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

Aim:

The aim of this program is to **illustrate the concept of deadlock avoidance** by simulating **Banker's Algorithm** in C. The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm that checks whether the system is in a safe state before granting resources to a process.

Procedure:

1. **Define the Data Structures**: The system needs several matrices and vectors:
   - Allocation[][]: Keeps track of how many resources are currently allocated to each process.
   - Max[][]: Represents the maximum resources each process may need.
   - Available[]: Represents the available resources in the system.
   - Need[][]: Represents the remaining resources each process may need (calculated as Need[i][j] = Max[i][j] - Allocation[i][j]).
2. **Safety Check**: The algorithm checks if granting the request for resources leaves the system in a **safe state**. This is done by simulating the allocation and verifying if all processes can eventually finish with the available resources.
3. **Granting Resources**: Before granting a resource request, the system checks if the request is **less than or equal to the need of the process** and if the request is **less than or equal to the available resources**.
4. **Simulation**:
   - The algorithm simulates whether the system can proceed with all processes completing successfully (safe state).
   - If the system is in a safe state, the request is granted. Otherwise, the request is denied.

Banker's Algorithm C Program:

c

Copy code

```c
#include <stdio.h>

#include <stdbool.h>
```

```
// Number of processes and resources

#define P 5

#define R 3


// Function to calculate if the system is in a safe state

bool isSafeState(int processes[], int avail[], int max[][R], int allot[][R]) {

    int work[R];

    bool finish[P];


    // Initialize work[] and finish[]

    for (int i = 0; i < R; i++) {

        work[i] = avail[i];

    }


    for (int i = 0; i < P; i++) {

        finish[i] = false;

    }


    int count = 0;

    while (count < P) {

        bool progressMade = false;


        for (int p = 0; p < P; p++) {

            // Check if process p is not finished and its needs can be satisfied with current available
resources

            if (!finish[p]) {

                bool canAllocate = true;

                for (int r = 0; r < R; r++) {
```

```
                if (max[p][r] - allot[p][r] > work[r]) {
                    canAllocate = false;
                    break;
                }
            }

            if (canAllocate) {
                // Add the allocated resources of process p to work[]
                for (int r = 0; r < R; r++) {
                    work[r] += allot[p][r];
                }
                finish[p] = true;
                count++;
                progressMade = true;
            }
        }
    }

    // If no process can be allocated, break out
    if (!progressMade) {
        return false; // The system is not in a safe state
    }
  }
  return true; // The system is in a safe state
}

bool requestResources(int processes[], int avail[], int max[][R], int allot[][R], int request[], int pid) {
```

```
// Check if request is valid (request <= need)
for (int i = 0; i < R; i++) {
    if (request[i] > max[pid][i] - allot[pid][i]) {
        printf("Error: Process has exceeded its maximum claim!\n");
        return false;
    }
}

// Check if request is less than or equal to available resources
for (int i = 0; i < R; i++) {
    if (request[i] > avail[i]) {
        printf("Resources are not available!\n");
        return false;
    }
}

// Pretend to allocate the resources to process pid
for (int i = 0; i < R; i++) {
    avail[i] -= request[i];
    allot[pid][i] += request[i];
}

// Check if the system is in a safe state
if (isSafeState(processes, avail, max, allot)) {
    printf("Request can be granted safely.\n");
    return true;
} else {
```

```
        // Rollback the allocation
        for (int i = 0; i < R; i++) {
            avail[i] += request[i];
            allot[pid][i] -= request[i];
        }
        printf("Request cannot be granted safely.\n");
        return false;
    }
}

int main() {
    // Initialize the processes, available resources, maximum resources and allocation
    int processes[] = {0, 1, 2, 3, 4};

    int avail[] = {3, 3, 2}; // Available resources

    // Maximum resources needed by each process
    int max[][R] = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3}
    };

    // Resources allocated to each process
    int allot[][R] = {
```

```
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
    };

    int pid, request[R];

    // Request resources for process
    printf("Enter process ID for resource request (0-4): ");
    scanf("%d", &pid);

    printf("Enter request for resources (format: Request1 Request2 Request3): ");
    for (int i = 0; i < R; i++) {
        scanf("%d", &request[i]);
    }

    // Try to allocate resources
    if (requestResources(processes, avail, max, allot, request, pid)) {
        printf("Resources allocated successfully.\n");
    } else {
        printf("Request denied.\n");
    }

    return 0;
}
```

Output:

Output

Clear

```
Enter process ID for resource request (0-4): 192372048
Enter request for resources (format: Request1 Request2 Request3): 1 0 2
Resources are not available!
Request denied.

Enter process ID for resource request (0-4): 3
Enter request for resources (format: Request1 Request2 Request3): 1 1 0
Request can be granted safely.
Resources allocated successfully.
```