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.

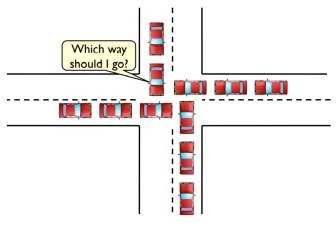
Banker’s algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from 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 **Rj 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 **Pi** may request at most **‘k’** instances of resource type **Rj.**

## 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 **Pi** is currently allocated **‘k’** instances of resource type **Rj**

## 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 **Pi** currently need **‘k’** instances of resource type **Rj**

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

Allocationi specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi 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

1. Find an i such that both
   1. Finish[i] = false
   2. Needi <= Work

if no such i exists goto step (4)

1. Work = Work + Allocation[i]

Finish[i] = true goto step (2)

1. if Finish [i] = true for all i

then the system is in a safe state

## Resource-Request Algorithm

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

1. If Requesti <= Needi

Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

1. If Requesti <= Available

Goto step (3); otherwise, Pi must wait, since the resources are not available.

1. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available = Available – Requesti Allocationi = Allocationi + Requesti Needi = Needi– Requesti

**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>> max\_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 << "**\n**Farmer**\t** Allocation**\t** Need**\t\t**Max**\t\t** Available**\t**";

**for** (int i = 0; i < no\_of\_farmers; i++){

cout << "**\n**F" << 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 << max\_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);

max\_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);

vector<int> safeSequence(n);

vector<int> work(r);

vector<int> temp\_avail = available\_resources;

**for** (int i = 0; i < r ; i++)

work[i] = temp\_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];

safeSequence[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" << safeSequence[i] + 1 << " -> ";

cout << "Farmer" << safeSequence[n - 1] + 1 << "**\n**";

}

void remove\_deadlock()

{

vector<int> max\_possible = available\_resources;

**for**(int j = 0 ; j < no\_of\_farmers ; j++)

{

**for**(int k = 0 ; k < number\_of\_resources ; k++)

{

max\_possible[k] += allocated\_resources[j][k];

}

}

**for**(int j = 0 ; j < no\_of\_farmers ; j++)

{

**for**(int k = 0 ; k < number\_of\_resources ; k++)

{

**if**(max\_matrix[j][k] > max\_possible[k])

{

cout<<"Deadlock cannot be Removed as one or more farmers are requesting resources greater than the overall available resources.**\n**";

**return**;

}

}

}

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] = max\_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] = max\_matrix[j][k];

}

}

}

cout<<"After De-allocation of resources from the farmers mentioned above , data is :**\n**";

print();

safe\_sequence();

available\_resources = avail\_rs;

allocated\_resources = temp;

needed\_resources = need;

}

};

int main()

{

cout<<"**\n**Enter the number of resources: ";

int noOfResources;

cin >> noOfResources;

cout<<"**\n**Enter the available resources:**\n**";

vector<int> availResources(noOfResources);

**for**(int i=0; i < noOfResources; i+=1){

cin >> availResources[i];

}

manager man(noOfResources , availResources);

int x = 1;

cout<<"Press 1 for adding farmer**\n**Press 2 for checking safe state and getting safe sequence**\n**Press 3 to print available date**\n**Press 4 to exit.**\n** ";

cin>>x;

vector<Farmer> farmers;

vector<int> currently\_allocated(noOfResources);

vector<int> maximum\_required(noOfResources);

**while**( x< 4)

{

**if**(x == 1)

{

cout<<"Enter resources currently allocated to farmer **\n**";

**for**(int i = 0 ; i < noOfResources ; i++)

{

cin >> currently\_allocated[i];

}

cout<<"Enter maximum resources demanded by farmer **\n**";

**for**(int i = 0 ; i < noOfResources ; i++)

{

cin >> maximum\_required[i];

}

Farmer f(currently\_allocated , maximum\_required);

farmers.push\_back(f);

man.add\_farmer(f);

cout<<"Press 1 for adding farmer**\n**Press 2 for checking safe state and getting safe sequence**\n**Press 3 to print available date**\n**Press 4 to exit. **\n**";

cin>>x;

}

**else** **if**(x == 2)

{

**if**(!man.deadlock\_check()){

cout<<"No deadlock**\n**Safe Sequence is :**\n**";

man.safe\_sequence();

}

**else**

{

cout<<"Deadlock Occurs :**\n**";

int y;

cout<<"Press 1 to remove deadlock and print safe sequence, any other key to continue**\n**";

cin>>y;

**if**(y == 1)

{

man.remove\_deadlock();

}

}

cout<<"Press 1 for adding farmer**\n**Press 2 for checking safe state and getting safe sequence**\n**Press 3 to print available date**\n**Press 4 to exit.**\n** ";

cin>>x;

}

**else** **if**(x == 3)

{

man.print();

cout<<"Press 1 for adding farmer**\n**Press 2 for checking safe state and getting safe sequence**\n**Press 3 to print available date**\n**Press 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**

**1**

**1**

**0 0 0**

**20 20 20**

**2**

**ScreenShot:**

