Vansh Jalora
231070074
SY CE BTech
Batch-D

Operating Systems — Lab 7

AIM:

To implement the Banker's Algorithm to determine whether a process is in its Safe State or not, and to avoid Deadlocks.

Theory

Key Components in Banker's Algorithm:

1. **Processes**: These are the programs running in the system that require resources (like CPU, memory, etc.).
2. **Resources**: There are multiple types of resources, and each process may require a different number of these resources at any given time.
3. **Safe State**: A system is in a safe state if it is possible to allocate resources to all processes in some order without causing a deadlock.
4. **Unsafe State**: If the system grants a resource request and there's a chance that a deadlock would occur later, the system is in an unsafe state.

Components of the Banker's Algorithm:

1. **Available**: This is a vector of size "m" (where m is the number of resource types). It indicates the number of available instances of each resource type.
2. **Max**: This is an "n x m" matrix (where n is the number of processes and `m` is the number of resource types). It indicates the maximum demand of each process for each resource.
3. **Allocation**: This is an "n x m" matrix that indicates how many instances of each resource are currently allocated to each process.

4. **Need**: This is an “n x m” matrix calculated from “Max - Allocation”, representing how many more instances of each resource each process still needs in order to complete its execution.

Advantages:

- **Deadlock avoidance**: The Banker's Algorithm helps in preventing deadlocks by ensuring that the system always remains in a safe state.
- **Fair resource allocation**: It ensures that no process is starved, as long as the system can allocate resources safely.

Disadvantages:

- **High overhead**: Checking the safety state for every request is computationally expensive, especially in large systems.
- **Static maximum needs**: The algorithm requires knowledge of maximum resource requirements ahead of time, which may not always be feasible in dynamic environments.

Algorithm

```
Algo
Initialise:
 Process vector for n processes.
Allocation[n][m] matrix for resources allocated to each
process of m resources
Max[n][m] - max resources can be allocated to each
process.
Need[n][m] = Max[n][m] - Allocation[n][m]
// Remaining needs for each process
Initialise Work and Finish as vectors of length m and n.
Work=Available
Finish[n]={false, false, false,false......}

While false in Finish[]:
     Progress=false
For i in Process:
     If Finish[i]=true and Need[i]<Work:
     progress=true
     work +=allocation
     Set Finish[i]=true
     If progress==false:
         Break //prevents infinite loop
     If all elements in Finish are true:
         return safe state
     Else:
         return not safe state
```

