OS LAB PROJECT

NAME:

FARM EQUIPMENT MANAGEMENT

OBJECTIVE:

1. To maintain an application that handles the equipment needs of farmers and maintains data of total number of resources, allocated resources, customers using banker's algorithm.
2. To provide a sequence which satisfies farmers needs in the best way possible.
3. To provide feasible solutions so that we can satisfy everyone's needs by deadlock removal.

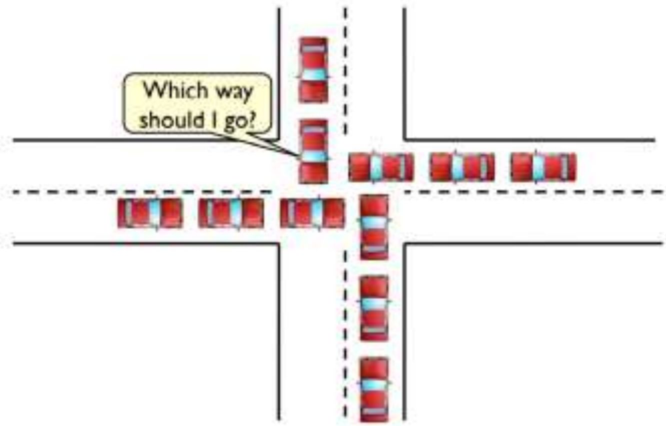
Banker’s algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

Deadlock: definition

There exists a cycle of processes such that each process cannot proceed until the next process takes some specific action. Result: all processes in the cycle are stuck!

Deadlock in the real world



Necessary Conditions for Deadlock:

Mutual exclusion

- Processes claim exclusive control of the resources they require

Hold-and-wait condition

- Processes hold resources already allocated to them while waiting for additional resources

No pre-emption condition

- Resources cannot be removed from the processes holding them until used to completion

Circular wait condition

- A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain

The banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

Following **Data structures** are used to implement the Banker’s Algorithm:
Let **‘n’** be the number of processes in the system and **‘m’** be the number of resources types.

Available :

- It is a 1-d array of size **‘m’** indicating the number of available resources of each type.

- Available[j] = k means there are **‘k’** instances of resource type $\mathbf{R_j}$

Max :

- It is a 2-d array of size ‘**n*m’** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P_i}$ may request at most **‘k’** instances of resource type $\mathbf{R_j}$.

Allocation :

- It is a 2-d array of size **‘n*m’** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **‘k’** instances of resource type $\mathbf{R_j}$

Need :

It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.

Need [i,   j] = k means
process $\mathbf{P_i}$ currently need **'k'** instances of resource type $\mathbf{R_j}$

Need [i,   j] = Max [i, j] – Allocation [i, j]

$Allocation_i$ specifies the resources currently allocated to process P_i and $Need_i$ specifies the additional resources that process P_i may still request to complete its task.
Banker's algorithm consists of Safety algorithm and Resource request algorithm

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.

Initialize: Work = Available

Finish[i] = false; for i=1, 2, 3, 4….n

2) Find an i such that both

a) Finish[i] = false

b) $Need_i$ <= Work

if no such i exists goto step (4)

3) Work = Work + Allocation[i]

Finish[i] = true

goto step (2)

4) if Finish [i] = true for all i

then the system is in a safe state

Resource-Request Algorithm

Let $Request_i$ be the request array for process P_i. $Request_i [j] = k$ means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If $Request_i <= Need_i$
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.
2) If $Request_i <= Available$
Goto step (3); otherwise, P_i must wait, since the resources are not available.
3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:
Available = Available – Requesti
$Allocation_i = Allocation_i + Request_i$
$Need_i = Need_i – Request_i$

PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;
class Farmer
{
public:
   friend class manager;
   vector<int> allocation ;
   vector<int>request;
   Farmer()
   = default;
   Farmer(vector<int> &allocations ,
vector<int>& requests)
   {
      allocation.clear();
      for(auto it : allocations)
         allocation.push_back(it);
      request.clear();
      for(auto it : requests)
         request.push_back(it);
```

