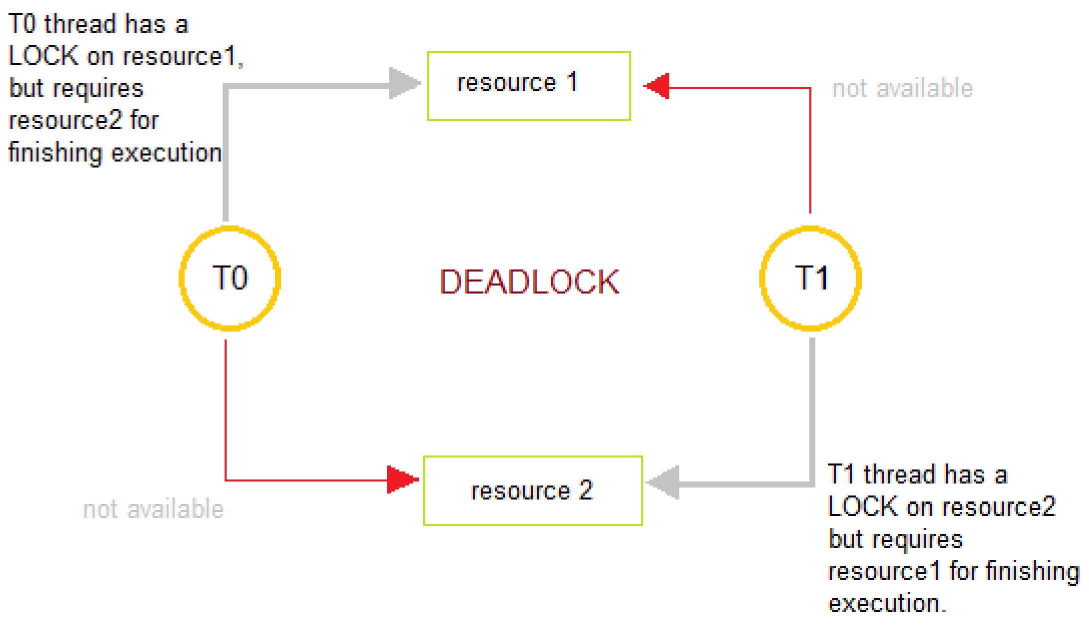
Student Name: Sai Pranup
Student ID : 11804528
Email Address: saipranupyadav@gmail.com
GitHub Link: https://github.com/saipranup/OS-Bankers-problem
Code: qstn no. 19

Description:

Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process. In simple, deadlock is a situation where - The execution of two or more processes is blocked because each process holds some resource and waits for another resource held by some other process.


T0 thread has a LOCK on resource1, but requires resource2 for finishing execution
resource 1
not available
T0
DEADLOCK
T1
not available
resource 2
T1 thread has a LOCK on resource2 but requires resource1 for finishing execution.

Here

- Process T0 holds resource2 and waits for resource1 which is held by process T1.
- Process T1 holds resource1 and waits for resource2 which is held by process T0.
- None of the two processes can complete and release their resource.
- Thus, both the processes keep waiting infinitely.

How to avoid Deadlocks :

Deadlocks can be avoided by avoiding at least one of the four conditions, because all this four conditions are required simultaneously to cause deadlock.

Mutual Exclusion :
Resources shared such as read-only files do not lead to deadlocks but resources, such as printers and tape drives, requires exclusive access by a single process.
Hold and Wait :
In this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others.
No Preemption :
Preemption of process resource allocations can avoid the condition of deadlocks, where ever possible.
Circular Wait :
Circular wait can be avoided if we number all resources, and require that processes request resources only in strictly increasing(or decreasing) order.

Handling Deadlock :
The above points focus on preventing deadlocks. But what to do once a deadlock has occured. Following three strategies can be used to remove deadlock after its occurrence.

Preemption :
We can take a resource from one process and give it to other. This will resolve the deadlock situation, but sometimes it does causes problems.

Rollback :
In situations where deadlock is a real possibility, the system can periodically make a record of the state of each process and when deadlock occurs, roll everything back to the last checkpoint, and restart, but allocating resources differently so that deadlock does not occur.

Kill one or more processes :
This is the simplest way, but it works.

