**Vidyavardhini's College of Engineering & Technology**
Department of Computer Engineering
Academic Year : 2023-24

## EXPERIMENT NO 7

| |
|---|
| Experiment No 7: Deadlock avoidances |
| Date of Performance:14/03/23 |
| Date of Submission:27/03/24 |
| Name: Jenil Kotadia |
| Roll No : 19 |

AIM: IMPLEMENT DEADLOCK DETECTION AND AVOIDANCES USING BANKERS ALGORITHM

OBJECTIVE :Deadlock detection and avoidance using the Banker's algorithm is a crucial concept in operating systems and concurrent programming. The Banker's algorithm is primarily used in resource allocation scenarios to prevent deadlock by ensuring that the system never enters an unsafe state. Here's a Python implementation demonstrating deadlock detection and avoidance using the Banker's algorithm

THEORY :

Deadlock detection involves periodically checking the system's state to determine if a deadlock has occurred. This is typically done by constructing a resource allocation graph and looking for cycles within it. If a cycle is found, it indicates that deadlock has occurred. However, detecting deadlock alone is not sufficient; we also need to resolve it.

## SAFETY ALGORITHM:

The safety algorithm is a key component of the Banker's algorithm for deadlock avoidance. Its purpose is to determine whether the system is in a safe state before allocating resources to a process. A safe state is a state in which the system can allocate resources to processes in some order and still avoid deadlock.

The safety algorithm is a crucial part of the Banker's algorithm for deadlock avoidance. It checks whether the system is in a safe state before allocating resources to a process. This is done by simulating the allocation process and ensuring that resources can be allocated in a way that doesn't lead to deadlock. If the system is in a safe state, resources are allocated; otherwise, the allocation is denied to prevent deadlock

## ADVANTAGE :

The safety algorithm, a key component of the Banker's algorithm for deadlock avoidance, functions by evaluating whether the system is in a state where resources can be allocated to processes without the risk of deadlock. It operates by simulating the allocation of resources and ensuring that at any given point, there exists a safe sequence of processes that can complete their execution without leading to deadlock.

This algorithm iterates through each process, attempting to find one that can have its resource needs met based on the available resources. If such a process is found, its resources are temporarily allocated, and the algorithm checks if the system remains in a safe state. If the system can continue to find a safe sequence of processes until all processes are accommodated or until no more processes can be satisfied, the system is deemed to be in a safe state, and resource allocation can proceed.

## CODE:

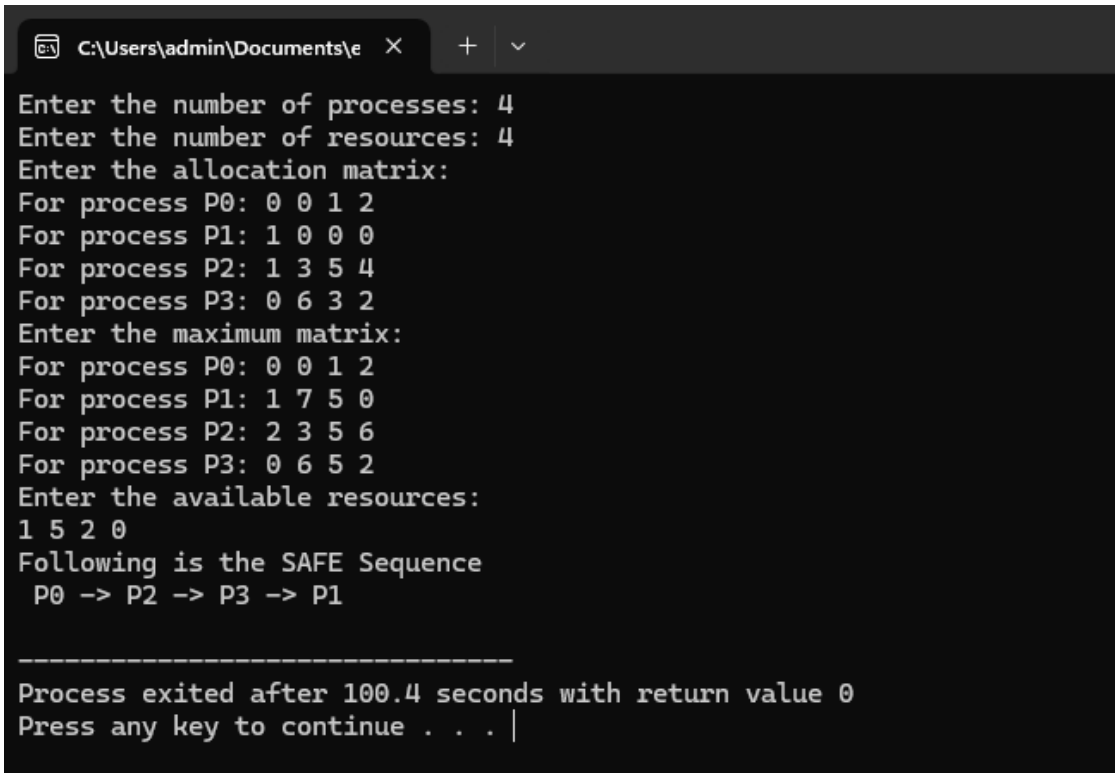
```
#include <stdio.h>
```

```
int main()  {
int n, m, i, j, k;

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

int alloc[n][m], max[n][m], avail[m];

printf("Enter the allocation matrix:\n");
for (i = 0; i < n; i++) {
    printf("For process P%d: ", i);
    for (j = 0; j < m; j++) {
        scanf("%d", &alloc[i][j]);
    }
}

printf("Enter the maximum matrix:\n");
for (i = 0; i < n; i++) {
    printf("For process P%d: ", i);
    for (j = 0; j < m; j++) {
        scanf("%d", &max[i][j]);
    }
}

printf("Enter the available resources:\n");
for (j = 0; j < m; j++) {
    scanf("%d", &avail[j]);
```

```
}

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 < n; 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\n");
        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\n", ans[n - 1]);
}


return 0;
}
```

OUTPUT:




## CONCLUSION :

The safety algorithm plays a critical role within the Banker's algorithm framework for deadlock avoidance. By rigorously assessing the system's state and ensuring that resources are allocated in a manner that preserves safety, it effectively prevents deadlock scenarios from arising. This proactive approach not only safeguards against system instability but also promotes efficient resource utilization and adaptability to dynamic resource demands. Thus, the safety algorithm stands as a fundamental mechanism for maintaining system stability and reliability in the realm of operating systems and concurrent programming.