Course COM 312:- Operating System lab

Group :-7

Ayush Pandita – 2021a1r174
Rahul Sharma – 2021a1r171
Ravinder Singh Bogal- 2021a1r176
Sourav Salaria-2021a1r169

Submitted to: Mr. Saurabh Sharma
Professor

Department of computer science and Engineering
MIET(Autonomous),Jammu

Project Title:- Simulate The Prediction of Deadlock In Operating System When all Processes announce their resource requirement in advance.

Deadlock Avoidance

Problem Statement 16:-

Solution:- Avoidance Algorithms The deadlock-avoidance algorithm helps you to dynamically assess the resource-allocation state so that there can never be a circular-wait situation . A single instance of a resource type . Use a resource-allocation graph Cycles are necessary which are sufficient for Deadlock Multiples instances of a resource type . Cycles are necessary but never sufficient for Deadlock . Uses the banker's algorithm . It is occur for safe state not for unsafe state .

Features:-

- It contains various resources that meet the requirements of each process.
- Each process should provide information to the operating system for upcoming resource requests, the number of resources, and how long the resources will be held.
- It helps the operating system manage and control process requests for each type of resource in the computer system .
- The algorithm has a Max resource attribute that represents indicates each process can hold the maximum number of resources in a system.

Real world example of bankers Algorithm:-

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system.

Safety Algorithm used:-

Step1- Initialize work = Available

Finish[i]= False , for i = 0,1,2,…n-1

Step2- Check the availability

Need[i]<=work go to step3

Else Finish[i] == False If I does not exist go to step4

Step3-work= work + Allocation(i)

Finish[i] = true then go to step2

Step4-if Finish[i] == true for all process system is safe state

Resource Request Algorithm:-

Step 1- if request <= need, go to step2

Else error

Step2- if request <=available, go to step3

Else wait

Step3- Available = Available – request

Allocation = allocation + request

Need = need – request

Step4- Check new state is safe or not .

Example- Consider The following System

Process	Allocation	Max	Available
	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

Ans or Output or Proof Need Matrix(Max – Allocation)

	A	B	C	D
P0	0	0	0	0
P1	0	7	5	0
P2	1	0	0	2
P3	0	0	2	0
P4	0	6	4	2

Following is the Safe Sequence

P0 P2 P3 P4 P1 .

Algorithm Used for C program of Deadlock avoidance:-

Step1- Start the Program .

Step2- Declare The memory for the process .

Step3- Read the Number of process , resources , allocation matrix & available matrix .

Step4- Calculate the need matrix : need = max – allocation

Step 5- Compare each and every Process using the banker . ’s algorithm .

Step6- If the process is in safe state then it is not a deadlock process Otherwise it is a deadlock process .

Step7- Produce the result of state of process .

Step 8- stop the Program .

```
#include <stdio.h>

int main()

{

 int n, m, i, j, k, y,alloc[20][20],max[20][20],avail[50],ind=0;

 printf("Enter the no of Proceses:");

 scanf("%d",&n);

 printf("Enter the no of Resources:");

 scanf("%d",&m);

 printf("Enter the Allocation Matrix:");

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

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

 scanf("%d",&alloc[i][j]);
}
```