Banker's Algorithm is used majorly in the banking system to avoid deadlock. It helps you to identify whether a loan will be given or not.

This algorithm is used to test for safely simulating the allocation for determining the maximum amount available for all resources. It also checks for all the possible activities before determining whether allocation should be continued or not.
The basic data structures used to implement this algorithm are given below.

Let n be the total number of processes and m be the total number of resource types in the system.
Available: A vector of length m. It shows number of available resources of each type. If Available[i] = k, then k instances of resource Ri are available.

Max: An n×m matrix that contain maximum demand of each process. If Max[i,j] = k, then process Pi can request maximum k instances of resource type Rj.

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

Need: An n×m matrix that shows the remaining resource need of each process. If Need[i,j] = k, then process Pi may need k more instances of resource type Rj to complete the task.

Algorithm:

Description (purpose of use):

Steps of Algorithm:

1. Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.
   Initialize: Work= Available

   Finish [i]=false; for i=1,2,……,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 safe state.

   Now,

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
P1-200→ 5 3 2
         P4-002→5 3 4
                 P3-211→7 4 5
   --Deadlock Detected--
```

```
  3 3 2        --Available
P3-211→5 3 2
       P1-200→5 3 4
               P4-002→7 4 5
  --Deadlock Detected—
```

. . . It is designed to check the safe state whenever a resource is requested. It takes analogy of bank, where customer request to withdraw cash. Based on some data the cash is lent to the customer. The banker can’t give more cash than what the customer has requested for, and the total available cash. As this algorithm uses bank analogy so named as banker’s algorithm.

Code snippet:

```
#include<iostream>

using namespace std;

//        Number of processes const int P = 5;

//        Number of resources const int R = 3;
//        Function to find the need of each process

void calculateNeed(int need[P][R], int maxm[P][R], int allot[P][R]) {

//        Calculating Need of each P for (int i = 0 ; i < P ; i++)
```

```
for (int j = 0 ; j < R ; j++)

//          Need of instance = maxm instance - //allocated instance

need[i][j] = maxm[i][j] - allot[i][j]; }

//          Function to find the system is in safe state or not bool isSafe(int
processes[], int avail[], int maxm[][R],

int allot[][R]) { int need[P][R];

//          Function to calculate need matrix calculateNeed(need, maxm, allot);
//          Mark all processes as infinish

bool finish[P] = {0};

//          To store safe sequence int safeSeq[P];


//          Make a copy of available resources int work[R];

for (int i = 0; i < R ; i++) work[i] = avail[i];
//          While all processes are not finished

//          or system is not in safe state.

int count = 0;

while (count < P) {

bool found = false;

for (int p = 0; p < P; p++) {

//          First check if a process is finished,

//          if no, go for next condition

if (finish[p] == 0) {

//          Check if for all resources of

//          current P need is less
//	than work

int j;

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

if (need[p][j] > work[j])

break;

//	If all needs of p were satisfied. if (j == R) {

for (int k = 0 ; k < R ; k++) work[k] += allot[p][k];


//	Add this process to safe sequence. safeSeq[count++] = p;

//	Mark this p as finished

finish[p] = 1;

found = true; } } }

if (found == false){

cout << "System is not in safe state";

return false; } }

//	If system is in safe state then

//	safe sequence will be as below

cout << "System is in safe state.\nSafe"

" sequence is: ";

for (int i = 0; i < P ; i++)

cout << safeSeq[i] << " ";

return true;}

//	Driver code int main() {
```

```
int processes[] = {0, 1, 2, 3, 4};

//          Available instances of resources int avail[] = {3, 3, 2};

//          Maximum R that can be allocated int maxm[][R] = {{7, 5, 3},

{3, 2, 2}, {9, 0, 2},

{2, 2, 2},

{4, 3, 3}};

//          Resources allocated to processes int allot[][R] = {{0, 1, 0},

{2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2}};

//          Check system is in safe state or not isSafe(processes, avail, maxm, allot);
return 0;}
```

Output:

```
System is in safe state.

Safe sequence is: 1 3 4 0 2
```

Description:

There are 5 processes and 3 resource types, resource A with 10 instances, B with 5 instances and C with 7 instances. In three resource types A,B,C suppose at t0 following is shown:

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

GitHub Link: https://github.com/saipranup/OS-Bankers-problem