```
    }

};
class manager{
public:
    int no_of_farmers;
    int number_of_resources;
    vector<vector<int>> allocated_resources;
    vector<vector<int>> request_matrix;
    vector<vector<int>> needed_resources;
    vector<int> available_resources;
    vector<int> finished_allocation;
    manager()
    {
        no_of_farmers = 0;
        number_of_resources = 0;
    }
    manager(int resource_no , vector<int>
&values)
    {
```

```
        no_of_farmers = 0;
        number_of_resources = resource_no;
        for(auto it : values)
        {
            available_resources.push_back(it);
        }
    }
    void print()
    {

        cout << "\nFarmer\t Allocation\t
Need\t\tRequest\t\t Available\t";
     for (int i = 0; i < no_of_farmers; i++)
{
            cout << "\nF" << i + 1 << "\t ";
            for (int j = 0; j < number_of_resources;
j++)
            {
             cout << allocated_resources[i][j] << " ";
            }
            cout << "\t\t";
```

```
            for (int j = 0; j < number_of_resources;
j++)
            {
                cout << needed_resources[i][j] << " ";
            }
            cout << "\t\t";
            for (int j = 0; j < number_of_resources;
j++)
            {
                cout << request_matrix[i][j] << " ";
            }
            cout << "\t\t ";
            if (i == 0)
            {
                for (int j = 0; j < number_of_resources;
j++)
                    cout << available_resources[j] << " ";
            }
        }
        cout<<"\n";
    }
```

```
bool deadlock_check()
{
    finished_allocation.clear();
    finished_allocation.resize(no_of_farmers);
    finished_allocation.assign(no_of_farmers ,
0);
    int i , j , flag = 1;
    int n = no_of_farmers , r =
number_of_resources;
    vector<vector<int>> need_resource_temp
= needed_resources;
    vector<int> avail_resource =
available_resources;
    vector<vector<int>>
allocated_resource_temp =
allocated_resources;
    while (flag)
    {
        flag = 0;
        for (i = 0; i < n; i++)
        {
```

```
            int c = 0;
            for (j = 0; j < r; j++)
            {
                if ((finished_allocation[i] == 0) &&
(need_resource_temp[i][j] <= avail_resource[j]))
                {
                    c++;
                    if (c == r)
                    {
                        for (int k = 0; k < r; k++)
                        {
                            avail_resource[k] +=
allocated_resource_temp[i][j];
                            finished_allocation[i] = 1;
                            flag = 1;
                        }
                        if (finished_allocation[i] == 1)
                        {
                            i = n;
                        }
                    }
```

```
                }
            }
        }
    }
    j = 0;
    flag = 0;
    for (i = 0; i < n; i++)
    {
        if (finished_allocation[i] == 0)
        {
            j++;
            flag = 1;
        }
    }
    return flag;
}
void add_farmer(Farmer &f1)
{

    no_of_farmers++;
```

```
allocated_resources.push_back(f1.allocation);
       request_matrix.push_back(f1.request);
       vector<int> need;
       need.reserve(number_of_resources);
need.reserve(number_of_resources);
for(int i = 0 ; i < number_of_resources ; i++)
       {
          need.push_back(f1.request[i] -
f1.allocation[i]);
       }

       needed_resources.push_back(need);
       need.clear();
   }

   void safe_sequence() const
   {
       int n = no_of_farmers;
       int r = number_of_resources;
       vector<bool> finish(n);
```

```
        int safeSeq[n];
        int work[r];
        vector<int> avail = available_resources;
        for (int i = 0; i < r ; i++)
            work[i] = avail[i];
        int count = 0;
        vector<vector<int>> need =
needed_resources;
        vector<vector<int>> allot =
allocated_resources;
        while (count < n)
        {
            bool found = false;
            for (int p = 0; p < n; p++)
            {
                if (finish[p] == 0)
                {
                    int j;
                    for (j = 0; j < r; j++)
                        if (need[p][j] > work[j])
                            break;
```

```
                if (j == r)
                {
                    for (int k = 0 ; k < r ; k++)
                        work[k] += allot[p][k];
                    safeSeq[count++] = p;
                    finish[p] = true;
                    found = true;
                }
            }
        }

        if (!found)
        {
            cout << "System is not in safe state";
        }
    }
    for (int i = 0; i < n-1 ; i++)
        cout << "Farmer"<<safeSeq[i]+1 <<" -> ";
    cout<<"Farmer"<<safeSeq[n-1] + 1<<"\n";
}
void remove_deadlock()
```