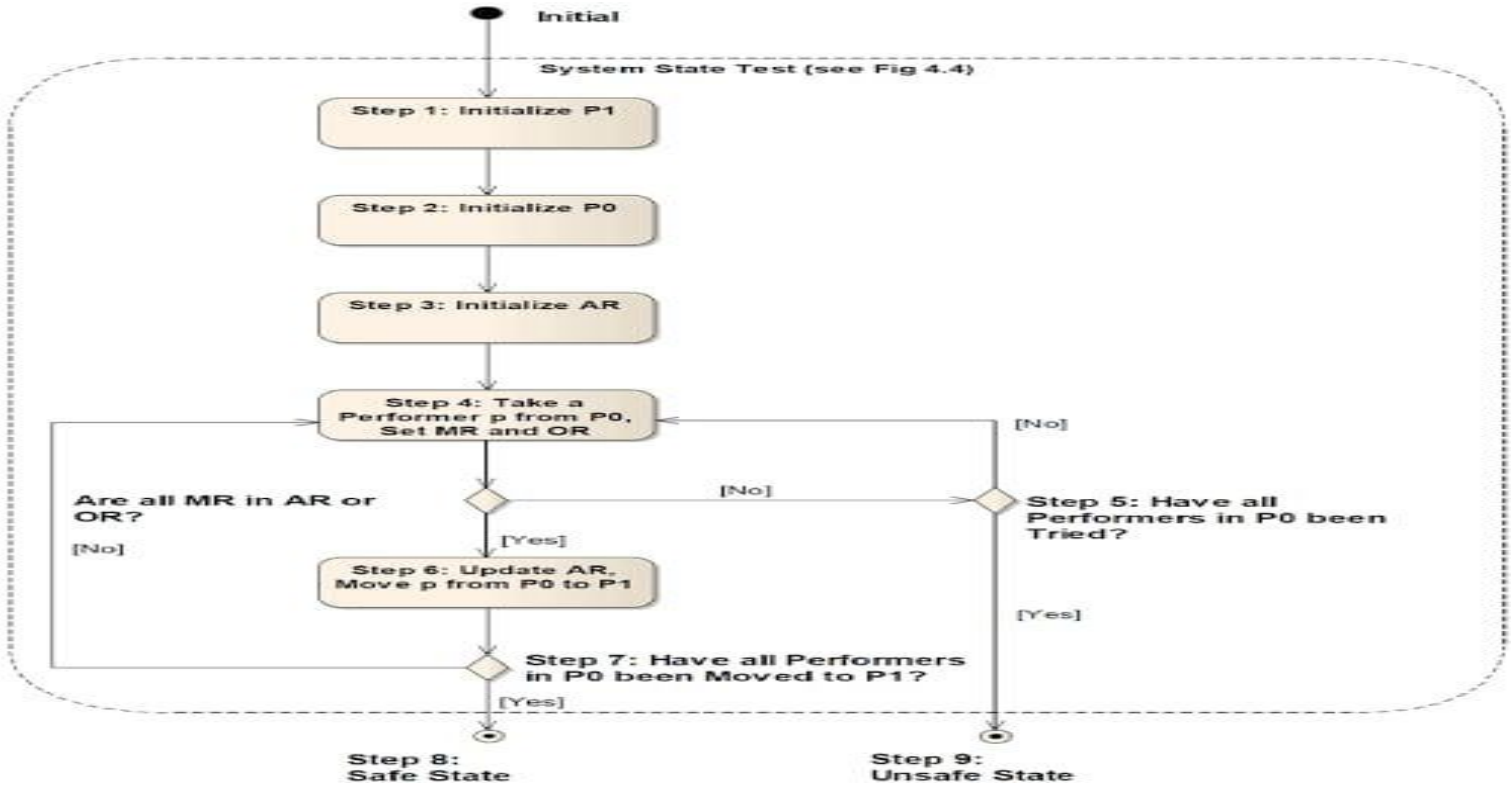
```
 printf("Enter the Max Matrix:");

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

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

 scanf("%d",&max[i][j]);

 }

printf("Enter the Available Matrix");

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

 scanf("%d",&avail[i]);

 int finish[n], safesequence[n],work[m],need[n][m];

//calculating NEED matrix

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

 for (j = 0; j < m; j++)
 need[i][j] = max[i][j] - alloc[i][j];

 }

 printf("NEED matrix is");

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

 {

 printf("\n");

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

 printf(" %d ",need[i][j]);

 }

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

 {

 work[i]=avail[i];
}
```

```
for (i = 0; i < n; i++) {
 finish[i] = 0;
 }
 for (k = 0; k < n; k++) {
 for (i = 0; i < n; i++)
 {
 if (finish[i] == 0)
 {
 int flag = 0;
 for (j = 0; j < m; j++)
 {
 if (need[i][j] > work[j])
{
```

