OS MINI PROJECT REPORT

Bankers Algorithm for deadlock avoidance

Batch - C
Computer Engineering

Shardul Khade
211070078
sdkhade_b21@ce.vjti.ac.in

Preet Jain
211070081
ppjain_b21@ce.vjti.ac.in

Abstract:

The Banker's algorithm is a deadlock avoidance algorithm used in operating systems to prevent deadlocks from occurring. It was first proposed by Dijkstra in 1965 and has since been widely used in various operating systems.

The Banker's algorithm is based on the concept of resource allocation. It assumes that each process in the system has a maximum demand for each resource and that the system has a fixed number of resources of each type. The algorithm works by checking if a particular allocation request from a process can be granted without causing a deadlock.

The algorithm maintains a data structure called the "resource allocation graph," which keeps track of which processes are holding which resources and which processes are waiting for which resources. Whenever a process requests a resource, the algorithm checks if granting the request will result in a safe state, i.e., a state in which no deadlocks can occur.

To determine whether a state is safe, the Banker's algorithm uses the concept of "safe sequence." A safe sequence is a sequence of processes that can complete their execution without causing a deadlock. The algorithm starts by assuming that no process is holding any resources and then checks if there is a safe sequence that can be executed. If a safe sequence exists, the algorithm grants the resource request and updates the resource allocation graph accordingly. If a safe sequence does not exist,

the algorithm denies the resource request and puts the requesting process in a waiting state.

Overall, the Banker's algorithm is an effective way to prevent deadlocks from occurring in operating systems. By using the concept of safe sequences, it can ensure that resources are allocated in a way that guarantees the absence of deadlocks, thus increasing the overall reliability and stability of the system.

INTRODUCTION :

The Banker's algorithm is a deadlock avoidance algorithm used in operating systems. It's a resource allocation and scheduling algorithm that's used to test whether a system is in a safe state or not. The algorithm is designed to ensure that the system avoids deadlocks by carefully tracking the resources available and resources needed by the processes.

IMPLEMENTATION :

- The given code implements the Banker's algorithm in C++ language. The code takes input from the user, such as the number of users and the number of resources, available resources, resource allocation, and maximum allocation. The code then calculates the need of each user by subtracting the allocation from the maximum allocation. Then, the code checks whether the current state is safe or not by simulating the allocation and releasing the resources of each process one by one until all processes are completed. If the system can allocate resources to all processes safely, then it returns the safe sequence. Otherwise, it returns deadlock.

- The code also provides an option to the user to display the current state of the system, and it also allows the user to make additional requests. The user can enter the process ID and the request. If the system can allocate the resources safely, then the request is accepted, and the system checks for deadlock again. If the system cannot allocate the resources safely, the request is rejected.

CODE :

Bankers_algo.cpp :

```
#include<bits/stdc++.h>
#include<iostream>

using namespace std;
/**
 *
 *
 * We have allocation and need and available
 *
 * GOAL0 - iterate and determine whether safe sequence or not.
 *
 * GOAL1 - Include additional requests.
 *
 *
 */
int seq[100];
bool isSafe(int n,int m,vector<vector<int>> allocation,
vector<vector<int>> need, vector<vector<int>> max_alloc, vector<int>
avail)
{
vector<bool> isFinished (n,false);
int flag=0,i=0;

while(i<n){
    int c=0;
   // cout<<isFinished[i]<<endl;
if(isFinished[i]==false)
 {
    for (int  j = 0; j< m; j++)
 {
    if( need[i][j]<=avail[j]){
        c++;
    }
 }
 if(c==m){
    //release allocated resources and isFinished[i]=True;
```

```
   c=0;

     for (int  j = 0; j< m; j++)
 {
    avail[j]+= allocation[i][j];
 }
    isFinished[i]=true;
    seq[flag++]=i;
      i=0;
 }
 else{
    i++;
}
}else{
    i++;
}
//cout<< "here \n";
}
if(flag!=n){

    std::cout<< "\n++++++++  Deadlock occurred|Unsafe State
++++++++\n";
return false;
}else{
   std:: cout <<"\nNo Deadlock\nSafe Sequence  : ";
    for (int i = 0; i < n; i++)
    {
       cout << seq[i]<<" ";

    }cout <<endl;

}
return true;

}
```

