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.

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

ν Max: n x m matrix.

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

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

ν 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>

int main()

{

    // P0, P1, P2, P3, P4 are the Process names here
```

```
int n, m, i, j, k;
        n = 5;
        m = 3; //
        int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
                                                { 2, 0, 0 }, // P1
                                                { 3, 0, 2 }, // P2
                                                { 2, 1, 1 }, // P3
                                                { 0, 0, 2 } }; // P4
        int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
                                        { 3, 2, 2 }, // P1
                                        { 9, 0, 2 }, // P2
                                        { 2, 2, 2 }, // P3
                                        { 4, 3, 3 } }; // P4
        int avail[3] = { 3, 3, 2 }; // Available Resources
        int f[n], ans[n], ind = 0;
        for (k = 0; k < n; k++) {
                f[k] = 0; }
        int need[n][m];
        for (i = 0; i < n; i++) {
                for (j = 0; j < m; j++)
                        need[i][j] = max[i][j] - alloc[i][j]; }
        int y = 0;
        for (k = 0; k < 5; k++) {
```

```
            for (i = 0; i < n; i++) {
                if (f[i] == 0) {
                    int flag = 0;
                    for (j = 0; j < m; j++) {
                        if (need[i][j] > avail[j]){
                            flag = 1;
                            break; } }
                    if (flag == 0) {
                        ans[ind++] = i;
                        for (y = 0; y < m; y++)
                            avail[y] += alloc[i][y];
                        f[i] = 1; } } } }
    int flag = 1;
    for(int i=0;i<n;i++) {
    if(f[i]==0) {
            flag=0;
            printf("The following system is not safe");
            break; } }
    if(flag==1) {
    printf("Following is the SAFE Sequence\n");
    for (i = 0; i < n - 1; i++)
            printf(" P%d ->", ans[i]);
    printf(" P%d", ans[n - 1]);
```

```
        }

        return (0);

}
```

Output:

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```

Conclusion:

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

We can determine whether a computer system is in a safe or unsafe state primarily in the context of concurrent execution and resource allocation. In a safe state, the system can ensure that all processes can eventually complete their execution without entering a deadlock, where processes are indefinitely blocked waiting for resources held by others. Additionally, a safe state guarantees that processes can proceed without violating the system's integrity or causing unexpected failures.

Overall, a system is considered safe when it can maintain stability, integrity, availability, and security while efficiently managing resources and accommodating concurrent execution. Unsafe conditions arise when the system fails to meet these criteria, leading to deadlocks, resource contention, data corruption, failures, or security breaches. Therefore, ensuring system safety requires careful design, implementation, and management of the system's resources, processes, and security measures