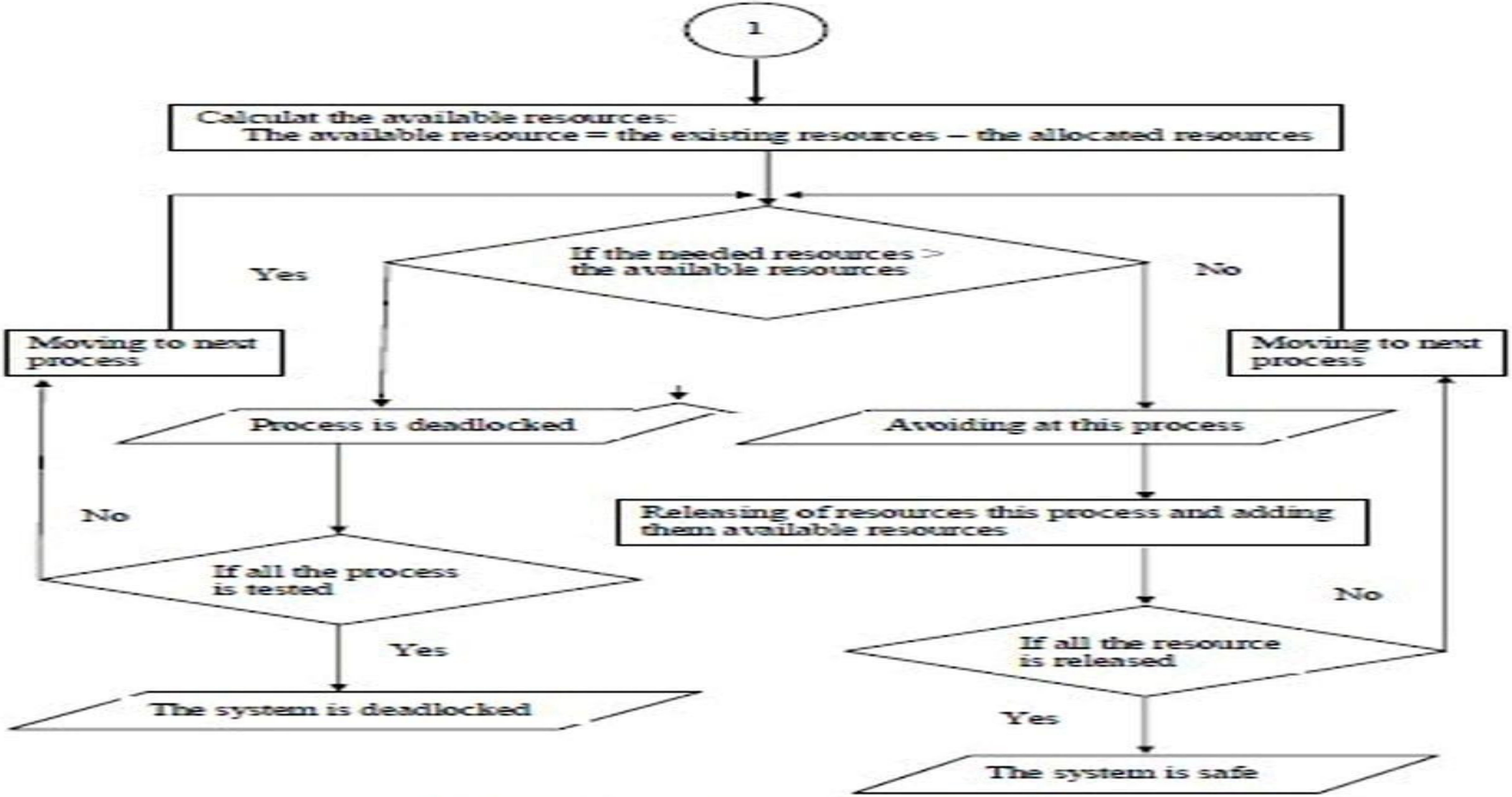
```
flag = 1;

break;

}

}

if (flag == 0) {

safesequence[ind++] = i;

for (y = 0; y < m; y++)

work[y] += alloc[i][y];

finish[i] = 1;

}

}

}
```

```
 }

 printf("\nFollowing is the SAFE Sequence\n");

 for (i = 0; i <= n - 1; i++)

 printf(" P%d ", safesequence[i]);

}
```

Flowchart of bankers algorithm


Initial
System State Test (see Fig 4.4)
Step 1: Initialize P1
Step 2: Initialize P0
Step 3: Initialize AR
Step 4: Take a Performer p from P0, Set MR and OR
[No]
Are all MR in AR or OR?
[No]
Step 5: Have all Performers in P0 been Tried?
[No]
[Yes]
Step 6: Update AR, Move p from P0 to P1
[Yes]
Step 7: Have all Performers in P0 been Moved to P1?
[Yes]
Step 8: Safe State
Step 9: Unsafe State

Flowchart for Safety Algorithm



1

Calculat the available resources:
The available resource = the existing resources – the allocated resources

