**Experiment 7**

**Aim:** To study and implement Deadlock Avoidance Algorithm

**Learning Objectives:**

* Understand the concept of deadlock avoidance and its importance in operating systems.
* Learn to implement deadlock avoidance algorithms, focusing on safe resource allocation**.**

**Theory :**

**Deadlock and Its Consequences**

In operating systems, a deadlock occurs when a set of processes becomes stuck in a state where each process is waiting for a resource that another process in the set holds. This leads to a situation where none of the processes can proceed, effectively halting system operations. Deadlock is a critical issue in concurrent systems, particularly in environments where multiple processes require access to shared resources.

**Deadlock Avoidance**

Deadlock avoidance is a strategy used to ensure that the system never enters an unsafe state where deadlock could potentially occur. Unlike deadlock prevention, which imposes strict constraints to prevent deadlocks entirely, deadlock avoidance allows for more flexible resource allocation by carefully analyzing the state of the system before granting resources.

**Banker's Algorithm**

One of the most well-known deadlock avoidance algorithms is the Banker's Algorithm. It is named for its analogy to a banking system where the bank must ensure that it can meet the needs of all its customers without running out of funds. In the context of operating systems, the Banker's Algorithm works by ensuring that each resource allocation does not lead to an unsafe state.

**The algorithm involves:**

* Maximum Demand Calculation: Each process declares the maximum number of resources it may need.
* Resource Allocation: Resources are allocated to processes only if the system can guarantee that it will remain in a safe state after the allocation.
* Safety Check: The system checks if there is a sequence of processes that can be safely executed without leading to a deadlock.

If the system finds that the resource allocation will result in an unsafe state, it does not grant the resource request, thus avoiding the risk of deadlock.

**Implementing Deadlock Avoidance**

Implementing a deadlock avoidance algorithm involves creating a program that tracks the current resource allocation, maximum demand, and availability of resources. The program must continuously check the state of the system before allocating resources to ensure that no unsafe state is reached.

**Learning Outcomes:**

* Ability to apply deadlock avoidance strategies to prevent system deadlocks.
* Understanding of how to analyze resource allocation states to determine system safety.

**Implementation:**

**P = 5**

**R = 3**

**def calculateNeed(need, maxm, allot):**

**for i in range(P):**

**for j in range(R):**

**need[i][j] = maxm[i][j] - allot[i][j]**

**def isSafe(processes, avail, maxm, allot):**

**need = []**

**for i in range(P):**

**l = []**

**for j in range(R):**

**l.append(0)**

**need.append(l)**

**calculateNeed(need, maxm, allot)**

**finish = [0] \* P**

**safeSeq = [0] \* P**

**work = [0] \* R**

**for i in range(R):**

**work[i] = avail[i]**

**count = 0**

**while (count < P):**

**found = False**

**for p in range(P):**

**if (finish[p] == 0):**

**for j in range(R):**

**if (need[p][j] > work[j]):**

**break**

**if (j == R - 1):**

**for k in range(R):**

**work[k] += allot[p][k]**

**safeSeq[count] = p**

**count += 1**

**finish[p] = 1**

**found = True**

**if (found == False):**

**print("System is not in safe state")**

**return False**

**print("System is in safe state.",**

**"\nSafe sequence is: ", end=" ")**

**print(\*safeSeq)**

**return True**

**processes = [0, 1, 2, 3, 4]**

**avail = [3, 3, 2]**

**maxm = [[7, 5, 3], [3, 2, 2], [9, 0, 2], [2, 2, 2], [4, 3, 3]]**

**allot = [[0, 1, 0], [2, 0, 0], [3, 0, 2], [2, 1, 1], [0, 0, 2]]**

**isSafe(processes, avail, maxm, allot)**

**Output:**

****