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:

It is a banker algorithm used to avoid deadlock and allocate resources safely to each process in the computer system. The 'S-State' examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes. The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources. In this section, we will learn the Banker's Algorithm in detail. Also, we will solve problems based on the Banker's Algorithm. To understand the Banker's Algorithm first we will see a real word example of it.

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system.

Similarly, it works in an operating system. When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as deadlock avoidance algorithm or deadlock detection in the operating system.

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.

Resource-Request Algorithm for Process Pi

Request i = request vector for process Pi . If Requesti [j] = k then process Pi wants k instances of resource type Rj
1. If Requesti ≤ Needi go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available 3. Pretend to allocate requested resources to Pi by modifying the state as follows:
Available = Available – Requesti ;
Allocationi = Allocationi + Requesti ;
Needi = Needi – Requesti ;
1.If safe ⇒ the resources are allocated to Pi
2. If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored.

Program:

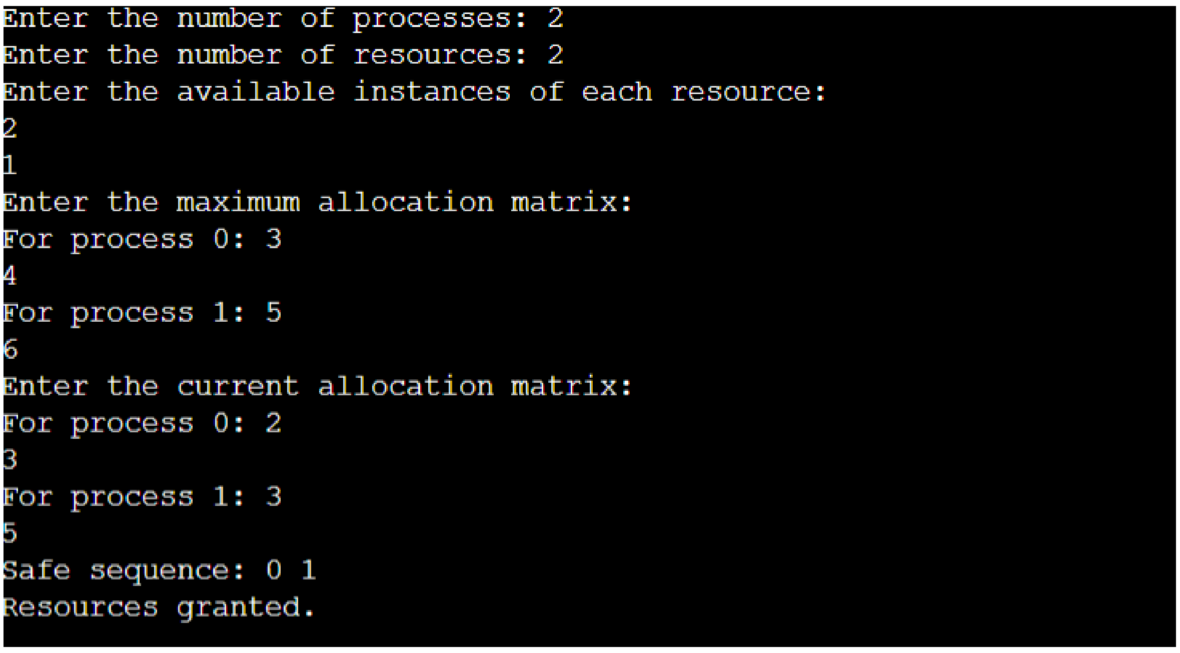
```
#include <stdio.h>

#include <stdbool.h>

#define MAX_PROCESSES 10

#define MAX_RESOURCES 10

int available[MAX_RESOURCES];

int max_allocation[MAX_PROCESSES][MAX_RESOURCES];

int current_allocation[MAX_PROCESSES][MAX_RESOURCES];

int need[MAX_PROCESSES][MAX_RESOURCES];

bool finished[MAX_PROCESSES];

int num_processes, num_resources;

// Function to check if the requested resources can be allocated

bool isSafe(int process_id, int request[]) {

   // Check if the requested resources exceed the need

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

      if (request[i] > need[process_id][i])
```

```
            return false;
    }
    // Check if the requested resources are available
    for (int i = 0; i < num_resources; i++) {
        if (request[i] > available[i])
            return false;
    }
    return true;
}
// Function to allocate resources to a process
void allocateResources(int process_id, int request[]) {
    for (int i = 0; i < num_resources; i++) {
        available[i] -= request[i];
        current_allocation[process_id][i] += request[i];
        need[process_id][i] -= request[i];
    }
}

// Function to release resources from a process
void releaseResources(int process_id, int request[]) {
    for (int i = 0; i < num_resources; i++) {
        available[i] += request[i];
        current_allocation[process_id][i] -= request[i];
        need[process_id][i] += request[i];
    }
}

// Banker's algorithm for deadlock avoidance
bool bankerAlgorithm() {
    bool safe_sequence_exists = true;
    int work[MAX_RESOURCES];