COMPLETE PROGRAM

```
#include <bits/stdc++.h>
using namespace std;

void bankers(){
    int p, r;
    cout<<"Enter number of process and no of resources:";
    cin>>p>>r;
    int Max[p][r];
    int Allocation[p][r];
    int Need[p][r];
    int Available[r];
    int Finished[p];

    //Input Max Matrix
    cout<<"Max:"<<endl;
    cout<<"Process\t";
    for (int i = 0; i < r; i++){
        cout<<(char)(65+i)<<" ";
    }
    cout<<endl;
    for (int i = 0; i < p; i++){
        cout<<"P"<<i<<"\t";
        for (int j = 0; j < r; j++){
            cin>>Max[i][j];
        }
    }

    //Input Allocation Matrix
    cout<<"Allocation:"<<endl;
    cout<<"Process\t";
    for (int i = 0; i < r; i++){
        cout<<(char)(65+i)<<" ";
    }
    cout<<endl;
    for (int i = 0; i < p; i++){
        cout<<"P"<<i<<"\t";
        for (int j = 0; j < r; j++){
```

```
            cin>>Allocation[i][j];
        }
    }

    //Input Available Vector
    cout<<"Available:"<<endl;
    for (int i = 0; i < r; i++){
        cout<<(char)(65+i)<<" ";
    }
    cout<<endl;
    for (int j = 0; j < r; j++){
        cin>>Available[j];
    }

    //Calculate Need Matrix
    //Need = Max - Allocation
    for (int i = 0; i < p; i++){
        for (int j = 0; j < r; j++){
            Need[i][j] = Max[i][j] - Allocation[i][j];
        }
    }

    //Print Need Matrix
    cout<<"Need:"<<endl;
    cout<<"Process\t";
    for (int i = 0; i < r; i++){
        cout<<(char)(65+i)<<" ";
    }
    cout<<endl;
    for (int i = 0; i < p; i++){
        cout<<"P"<<i<<"\t";
        for (int j = 0; j < r; j++){
            cout<<Need[i][j]<<" ";
        }
        cout<<endl;
    }

    //Initialize Finished Matrix
    for (int i = 0; i < p; i++){
        Finished[i] = false;
    }

    int safe_count = 0;
```

```
//Find which process can happen if Available >= Need
bool safe = true;
for (int k = 0; k < p; k++) {
    bool executed = false;

    for (int i = 0; i < p; i++) {
        if (Finished[i]) {
            continue;
        }

        bool can_execute = true;
        for (int j = 0; j < r; j++) {
            if (Need[i][j] > Available[j]) {
                can_execute = false;
                break;
            }
        }

        if (can_execute) {
            for (int j = 0; j < r; j++) {
                Available[j] += Allocation[i][j];
            }
            Finished[i] = true;
            cout << "Process P" << i << " finished" << endl;
            executed = true;
        }
    }

    if (!executed) {
        safe = false;
        break;
    }
}

for (int i = 0; i < p; i++){
    safe = Finished[i];
    if (!safe){
        cout<<"System is not in safe state"<<endl;
        break;
    }
}
if (safe){
```

```
        cout<<"System is in safe state"<<endl;
    }

}

int main(void){
    bankers();
}
```

OUTPUT

For Safe State System:

```
C:\Users\Vansh_Prac\SY_Prac\Python\OS-Lab\Lab7>cd "c:\Users\Vansh_Prac\SY_Prac\Python\OS-Lab\Lab7\" && g++ bankers.cpp -o bankers && "c:\Users\Va
nsh_Prac\SY_Prac\Python\OS-Lab\Lab7\"bankers
Enter number of process and no of resources:5 3
Max:
Process A B C
P0      7 5 3
P1      3 2 2
P2      9 0 2
P3      2 2 2
P4      4 3 3
Allocation:
Process A B C
P0      0 1 0
P1      2 0 0
P2      3 0 2
P3      2 1 1
P4      0 0 2
Available:
A B C
3 3 2
Need:
Process A B C
P0      7 4 3
P1      1 2 2
P2      6 0 0
P3      0 1 1
P4      4 3 1
Process P1 finished
Process P3 finished
Process P4 finished
Process P0 finished
Process P2 finished
System is in safe state
```

For System not in Safe State:

```
C:\Users\Vansh_Prac\SY_Prac\Python\OS-Lab\Lab7>cd "c:\Users\Vansh_Prac\SY_Prac\Python\OS-Lab\Lab7\" && g++ bankers.cpp -o bankers && "c:\Users\Va
nsh_Prac\SY_Prac\Python\OS-Lab\Lab7\"bankers
Enter number of process and no of resources:5 3
Max:
Process A B C
P0      7 5 3
P1      3 2 2
P2      9 0 2
P3      4 2 2
P4      5 3 3
Allocation:
Process A B C
P0      0 1 0
P1      2 0 0
P2      3 0 2
P3      2 1 1
P4      0 0 2
Available:
A B C
2 1 0
Need:
Process A B C
P0      7 4 3
P1      1 2 2
P2      6 0 0
P3      2 1 1
P4      5 3 1
System is not in safe state
```

Conclusion:

The implementation of the Banker's Algorithm successfully demonstrates its ability to prevent deadlock by ensuring that processes are always allocated resources in a safe sequence. By continuously evaluating resource requests against the system’s current state, the algorithm provides a reliable method to determine whether the system can proceed without entering an unsafe or deadlock state. This approach proves to be effective in managing shared resources in multi-process systems, ensuring both system safety and optimal resource utilization. Despite its computational complexity, the Banker's Algorithm remains a crucial technique for deadlock avoidance in resource management.