SIMULATION BASED ASSIGNMENT ASSESMENT

On

<u>SAFE STATE (DEAD LOCK)</u>

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE AND ENGINEERING

By

NAME: MANISH KUMAR SINGH

Mail ID: manishsinghonline2@gmail.com

GitHub: github.com/manishsinghgithub

Registration number: 11814398

Roll No: A19

Section: K18GA

School of Computer Science and Engineering

Lovely Professional University

Phagwara, Punjab (India)

SOLUTION CODE:QUESTION:-19

DESCRIPTION:

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes is a safe sequence for the current allocation state if, for each Pi , the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i. In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

ALGORITHM:

In order to check whether the system is in safe sate or to make arrangement of processes in such a way that system should remain in safe state. A very popular algorithm is given named as 'Banker's Algorithm'.

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.

1) **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 $\mathbf{R_j}$

2) **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 **P**$_i$ may request at most **'k'** instances of resource type **R**$_j$.

3) 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 **P**$_i$ is currently allocated **'k'** instances of resource type **R**$_j$

4) 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 **P**$_i$ currently need **'k'** instances of resource type **R**$_j$
  for its execution.
- 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
$Allocation_i = Allocation_i + Request_i$
$Need_i = Need_i – Request_i$

PURPOSE OF USE:

Banker's Algorithm is deadlock avoidance algorithm, we use it to make sure our system should remain in safe state while multiple processes are running, requesting for resources and using resources.

CODE SNIPPET:

```
#include <stdio.h>
#include <stdlib.h>
#include<unistd.h>
int main()
{
	// P0, P1, P2, P3, P4 are the Process names here

	int n, m, i, j, k;
	n = 5; // Number of processes
	m = 3; // Number of resources
	int alloc[5][3] = { { 0, 1, 0 },	// P0 // Allocation Matrix
						{ 2, 0, 0 },	// P1
						{ 3, 0, 2 },	// P2
						{ 2, 1, 1 },	// P3
```

```
                                        { 0, 0, 2 } };    // P4

int max[5][3] = { { 7, 5, 3 },       // P0 // MAX Matrix
                  { 3, 2, 2 },        // P1
                  { 9, 0, 2 },        // P2
                  { 2, 2, 2 },        // P3
                  { 4, 3, 3 } };      // P4

int avail[3] = { 3, 3, 2 };          // Available Resources

int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++) {
        f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
                need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
                if (f[i] == 0) {

                        int flag = 0;
                        for (j = 0; j < m; j++) {
                                if (need[i][j] > avail[j]){
                                        flag = 1;
                                        break;
                                }
                        }
```

```
                if (flag == 0) {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }

    printf("Following is the SAFE Sequence:-\n");
    for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d", ans[n - 1]);

    return (0);

}
```

Boundary condition of implemented code:

- It requires the number of processes to be fixed, no addition process can start while it is executing.
- It requires that the number of resources remain fixed, no resource may go down for any reason without the possibility of deadlock.
- It allows all requests to be granted in finite time, but one year is a finite amount of time (not beneficial in long term time).
- All resources must know and state their maximum resource need in advance.

Test cases:

Case 1:

If value in allocation matrix get negative for process 0, process 0 will never get chance to use resources.

The output of program: P1 -> P3 -> P4 -> P2 -> P1

Case 2:

If available matrix get zero value then deadlock condition will occur.

The output will be: P2949205 -> P5439555 -> P32 -> P0 -> P1

Case 3:

If the number of resources greater (2+number of processes) than number processes, the deadlock condition occur.

Program output: P1 -> P5439555 -> P32 -> P0 -> P1

Case 4: If program does not follow any boundary condition which is mentioned above, the solution we get will either wrong or a deadlock condition.