Operating System Assignment
On
Banker's Algorithm

Student Name: GURRAM SANTHOSH REDDY

Student ID: 11705347

Roll no:50

Email Address: santhoshreddy4148@gmail.com

GitHub Link: https: https://github.com/Gurram99/santhosh123

Code: 19

Submitted To: Ms.shaina Gupta

Description:

This algorithm is commonly known as the banker's algorithm. 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 the needs of all its customers. 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.

Algorithm:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int Max[10][10], need[10][10], alloc[10][10], avail[10], completed[10], safeSequence[10];
   int p, r, i, j, process, count;
   count = 0;

   printf("Enter the no of processes : ");
   scanf("%d", &p);

   for(i = 0; i< p; i++)
      completed[i] = 0;

   printf("\n\nEnter the no of resources : ");
   scanf("%d", &r);

   printf("\n\nEnter the Max Matrix for each process : ");
   for(i = 0; i < p; i++)
   {
```

```
    printf("\nFor process %d : ", i + 1);
    for(j = 0; j < r; j++)
        scanf("%d", &Max[i][j]);
}

printf("\n\nEnter the allocation for each process : ");
for(i = 0; i < p; i++)
{
    printf("\nFor process %d : ",i + 1);
    for(j = 0; j < r; j++)
        scanf("%d", &alloc[i][j]);
}

printf("\n\nEnter the Available Resources : ");
for(i = 0; i < r; i++)
    scanf("%d", &avail[i]);

for(i = 0; i < p; i++)

    for(j = 0; j < r; j++)
        need[i][j] = Max[i][j] - alloc[i][j];

    do
    {
        printf("\n Max matrix:\tAllocation matrix:\n");

        for(i = 0; i < p; i++)
        {
            for( j = 0; j < r; j++)
                printf("%d ", Max[i][j]);
```

```
    printf("\t\t");
    for( j = 0; j < r; j++)
        printf("%d ", alloc[i][j]);
    printf("\n");
}

process = -1;

for(i = 0; i < p; i++)
{
    if(completed[i] == 0)//if not completed
    {
        process = i ;
        for(j = 0; j < r; j++)
        {
            if(avail[j] < need[i][j])
            {
                process = -1;
                break;
            }
        }
    }
    if(process != -1)
        break;
}

if(process != -1)
{
    printf("\nProcess %d runs to completion!", process + 1);
    safeSequence[count] = process + 1;
```

```
            count++;
            for(j = 0; j < r; j++)
            {
                avail[j] += alloc[process][j];
                alloc[process][j] = 0;
                Max[process][j] = 0;
                completed[process] = 1;
            }
        }
    }
    while(count != p && process != -1);

    if(count == p)
    {
        printf("\nThe system is in a safe state!!\n");
        printf("Safe Sequence : < ");
        for( i = 0; i < p; i++)
            printf("%d ", safeSequence[i]);
        printf(">\n");
    }
    else
        printf("\nThe system is in an unsafe state!!");

}
```

Algorithm:

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource Types:

Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.

Max: An n × m matrix defines the maximum demand of each process. If Max[i][j] equals k, then process Pi may request at most k instances of resource type Rj.

Allocation: An n × m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals k, then process Pi is currently allocated k instances of resource type Rj.

Need: An n × m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task. Note that Need[i][j] equals Max[i][j] −Allocation[i][j].

Boundary conditions:
Maximum no of process and resources are five.

Test cases:

Test case 1:

Maximum Instances Available			Processes	Allocation			Maximum		
				A	B	C	A	B	C
A	B	C	P0	0	1	0	7	5	3
10	5	7	P1	2	0	0	3	2	2
			P2	3	0	2	9	0	2
			P3	2	1	1	2	2	2
			P4	0	0	2	4	3	3

Test case 2:

Maximum Instances Available			Processes	Allocation			Maximum		
				A	B	C	A	B	C
A	B	C	P0	0	1	0	7	5	3
10	5	7	P1	2	0	0	3	2	2
			P2	3	0	2	9	0	2
			P3	2	1	1	2	2	7
			P4	0	0	2	4	3	3

GitHub Link: https://github.com/Gurram99/santhosh123