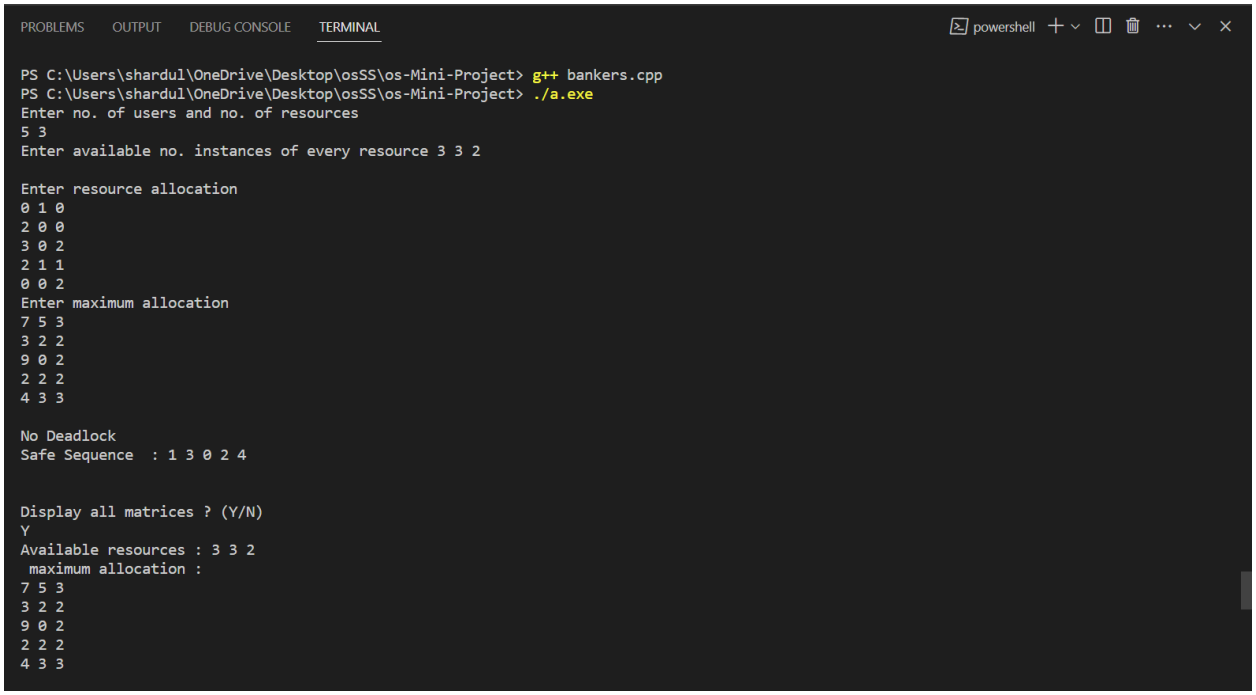
```
void display(int n,int m,vector<vector<int>> allocation,
vector<vector<int>> need, vector<vector<int>> max_alloc, vector<int>
avail){
    cout << "Available resources : ";
    for(int i=0;i<m;i++){
        cout<< avail[i] <<" ";
    }
    cout<<endl;

    cout<< " maximum allocation : \n";
for(int i=0;i<n;i++){
    for (int j = 0; j < m; j++)
    {
        cout<<max_alloc[i][j]<<" ";

    }cout<<endl;
}cout<<endl;

cout<< "Allocation : \n";
for(int i=0;i<n;i++){
    for (int j = 0; j < m; j++)
    {
       cout<<allocation[i][j]<<" ";

    }cout<<endl;
}
cout<<endl;
cout<< "Need : \n";
for(int i=0;i<n;i++){
    for (int j = 0; j < m; j++)
    {
       cout<<need[i][j]<<" ";

    }cout<<endl;
}
cout<<endl;

}
```