If the needed resources > the available resources

Yes

No

Moving to nest process

Moving to nest process

Process is deadlocked

Avoiding at this process

Releasing of resources this process and adding them available resources

No

If all the process is tested

Yes

The system is deadlocked

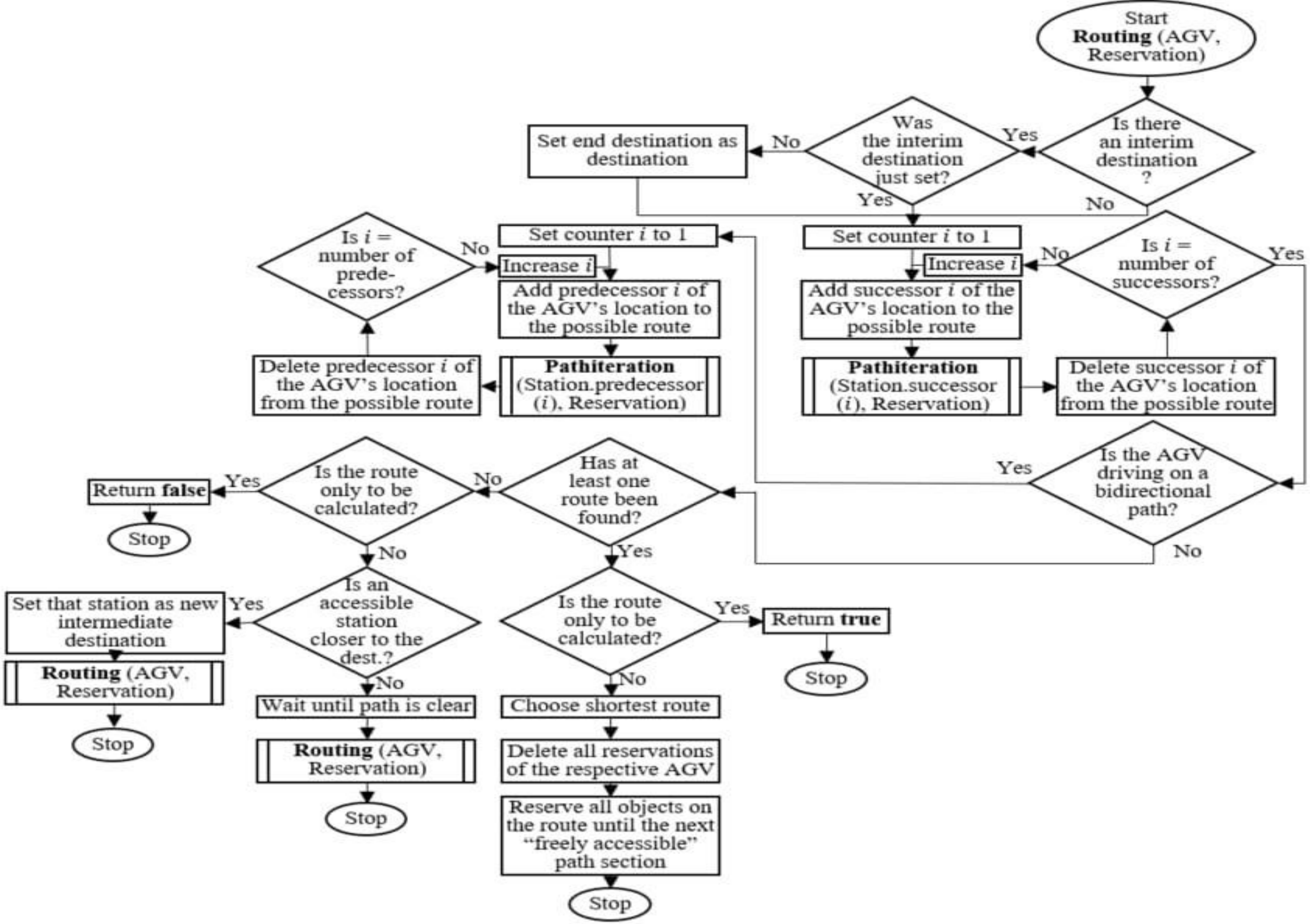
No

If all the resource is released

Yes

The system is safe

Flowchart for deadlock avoidance :-



Start Routing (AGV, Reservation)
Is there an interim destination ?
Yes
No
Was the interim destination just set?
No
Yes
Set end destination as destination
Set counter i to 1
Increase i
Add successor i of the AGV's location to the possible route
Pathiteration (Station.successor (i), Reservation)
Delete successor i of the AGV's location from the possible route
Is i = number of successors?
No
Yes
Is the AGV driving on a bidirectional path?
Yes
No
Set counter i to 1
Increase i
Add predecessor i of the AGV's location to the possible route
Pathiteration (Station.predecessor (i), Reservation)
Delete predecessor i of the AGV's location from the possible route
Is i = number of prede-cessors?
No
Has at least one route been found?
No
Yes
Is the route only to be calculated?
Yes
Return false
Stop
No
Is an accessible station closer to the dest.?
Yes
Set that station as new intermediate destination
Routing (AGV, Reservation)
Stop
No
Wait until path is clear
Routing (AGV, Reservation)
Stop
Is the route only to be calculated?
Yes
Return true
Stop
No
Choose shortest route
Delete all reservations of the respective AGV
Reserve all objects on the route until the next "freely accessible" path section
Stop