**LOVELY PROFESSIONAL UNIVERSITY**  
**Academic Task-3 (Operating System)**

School of Computer Science and Engineering Faculty of Technology And Sciences

Name of the faculty member :Ms. Isha

Course Code: CSE 316 Course Title: Operating System

Term: 2018-2019

Max. Marks:30 Date of Submission: 15:04:2019

**Student Name : ROSHNI TIRKEY**

**Student ID: 11400072**

**Email Address:lovelyalia75 @gmail.com**

Ques. 11. Reena’s operating system uses an algorithm for deadlock avoidance to manage the allocation of resources say three namely A, B, and C to three processes P0, P1, and P2. Consider the following scenario as reference .user must enter the current state of system as given in this example :

Suppose P0 has 0,0,1 instances , P1 is having 3,2,0 instances and P2 occupies 2,1,1 instances of A,B,C resource respectively.

Also the maximum number of instances required for P0 is 8,4,3 and for p1 is 6,2,0 and finally for P2 there are 3,3,3 instances of resources A,B,C respectively. There are 3 instances of resource A, 2 instances of resource B and 2 instances of resource C available. Write a program to check whether Reena’s operating system is in a safe state or not in the following independent requests for additional resources in the

current state:

1. Request1: P0 requests 0 instances of A and 0 instances of B and 2 instances of C.
2. Request2: P1 requests for 2 instances of A, 0 instances of B and 0 instances of C.

All the request must be given by user as input.

**Description of the program:*Deadlock***is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.  
Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

**Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)**  
***Mutual Exclusion:*** One or more than one resource are non-sharable (Only one process can use at a time)  
***Hold and Wait:***A process is holding at least one resource and waiting for resources.  
***No Preemption:*** A resource cannot be taken from a process unless the process releases the resource.  
***Circular Wait:*** A set of processes are waiting for each other in circular form.

**Methods for handling deadlock**  
There are three ways to handle deadlock  
1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

**Bankers 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.

**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**

for its execution.

* 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 consist of Safety algorithm and Resource request algorithm

**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) Needi <= 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 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.*

*2) If Requesti <= Available  
Goto step (3); otherwise, Pi 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  
Allocationi = Allocationi + Requesti  
Needi = Needi– Requesti*

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Allocation | Need | Availability |
|  | A B C |  |  |
| P0 | 0 0 1 | 8 4 3 | 2 1 0 |
| P1 | 3 2 0 | 6 2 0 |  |
| P2 | 2 1 1 | 3 3 3 |  |

#include<iostream>

#include<stdio.h>

using namespace std;

int main()

{

int n;

int r;

int i,j,k;

int need[10][10],alloc[10][10],max[10][10];

int avail[10],p[10];

printf("\nEnter number of process :");

scanf("%d",&n);

printf("\n Enter resources available : ");

scanf("%d",&r);

printf("\nEnter instances for resources :\n");

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

{ printf("R%d ",i+1);

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

}

printf("\n Enter allocation matrix \n");

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

{

printf("p%d",i+1); p[i]=0;

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

{

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

}

}

printf("\n Enter MAX matrix \n");

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

{

printf("p%d",i+1);

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

{

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

}

}

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

{

printf("\np%d\t",i+1) ;

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

{

need[i][j]=max[i][j]-alloc[i][j];

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

}

}

printf("\n\n");

int flag=0;

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

{

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

{

if(avail[j]>=need[i][j])

flag=1;

else

flag=0;

}

}

if(flag==0)

printf("Unsafe State");

else

printf("Safe State");

}