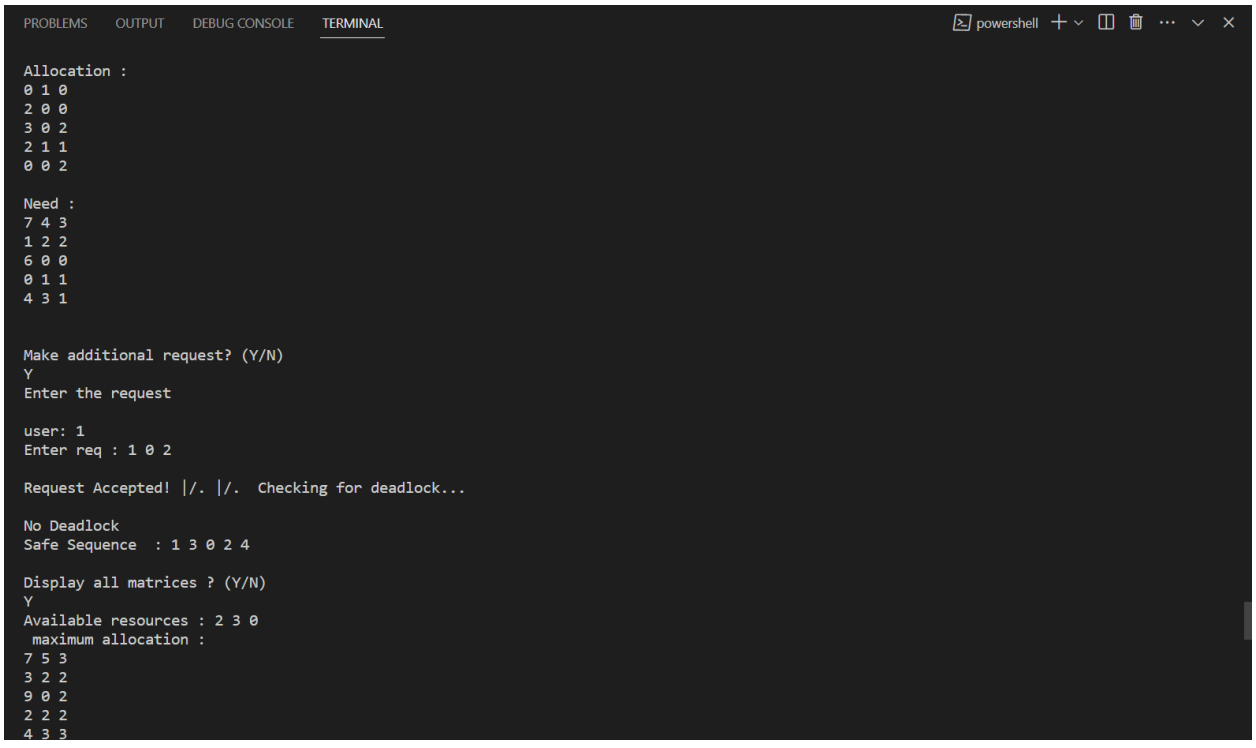
```
    void calc_need(int n,int m ,vector<vector<int>>
allocation,vector<vector<int>>& need,vector<vector<int>> max_alloc )
{
    for(int i=0;i<n;i++)
    {

        for (int j = 0; j < m; j++)
        {

             need[i][j] = max_alloc[i][j]-allocation[i][j];

        }
    }
}

int main(){
int n , m ;
cout<<"Enter no. of users and no. of resources \n";
cin>> n>>m;
vector<vector<int>> allocation (n,vector<int> (m,0));
vector<vector<int>> need (n,vector<int> (m,0));
vector<vector<int>> max_alloc (n,vector<int> (m,0));
vector<int>avail(m,0);
//input resources available
cout<< "Enter available no. instances of every resource ";
for(int i=0;i<m;i++){
 cin>> avail[i];
}
cout <<endl;
// input allocation
cout<< "Enter resource allocation\n";
for(int i=0;i<n;i++){
    for (int j = 0; j < m; j++)
    {
        cin>>allocation[i][j];

    }
```

