B. Siva Shirish-192324016

17. Illustrate the deadlock avoidance concept by simulating Banker's algorithm with C.

AIM:

To illustrate the concept of deadlock avoidance by simulating the Banker's Algorithm in C, ensuring system safety by allocating resources only when a safe sequence exists.

ALGORITHM:

1. **Input Data:**
   - Read the number of processes (n) and resources (m).
   - Input the Allocation matrix, Max matrix, and Available resources.
2. **Calculate Need Matrix:**
   - Compute the Need matrix using the formula:
     Need[i][j] = Max[i][j] - Allocation[i][j].
3. **Initialize Variables:**
   - Set Work = Available resources.
   - Set Finish array to false for all processes.
   - Initialize an empty Safe Sequence array.
4. **Find a Process to Allocate:**
   - Search for an unfinished process i such that:
     Need[i][j] <= Work[j] for all j.
5. **Allocate Resources if Safe:**
   - If such a process is found:
     - Add the allocated resources of i to Work:
       Work[j] += Allocation[i][j] for all j.
     - Mark i as finished (Finish[i] = true).
     - Add i to the Safe Sequence array.
6. **Repeat Allocation Check:**
   - Continue steps 4 and 5 until either all processes are finished or no suitable process is found.

7. **Check System State:**
    - If all processes are marked finished, the system is in a **safe state**, and the safe sequence is printed.
    - If not, the system is in an **unsafe state**, and no safe sequence exists.

PROCEDURE:

1. **Start:**
   Initialize variables to store the Allocation matrix, Max matrix, Available resources, and the Need matrix.
2. **Input Data:**
    - Enter the number of processes (n) and resources (m).
    - Input the Allocation matrix, Max matrix, and Available resources.
3. **Calculate Need Matrix:**
   Compute Need[i][j] for each process and resource using the formula:
   Need[i][j] = Max[i][j] - Allocation[i][j].
4. **Initialize Safety Check:**
    - Set Work = Available.
    - Mark all processes in the Finish array as false.
5. **Allocate Resources:**
    - Find an unfinished process i such that Need[i][j] <= Work[j] for all resources j.
    - If found:
        - Add Allocation[i][j] to Work[j] for all j.
        - Mark Finish[i] = true.
        - Add the process to the safe sequence.
6. **Repeat Allocation:**
   Repeat Step 5 until all processes are marked finished or no suitable process is found.
7. **Check System Safety:**
    - If all processes are marked Finish = true, the system is in a safe state. Print the safe sequence.
    - Otherwise, declare the system to be in an unsafe state.

8. **Stop:**
   End the procedure.

CODE:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 10
#define RESOURCES 3

int allocation[MAX][RESOURCES], max[MAX][RESOURCES], need[MAX][RESOURCES];
int available[RESOURCES], safeSequence[MAX], processCount;

bool isSafe() {
   int work[RESOURCES];
   bool finish[MAX] = {0};
   int count = 0;

   for (int i = 0; i < RESOURCES; i++)
      work[i] = available[i];

   while (count < processCount) {
      bool found = false;
      for (int p = 0; p < processCount; p++) {
         if (!finish[p]) {
            int j;
            for (j = 0; j < RESOURCES; j++)
               if (need[p][j] > work[j])
                  break;

            if (j == RESOURCES) {
               for (int k = 0; k < RESOURCES; k++)
                  work[k] += allocation[p][k];
               safeSequence[count++] = p;
               finish[p] = true;
               found = true;
            }
         }
      }
      if (!found)
         return false;
   }
   return true;
}

void requestResources(int processID, int request[]) {
   for (int i = 0; i < RESOURCES; i++) {
      if (request[i] > need[processID][i]) {
         printf("Error: Process has exceeded its maximum claim.\n");
```

```
            return;
        }
    }

    for (int i = 0; i < RESOURCES; i++) {
        if (request[i] > available[i]) {
            printf("Process is waiting for resources.\n");
            return;
        }
    }

    for (int i = 0; i < RESOURCES; i++) {
        available[i] -= request[i];
        allocation[processID][i] += request[i];
        need[processID][i] -= request[i];
    }

    if (isSafe()) {
        printf("Resources allocated successfully.\n");
    } else {
        for (int i = 0; i < RESOURCES; i++) {
            available[i] += request[i];
            allocation[processID][i] -= request[i];
            need[processID][i] += request[i];
        }
        printf("Resources allocation leads to unsafe state. Reverting.\n");
    }
}

int main() {
    processCount = 5;

    int initialAvailable[RESOURCES] = {3, 2, 2};
    for (int i = 0; i < RESOURCES; i++)
        available[i] = initialAvailable[i];

    int maxClaims[MAX][RESOURCES] = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3}
    };

    int allocations[MAX][RESOURCES] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
```

```
        {0, 0, 2}
    };

    for (int i = 0; i < processCount; i++) {
        for (int j = 0; j < RESOURCES; j++) {
            max[i][j] = maxClaims[i][j];
            allocation[i][j] = allocations[i][j];
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }

    int request[RESOURCES] = {1, 0, 2};
    requestResources(1, request);

    return 0;
}
```

OUTPUT: