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

School of Computer Science and Engineering Faculty of Technology and Sciences

Name of the faculty member: Prof. Isha

Course Code: CSE 316 Course Title: Operating System

Term: Spring term (2019-2020)   
Max. Marks:30 Date of Submission: 12th April 2020

**Student Name: Bhagesh Khatri**

**Student ID: 11914224**

**Email Address: khatribhagesh1122@gmail.com**

**GitHub Link:** <https://github.com/khatribhagesh1122>

**1. Description:**

**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.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2015/06/deadlock.png)

**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 Pre-emption:** 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.  
One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.  
Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources which process WILL need are known to us prior to execution of the process. We use Banker’s algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

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

3) **Ignorance:** 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.

**2. Algorithm:**

Steps of Algorithm:

1. Let *Work* and *Finish* be vectors of length m and n respectively. Initialize *Work= Available*. For *i=0, 1, …., n-1*, if *Requesti* = 0, then *Finish[i]* = true; otherwise, *Finish[i]*= false.
2. Find an index i such that both  
   a) *Finish[i] == false*  
   b) *Requesti <= 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]== false* for some i, 0<=i<n, then the system is in a deadlocked state. Moreover, if *Finish[i]==false* the process Pi is deadlocked.

For example,

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Available | | | Processes | Allocation | | | Max | | |
| A | B | C | A | B | C | A | B | C |
| 3 | 3 | 2 | P0 | 0 | 1 | 0 | 7 | 5 | 3 |
|  | | | 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 |

3 3 2 //Available

P1 – 2 0 0

5 3 2

P4 – 0 0 2

5 3 4

P3 – 2 1 1

7 4 5

Deadlock Detected

3 3 2 //Available

P3 – 2 1 1

5 3 2

P1 – 2 0 0

5 3 4

P4 – 0 0 2

7 4 5

Deadlock Detected

**3. Purpose of use:**

**4. Code snippet:**

// solution was done in dev c++ so if you run it in ubuntu just use the stdio header file thankyou and it works for both the questions

#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");

}

**5. Description:**

**6. Test Cases:**

**GitHub Link:** <https://github.com/khatribhagesh1122>