BANKER'S ALGORITHM

Experiment No: 5 Date: / /23

Aim: a) To implement banker's algorithm

Theory:

Bankers' algorithm is the deadlock avoidance algorithm used to a resource allocation system with multiple instances of each resource type. The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- *Available:* Vector of length m. If available [j] = k, there are k instances of resource type Rj available.
- *Max:* $n \times m$ matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj.
- *Allocation:* $n \times m$ matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj.
- *Need:* $n \times m$ matrix. If $Need$[i,j] = k, then Pi may need k more instances of Rj to complete its task.

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

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize: *Work* := *Available*

Finish [i] = *false* for i - 1,3, …, n.

2. Find and i such that both:

(a) *Finish* [i] = *false*

(b) $Needi \leq Work$

If no such i exists, go to step 4.

3. *Work* := *Work* + *Allocationi*
Finish[*i*] := *true*
go to step 2.

4. If *Finish* [*i*] = true for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process *Pi*

Requesti = request vector for process *Pi*. If *Requesti*[*j*] = *k* then process *Pi* wants *k* instances of resource type *Rj*.

1. If *Requesti* ≤ *Needi* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If *Requesti* ≤ *Available*, go to step 3. Otherwise *Pi* must wait, since resources are not available.

3. Pretend to allocate requested resources to *Pi* by modifying the state as follows:

Available := *Available* = *Requesti;*

Allocationi := *Allocationi* + *Requesti*;

Needi := *Needi* – *Requesti;;*

• *If safe* ⇒ *the resources are allocated to Pi.*

• *If unsafe* ⇒ *P*i *must wait, and the old resource-allocation state is restored*

Example :

```
------------------------ Initial State ------------------------
Maximum Demand Matrix (max_matrix):
A   B   C
7   5   3   (P1)
3   2   2   (P2)
9   0   2   (P3)
2   2   2   (P4)
4   3   3   (P5)
Allocation Matrix (alloc_matrix):
0   1   0   (P1)
2   0   0   (P2)
3   0   2   (P3)
2   1   1   (P4)
0   0   2   (P5)
Available Vector (available_vector):
A   B   C
3   3   2
```

```
------------------------- Request Scenario -------------------------
Request Matrix for P2 (request_matrix):
1   0   0
Checking conditions:
Need Matrix (need_matrix):
A   B   C
1   2   2
Available Resources (available_vector):
A   B   C
3   3   2
Request is valid.
Granting the request:
Allocated Matrix (alloc_matrix):
A   B   C
2   0   0   (P2)
Updated Need Matrix (need_matrix):
A   B   C
0   2   2
Updated Available Resources (available_vector):
A   B   C
2   3   2
------------------------- Final State -------------------------
Maximum Demand Matrix (max_matrix):
A   B   C
7   5   3   (P1)
3   2   2   (P2)
9   0   2   (P3)
2   2   2   (P4)
4   3   3   (P5)
Allocation Matrix (alloc_matrix):
0   1   0   (P1)
2   0   0   (P2)
3   0   2   (P3)
2   1   1   (P4)
0   0   2   (P5)
Available Vector (available_vector):
A   B   C
2   3   2
```

Program :

