a reliable mechanism for ensuring system stability and preventing resource contention issues.