Experiment No.7
Process Management: Deadlock a. Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Process Management: Deadlock

Objective:

a. Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm

Theory:

Data Structures for the Banker's Algorithm.

Let n = number of processes, and m = number of resources types.

v Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available

v Max: n x m matrix.

If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj

v Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj

v Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task

Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively. Initialize:
   Work = Available
   Finish [i] = false for i = 0, 1, …, n- 1 2. Find an i such that both:
   (a) Finish [i] = false

   (b) Needi ≤ Work

If no such i exists, go to step 4 3. Work = Work + Allocationi
Finish[i] = true go to step 2
4. If Finish [i] == true for all i, then the system is in a safe state.

Code:

```
#include <stdio.h>

// Maximum number of processes and resources
#define P 5
#define R 3

// Function to find the need of each process
void calculateNeed(int need[P][R], int max[P][R], int allot[P][R])
{
   for (int i = 0; i < P; i++)
      for (int j = 0; j < R; j++)
         need[i][j] = max[i][j] - allot[i][j];
}

// Function to find whether the system is in a safe state or not
int isSafe(int processes[], int avail[], int max[][R], int allot[][R])
{
   int need[P][R];
   calculateNeed(need, max, allot);

   int finish[P] = {0}; // Finish state for each process
   int safeSeq[P];      // Safe sequence
   int work[R];         // Available resources

   // Initialize work with available resources
   for (int i = 0; i < R; i++)
```

```
    work[i] = avail[i];

  int count = 0; // Count of finished processes

  // Loop until all processes are finished or system is not in a safe state
  while (count < P)
  {
    // Find an unfinished process whose needs can be satisfied with available
resources
    int found = 0;
    for (int p = 0; p < P; p++)
    {
      if (finish[p] == 0)
      {
        int j;
        for (j = 0; j < R; j++)
          if (need[p][j] > work[j])
            break;

        if (j == R) // All needs of process p are satisfied
        {
          // Add allocated resources of process p to the available resources
          for (int k = 0; k < R; k++)
            work[k] += allot[p][k];

          // Add process p to the safe sequence
          safeSeq[count++] = p;

          // Mark process p as finished
          finish[p] = 1;

          found = 1;
        }
      }
    }
```

```
        // If no process was found in this iteration, the system is not in a safe state
        if (found == 0)
        {
            printf("System is not in a safe state\n");
            return 0;
        }
    }

    // If reached here, the system is in a safe state
    printf("System is in a safe state\nSafe sequence is: ");
    for (int i = 0; i < P; i++)
        printf("%d ", safeSeq[i]);
    printf("\n");
    return 1;
}

// Function to simulate resource allocation
void resourceAllocation(int processes[], int avail[], int max[][R], int allot[][R], int
req[])
{
    // Try allocating resources for the requested process
    for (int i = 0; i < R; i++)
    {
        avail[i] -= req[i];
        allot[req[P]][i] += req[i];
    }

    // Check if the system is in a safe state after allocation
    isSafe(processes, avail, max, allot);
}

int main()
{
    int processes[] = {0, 1, 2, 3, 4}; // Processes
    int avail[] = {3, 3, 2};           // Available instances of resources
    int max[][R] = {{7, 5, 3},         // Maximum resources that can be allocated to
```

```
each process
                {3, 2, 2},
                {9, 0, 2},
                {2, 2, 2},
                {4, 3, 3}};
   int allot[][R] = {{0, 1, 0},        // Resources already allocated to each process
                 {2, 0, 0},
                 {3, 0, 2},
                 {2, 1, 1},
                 {0, 0, 2}};
   int req[P][R] = {{0, 0, 0},         // Resource request for each process
                {0, 2, 0},
                {3, 0, 2},
                {2, 1, 1},
                {0, 0, 2}};

   // Simulate resource allocation for each process
   for (int i = 0; i < P; i++)
   {
      printf("Process %d requests resources: ", i);
      for (int j = 0; j < R; j++)
         printf("%d ", req[i][j]);
      printf("\n");

      resourceAllocation(processes, avail, max, allot, req[i]);
      printf("\n");
   }

   return 0;
}
```

Output:

```
Process 0 requests resources: 0 0 0
System is in a safe state
Safe sequence is: 1 3 4 0 2

Process 1 requests resources: 0 2 0
System is in a safe state
Safe sequence is: 3 1 2 4 0

Process 2 requests resources: 3 0 2
System is not in a safe state

Process 3 requests resources: 2 1 1
System is not in a safe state

Process 4 requests resources: 0 0 2
System is not in a safe state

PS C:\Users\Lenovo\Downloads\AOA FINAL FOLDER>
```

Conclusion:

In conclusion, implementing the Banker's Algorithm to avoid deadlock in concurrent systems is a crucial strategy for ensuring system stability and reliability. By carefully managing resource allocation and dynamically assessing the safety of potential requests, the Banker's Algorithm helps prevent the occurrence of deadlock scenarios. Through the demonstration of this algorithm in our program, we've highlighted the importance of proper resource management and the role of proactive decision-making in maintaining system integrity. As we continue to develop and refine our understanding of concurrency control mechanisms, leveraging techniques like the Banker's Algorithm will remain essential for creating robust and efficient software systems.

When can we say that system is in a safe or unsafe state?

Safe State:

A system is considered to be in a safe state if there exists at least one sequence of processes that can complete their execution and release their resources, without leading to a deadlock.

In a safe state, even if all processes suddenly request their maximum allocation of resources, it is still possible to satisfy those requests without entering a deadlock.

The system can allocate resources to each process in a sequence such that all processes eventually complete execution and release their resources.

Unsafe State:

A system is considered to be in an unsafe state if no sequence of process execution exists that can guarantee deadlock-free execution.

In an unsafe state, the allocation of additional resources to processes may lead to a situation where some processes are unable to proceed due to resource shortages, potentially causing a deadlock.

An unsafe state indicates that allocating resources based on the current resource requests may result in a deadlock.