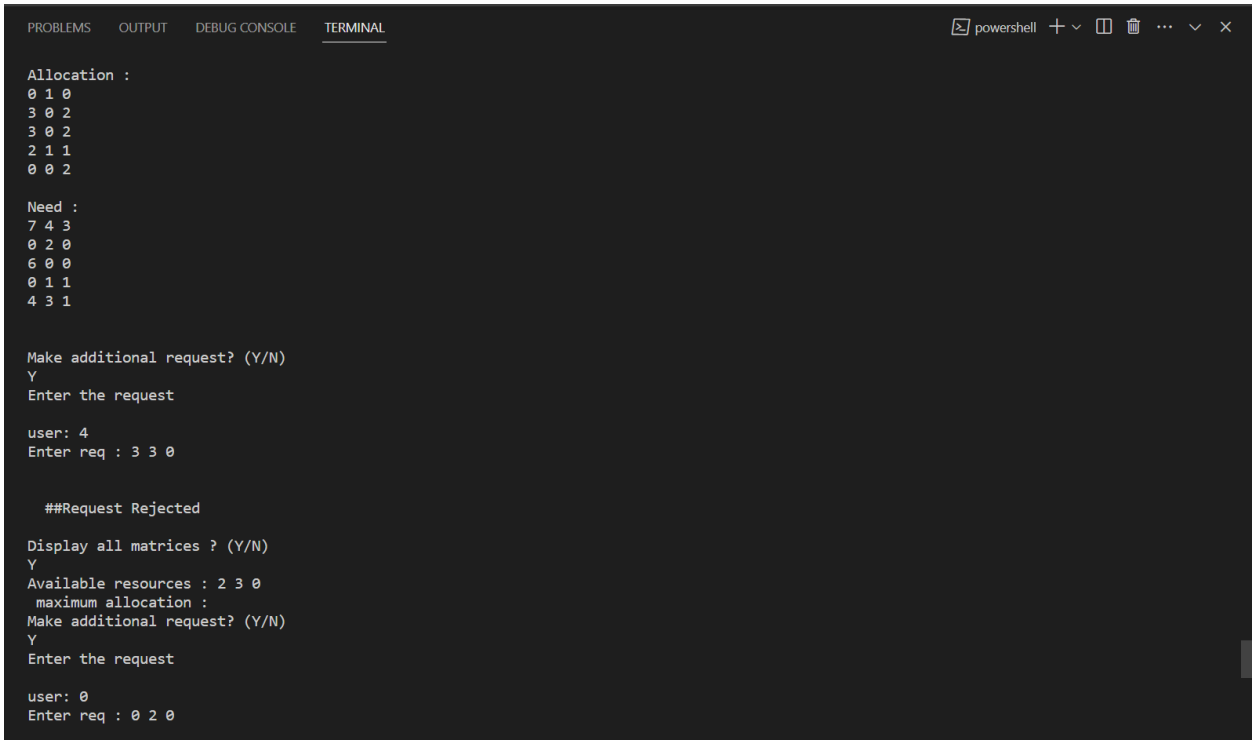
```
}

// input max
cout<< "Enter maximum allocation\n";
for(int i=0;i<n;i++){
    for (int j = 0; j < m; j++)
    {
        cin>>max_alloc[i][j];

    }
}
//calculate need
calc_need(n,m,allocation,need,max_alloc);
bool res = isSafe(n, m, allocation,  need,  max_alloc,  avail);
 cout<<endl;

while(true){
     cout <<"\nDisplay all matrices ? (Y/N)" <<endl;
 char opt;
 cin>>opt;
 if(opt=='Y')display( n, m, allocation,  need,  max_alloc, avail);
cout <<"\nMake additional request? (Y/N)" <<endl;
 char opt1;
 cin>>opt1;
 if(opt1=='Y')
 {


cout<< "Enter the request \n";
int req[m];
cout << "\nuser: ";
int user, lock=0;
cin>>user;
cout <<"Enter req : ";
for(int k =0 ; k<m ; k++){
    cin>> req[k];

}
```

```
cout <<endl;

for(int k =0 ; k<m ; k++ ){
   if( allocation[user][k]+req[k]>max_alloc[user][k] ||
avail[k]-req[k]<0){
    lock=1;
    break;
   }


}
if(!lock){
    for(int k =0 ; k<m ; k++ ){
    allocation[user][k]+=req[k];
    avail[k]-=req[k];
}
    calc_need(n,m,allocation,need,max_alloc);
    cout << "Request Accepted! |/. |/.  Checking for deadlock...\n";
if(!isSafe(n, m, allocation,  need,  max_alloc,  avail)){
break;
}

 }else {
 cout << "\n  ##Request Rejected\n";

 }

 }
 else{
    break;
 }
}
    return 0;
}
```