    // Initialize work and finished arrays
    for (int i = 0; i < num_resources; i++) {
```

```
    work[i] = available[i];
}
for (int i = 0; i < num_processes; i++) {
    finished[i] = false;
}

int num_finished = 0;
int safe_sequence[num_processes];

while (num_finished < num_processes) {
    bool found = false;

    for (int i = 0; i < num_processes; i++) {
        if (!finished[i] && isSafe(i, need[i])) {
            found = true;
            safe_sequence[num_finished++] = i;
            finished[i] = true;
            releaseResources(i, current_allocation[i]);

            // Update work
            for (int j = 0; j < num_resources; j++) {
                work[j] += current_allocation[i][j];
            }

            break;
        }
    }

    if (!found) {
        safe_sequence_exists = false;
        break;
    }
}
```

```
    if (safe_sequence_exists) {
        printf("Safe sequence: ");
        for (int i = 0; i < num_processes; i++) {
            printf("%d ", safe_sequence[i]);
        }
        printf("\n");
    } else {
        printf("No safe sequence exists.\n");
    }

    return safe_sequence_exists;
}

int main() {
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);

    printf("Enter the number of resources: ");
    scanf("%d", &num_resources);

    printf("Enter the available instances of each resource:\n");
    for (int i = 0; i < num_resources; i++) {
        scanf("%d", &available[i]);
    }

    printf("Enter the maximum allocation matrix:\n");
    for (int i = 0; i < num_processes; i++) {
        printf("For process %d: ", i);
        for (int j = 0; j < num_resources; j++) {
            scanf("%d", &max_allocation[i][j]);
        }
    }

    printf("Enter the current allocation matrix:\n");
    for (int i = 0; i < num_processes; i++) {
        printf("For process %d: ", i);
        for (int j = 0; j < num_resources; j++) {
            scanf("%d", &current_allocation[i][j]);
            need[i][j] = max_allocation[i][j] - current_allocation[i][j];
        }
    }

    if (bankerAlgorithm()) {
        printf("Resources granted.\n");
    } else {
        printf("Resources cannot be granted due to possible deadlock.\n");
    }

    return 0;
}
```

Output:



```
Enter the number of processes: 2
Enter the number of resources: 2
Enter the available instances of each resource:
2
1
Enter the maximum allocation matrix:
For process 0: 3
4
For process 1: 5
6
Enter the current allocation matrix:
For process 0: 2
3
For process 1: 3
5
Safe sequence: 0 1
Resources granted.
```

Conclusion:

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

Determining whether the system is in a safe or unsafe state often involves analyzing the current allocation and request status of resources, considering potential resource allocation sequences, and evaluating whether any sequence leads to deadlock. Techniques like the Banker's Algorithm are used to assess the safety of resource allocation in the system.