```
  {
      int ans = (1<<no_of_farmers);
      ans--;
      int cnt = no_of_farmers;
      vector<vector<int>> temp =
allocated_resources;
      vector<int> avail_rs = available_resources;
      vector<vector<int>> need =
needed_resources;
      for(int i = 0 ; i <(1<<no_of_farmers) ; i++)
      {
          allocated_resources = temp;
          available_resources = avail_rs;
          needed_resources = need;
          for(int j = 0 ; j < no_of_farmers ; j++)
          {
              if(i & (1<<j))
              {
                  for(int k = 0 ; k <
number_of_resources ; k++)
                  {
```

```
                    available_resources[k] +=
allocated_resources[j][k];
                    allocated_resources[j][k] = 0;
                    needed_resources[j][k] =
request_matrix[j][k];
                }
            }
        }
        if(!deadlock_check())
        {
            int mini = 0;
            for(int j = 0 ; j < no_of_farmers ; j++)
            {
                if(i & (1<<j))
                {
                    mini++;
                }
            }
            if(mini < cnt){
                cnt = mini;
                ans = i;
```

```
            }
         }
      }
      cout<<"Deallocate all Resources from 
following farmers :\n";
      for(int j = 0 ; j < no_of_farmers ; j++)
      {
         if(ans & (1<<j))
         {
            cout<<"Farmer"<<j+1<<" ";
         }
      }
      cout<<endl;
      available_resources = avail_rs;
      allocated_resources = temp;
      needed_resources = need;
      for(int j = 0 ; j < no_of_farmers ; j++)
      {
         if(ans & (1<<j))
         {
```

```
            for(int k = 0 ; k < number_of_resources
; k++)
            {
               available_resources[k] +=
allocated_resources[j][k];
               allocated_resources[j][k] = 0;
               needed_resources[j][k] =
request_matrix[j][k];
            }
         }
      }
      cout<<"After De-allocation , data is :\n";
      print();
      safe_sequence();
      available_resources = avail_rs;
      allocated_resources = temp;
      needed_resources = need;
   }

};
int main()
```

```
{
    cout<<"\nEnter the number of resources: ";
    int number_of_resources;
    cin>>number_of_resources;
    cout<<"\nEnter the available resources:\n";
    vector<int>
avail_resources(number_of_resources);
    for(int i=0; i < number_of_resources; i+=1){
        cin>>avail_resources[i];
    }
    manager man(number_of_resources ,
avail_resources);
    int x = 1;
    cout<<"Press 1 for adding farmer\nPress 2 for
checking safe state and getting safe
sequence\nPress 3 to print available
date\nPress 4 to exit.\n ";
    cin>>x;
    vector<Farmer> farmers;
    vector<int> allocated(number_of_resources) ,
request(number_of_resources);
```

```
while( x< 4)
{
    if(x == 1)
    {
        cout<<"Allocation Matrix of Farmer \n";
        for(int i = 0 ; i < number_of_resources ;
i++)
        {
            cin>>allocated[i];
        }
        cout<<"Request Matrix of Farmer \n";
        for(int i = 0 ; i < number_of_resources ;
i++)
        {
            cin>>request[i];
        }
        Farmer f(allocated , request);
        farmers.push_back(f);
        man.add_farmer(f);
        cout<<"Press 1 for adding farmer\nPress
2 for checking safe state and getting safe
```

```
sequence\nPress 3 to print available
date\nPress 4 to exit. \n";
          cin>>x;

      }
      else if(x == 2)
      {
          if(!man.deadlock_check()){
              cout<<"No deadlock\nSafe Sequence is
:\n";
              man.safe_sequence();

          }
          else
          {
              cout<<"Deadlock Occurs :\n";
              man.remove_deadlock();
          }
          cout<<"Press 1 for adding farmer\nPress
2 for checking safe state and getting safe
```