Output Snapshots:


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
powershell
PS C:\Users\shardul\OneDrive\Desktop\osSS\os-Mini-Project> g++ bankers.cpp
PS C:\Users\shardul\OneDrive\Desktop\osSS\os-Mini-Project> ./a.exe
Enter no. of users and no. of resources
5 3
Enter available no. instances of every resource 3 3 2
Enter resource allocation
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter maximum allocation
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
No Deadlock
Safe Sequence : 1 3 0 2 4
Display all matrices ? (Y/N)
Y
Available resources : 3 3 2
maximum allocation :
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
powershell
Allocation :
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Need :
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1
Make additional request? (Y/N)
Y
Enter the request
user: 1
Enter req : 1 0 2
Request Accepted! |/. |/. Checking for deadlock...
No Deadlock
Safe Sequence : 1 3 0 2 4
Display all matrices ? (Y/N)
Y
Available resources : 2 3 0
maximum allocation :
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3



EXPLANATION :

1. The program starts by including the required libraries for input/output operations and defining the namespace.
2. A global array seq is defined for storing the safe sequence.
3. A function isSafe is defined that takes six arguments: the number of users n, the number of resources m, the allocation matrix allocation, the need matrix need, the maximum allocation matrix max_alloc, and the available resources vector avail.
4. The function initializes a boolean vector isFinished of size n with false values and a flag variable flag with value 0.
5. The function enters a while loop that will iterate until i becomes equal to n.
6. The loop checks if the process i has not finished, i.e., isFinished[i] is false. If the process has not finished, the loop checks if it can be executed by checking if the need of the process is less than or equal to the available resources. If the condition is satisfied, the loop releases the allocated resources, marks the

process as finished, updates the available resources, stores the process number in the seq array, and sets i to 0 to start checking all processes again. If the condition is not satisfied, the loop increments i.

7. If the process has finished, the loop increments i.
8. If the loop has iterated over all processes, and the flag variable flag is not equal to n, it means the system is in a deadlock. Otherwise, the safe sequence is printed.
9. The function display is defined, which takes the same arguments as the isSafe function. This function is used to display the available resources, the maximum allocation matrix, the allocation matrix, and the need matrix.
10. The main function is defined.
11. The main function prompts the user to enter the number of users and the number of resources.
12. The function creates four 2D vectors of size nxm: allocation, need, max_alloc, and avail.
13. The user is prompted to enter the available resources.
14. The user is prompted to enter the allocation matrix.
15. The user is prompted to enter the maximum allocation matrix.
16. The need matrix is calculated by subtracting the allocation matrix from the maximum allocation matrix.
17. The isSafe function is called to check if the initial state is in a safe state.
18. The user is prompted to decide whether to display the matrices or not.
19. If the user chooses to display the matrices, the display function is called.
20. The user is prompted to decide whether to make an additional request or not.
21. If the user chooses to make an additional request, the user is prompted to enter the request for a specific user.
22. The request is added to the allocation matrix, and the available resources are updated.
23. The isSafe function is called again to check if the system is still in a safe state.
24. If the request is accepted, the function displays the safe sequence.
25. If the request is rejected, the function prints a message saying that the request has been rejected.

CONCLUSION :

- The Banker's algorithm is a very useful algorithm that ensures the system avoids deadlocks by allocating resources to processes carefully.
- The given code provides a good implementation of the algorithm in C++, and it allows the user to interact with the system and check the current state, as well as make additional requests
- In conclusion, the implementation of the Banker's Algorithm for our mini project has proven to be a valuable learning experience. We have successfully demonstrated the ability to prevent deadlock in a system with multiple processes and limited resources. By utilizing the Banker's Algorithm, we have ensured that all processes are able to access the necessary resources without causing deadlock or starvation. Through this project, we have gained a deeper understanding of the complexities involved in managing system resources and the importance of effective resource allocation. Overall, the implementation of the Banker's Algorithm has provided us with practical skills and knowledge that will be valuable in future projects and real-world scenarios..