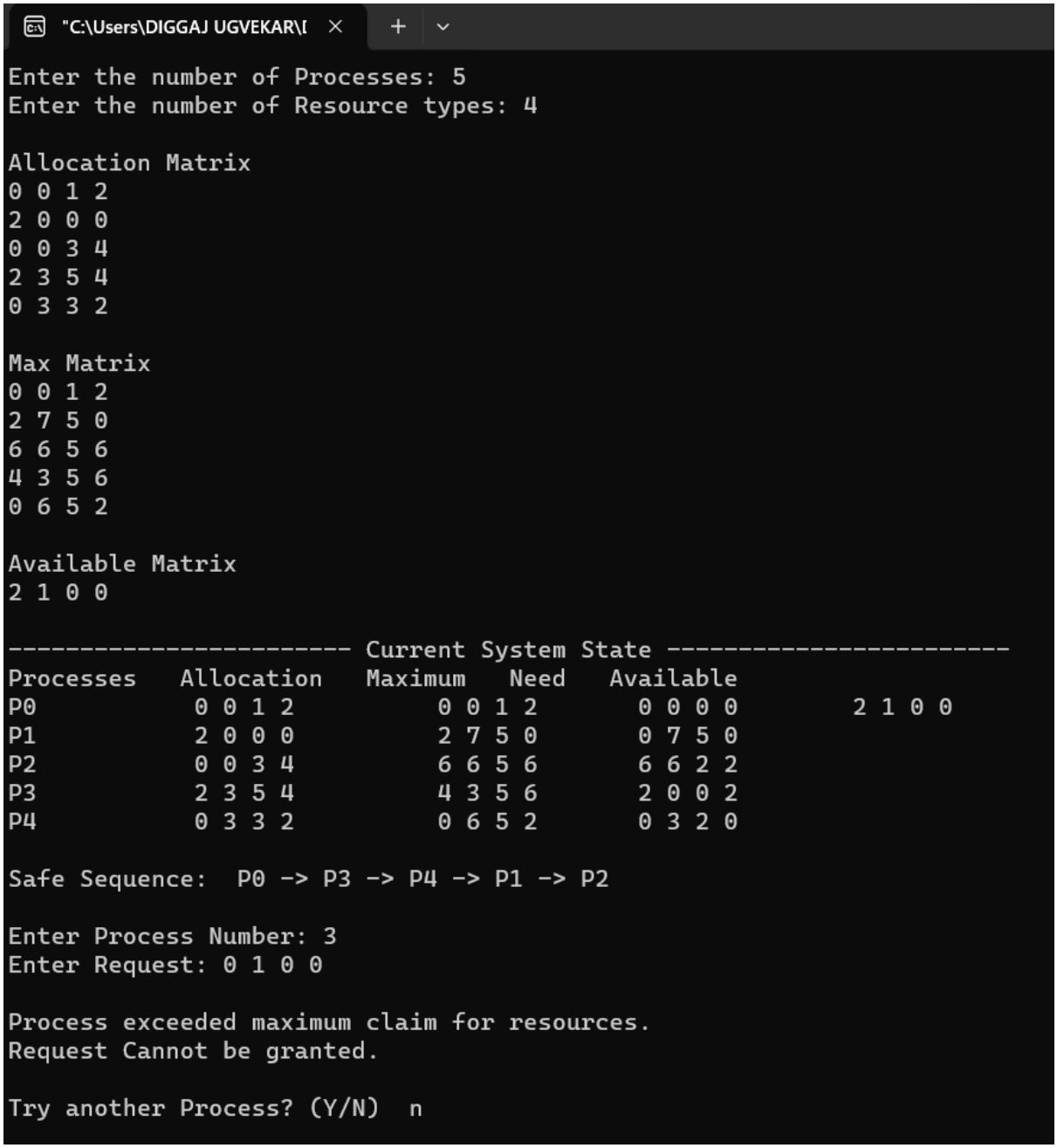
```
#include<iostream>
#include<iomanip>
#include<stdlib.h>

using namespace std;

int m, n, flag;

int allocation[10][10], maximum[10][10],
need[10][10];
int available[10];

void safety()
{
int safe[10], work[10];
bool finish[10];

for(int i = 0; i < n; i++)
{
safe[i] = -1;
finish[i] = false;
}

for(int i = 0; i < m; i++)
work[i] = available[i];

int k = 0, loop_flag;

do
{
loop_flag = 0;

for(int i = 0; i < n; i++)
{
flag = 0;

for(int j = 0; j < m; j++)
{
if (finish[i] == false && need[i][j] <=
work[j])
continue;
else
flag = 1;
}

if (flag == 0)
{
finish[i] = true;

for(int j = 0; j < m; j++)
work[j] = work[j] + allocation[i][j];

safe[k] = i;
k++;
loop_flag = 1;
}
}

flag = 0;

for(int i = 0; i < n; i++)
{
if (finish[i] == false)
flag = 1;
}

if (flag == 0)
break;

if (loop_flag == 0)
```

```
break;
} while(true);

if (flag == 0)
{
cout << endl << "Safe Sequence:  ";
for (int j = 0; j < n; j++)
{
if (j == n - 1)
{
cout << "P" << safe[j];
}
else
{
cout << "P" << safe[j] << " -> ";
}
}
cout << endl;
}
else
cout << endl << "Safe sequence doesn't
exist" << endl;
}

void display()
{
cout << endl << "------------------------
Current System State ---------------------
---" << endl;
cout << "Processes   Allocation   Maximum
Need   Available" << endl;

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

if (i == 0)
{
cout << "       ";
for (int j = 0; j < m; j++)
{
cout << available[j] << " ";
}
}

cout << endl;
}
}

int main()
{
cout << "Enter the number of Processes: ";
cin >> n;

cout << "Enter the number of Resource
types: ";
cin >> m;

cout << "\nAllocation Matrix" << endl;
for (int i = 0; i < n; i++)
{
for (int j = 0; j < m; j++)
cin >> allocation[i][j];
}

cout << endl << "Max Matrix" << endl;
```

```
for (int i = 0; i < n; i++)
{
for (int j = 0; j < m; j++)
cin >> maximum[i][j];
}

cout << endl << "Available Matrix" << endl;
for (int i = 0; i < m; i++)
cin >> available[i];

for (int i = 0; i < n; i++)
{
for (int j = 0; j < m; j++)
need[i][j] = maximum[i][j] - allocation[i][j];
}

display();
safety();

char ans = 'y';

do
{
int request[10], p;
cout << endl << "Enter Process Number: ";
cin >> p;

cout << "Enter Request: ";
for (int i = 0; i < m; i++)
cin >> request[i];

for (int i = 0; i < m; i++)
{
if (need[p][i] < request[i])
{
cout << endl << "Process exceeded maximum claim for resources.\nRequest Cannot be granted." << endl;
goto end;
}
if (available[i] < request[i])
{
cout << endl << "Process must wait. Resources not available." << endl;
goto end;
}
}

for (int i = 0; i < m; i++)
{
available[i] -= request[i];
allocation[p][i] += request[i];
need[p][i] -= request[i];
}

cout << endl << endl;
display();
safety();

if (flag == 1)
{
cout << "Request cannot be granted." << endl;
for (int i = 0; i < m; i++)
{
available[i] += request[i];
allocation[p][i] -= request[i];
need[p][i] += request[i];
}
cout << endl << "States Reverted:" << endl;
display();
}
```

```
else

{

cout << endl << "Safe Sequence Exists, and
the request can be granted immediately to
the process." << endl;

cout << "Snapshot after request:" << endl;

display();

}
```

```
end:

cout << endl << "Try another Process?
(Y/N)  ";

cin >> ans;

} while (ans == 'y' || ans == 'Y');


return 0;

}
```



Conclusion: The Bankers algorithm was studied and implemented successfully.