```
sequence\nPress 3 to print available
date\nPress 4 to exit.\n ";
            cin>>x;
        }
        else if(x == 3)
        {
            man.print();
            cout<<"Press 1 for adding farmer\nPress
2 for checking safe state and getting safe
sequence\nPress 3 to print available
date\nPress 4 to exit.\n ";
            cin>>x;
        }
        else
        {
            break;
        }
    }

    return 0;
}
```

Sample Input:

3
3 3 2
1
0 1 0
7 5 3
1
2 0 0
3 2 2
1
3 0 2
9 0 2
1
2 1 1
2 2 2
1
0 0 2
4 3 3
3
2
1
0 0 0
10 5 7
3
2
4

Output:

Screenshots:

```
C:\Jasleen\cmake-build-debug\Jasleen.exe

Enter the number of resources:3

Enter the available resources:
3 3 2
Press 1 for adding farmer
Press 2 for checking safe state and getting safe sequence
Press 3 to print available date
Press 4 to exit.
1
 Allocation Matrix of Farmer
0 1 0
Request Matrix of Farmer
7 5 3
Press 1 for adding farmer
Press 2 for checking safe state and getting safe sequence
Press 3 to print available date
Press 4 to exit.
1
Allocation Matrix of Farmer
2 0 0
Request Matrix of Farmer
3 2 2
Press 1 for adding farmer
Press 2 for checking safe state and getting safe sequence
Press 3 to print available date
Press 4 to exit.
1
Allocation Matrix of Farmer
3 0 2
Request Matrix of Farmer
9 0 2
Press 1 for adding farmer
Press 2 for checking safe state and getting safe sequence
Press 3 to print available date
Press 4 to exit.
1
Allocation Matrix of Farmer
2 1 1
Request Matrix of Farmer
2 2 2
Press 1 for adding farmer
Press 2 for checking safe state and getting safe sequence
Press 3 to print available date
Press 4 to exit.
1
Allocation Matrix of Farmer
```

```
0 0 2
Request Matrix of Farmer
4 3 3
Press 1 for adding farmer
Press 2 for checking safe state and getting safe sequence
Press 3 to print available date
Press 4 to exit.
3

Farmer   Allocation      Need          Request         Available
F1       0 1 0          7 4 3          7 5 3           3 3 2
F2       2 0 0          1 2 2          3 2 2
F3       3 0 2          6 0 0          9 0 2
F4       2 1 1          0 1 1          2 2 2
F5       0 0 2          4 3 1          4 3 3
Press 1 for adding farmer
Press 2 for checking safe state and getting safe sequence
Press 3 to print available date
Press 4 to exit.
2
 No deadlock
Safe Sequence is :
Farmer2 -> Farmer4 -> Farmer5 -> Farmer1 -> Farmer3
Press 1 for adding farmer
Press 2 for checking safe state and getting safe sequence
Press 3 to print available date
Press 4 to exit.
1
 Allocation Matrix of Farmer
0 0 0
Request Matrix of Farmer
10 5 7
Press 1 for adding farmer
Press 2 for checking safe state and getting safe sequence
Press 3 to print available date
Press 4 to exit.
3

Farmer   Allocation      Need          Request         Available
F1       0 1 0          7 4 3          7 5 3           3 3 2
F2       2 0 0          1 2 2          3 2 2
F3       3 0 2          6 0 0          9 0 2
F4       2 1 1          0 1 1          2 2 2
F5       0 0 2          4 3 1          4 3 3
F6       0 0 0          10 5 7         10 5 7
Press 1 for adding farmer
Press 2 for checking safe state and getting safe sequence
Press 3 to print available date
```

```
Press 4 to exit.
2
 Deadlock Occurs :
Deallocate all Resources from following farmers :
Farmer2
After De-allocation , data is :

Farmer   Allocation      Need           Request          Available
F1       0 1 0           7 4 3          7 5 3            5 3 2
F2       0 0 0           3 2 2          3 2 2
F3       3 0 2           6 0 0          9 0 2
F4       2 1 1           0 1 1          2 2 2
F5       0 0 2           4 3 1          4 3 3
F6       0 0 0           10 5 7         10 5 7
Farmer2 -> Farmer4 -> Farmer5 -> Farmer1 -> Farmer3 -> Farmer6
Press 1 for adding farmer
Press 2 for checking safe state and getting safe sequence
Press 3 to print available date
Press 4 to exit.
4

Process finished with exit code 0
```