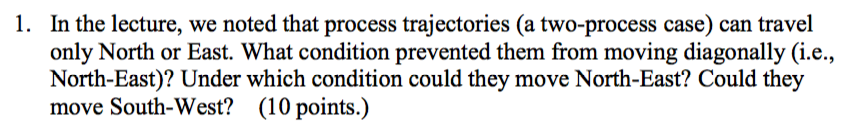
CS 520 Homework 4 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 10/22/2017



**Solution**:

1] Parallel processing/Parallel thread execution is not possible since we had considered single CPU for the two-process system. So, the trajectories can travel only North/East.

2] If there is addition of one more CPU, such that both processes are executed parallelly then the trajectory can move North-East/diagonally.

3] They **CANNOT move South-West.** That is because the axes used, utilize resource with **respect to time.** So as time progresses, the trajectory won’t be able to move south-west.

../Screen%20Shot%202017-10-22%20at%2011.30.05%20PM.png

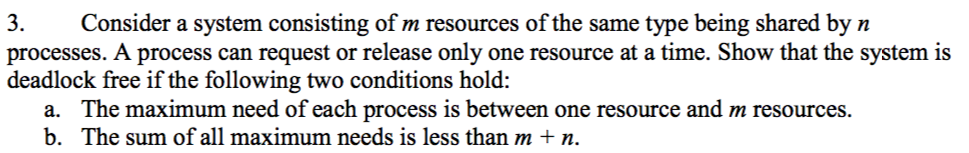
**Solution**:

1] Creating a program and showing that the above problem is deadlock free will take quite a lot of time & it is a tedious task. Proving it by a contradiction is much simpler in this case. That is, we say that the system is in deadlock but prove it false by proof.

2] As per the question, every process is holding one resource and needs one more resource to complete its own task and there is one spare resource available.   
Since every process requires max of two resources, the remaining resource can be utilized by any of the process and it can complete its task.

3] The process will release both of the resources after completion. The other two processes can now utilize the resources to complete their tasks.

**The assumption is hence contradicted**. Hence the system is not in Deadlock.



**Solution**:

A] A process will never be able to complete its work without any resource.

We can two assume two Cases for this.

1] Best Case: A process uses only **one** resource from the system.

Worst Case: So, to complete the work, A process will need to use all the available resources within the system which is denoted by **m**.

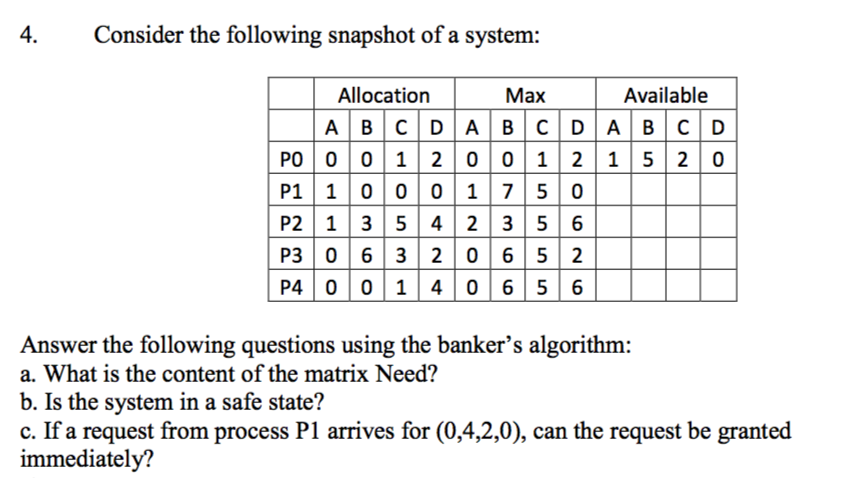
Therefore, the maximum need of any process will be between 1 and m.

B] Since we proved above that the maximum need of any process will be between 1 and m,

Need = Max – Allocation

So, Need + Allocation = Max

So, Max = Need + Allocation = n + m.



A] The definition of Need matrix: an m × n matrix, which indicates the remaining resources needed by each process to complete its work.

If Need[i][j] = k, it means that Process Pi needs k number of resources of type Rj. It’s calculated as Need[i][j] = Max[i][j] – Allocation[i][j]

So the need matrix is,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

B] **YES**, The system in the safe state.

The resources available are (1,5,2,0).

Work[i][j] = Max[i][j] – Allocation[i][j] + Available[i][j].

If Work[i][j] is greater than 0, then process can finish its own work.

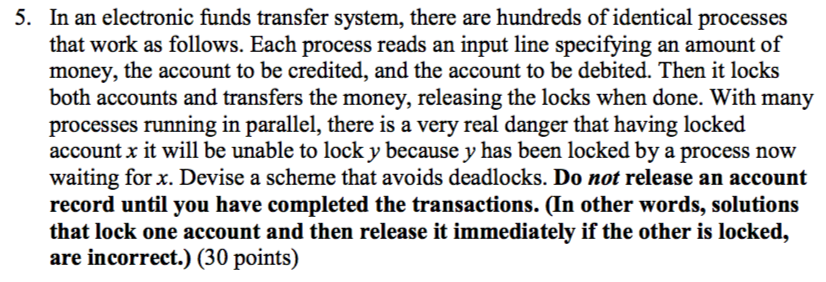
Hence, with the available matrix as (1,5,2,0),

Therefore, either P0 or P1 can finish work and release the resources.

This will allow other processes to finish their work.

**Therefore, it can be stated that the system is in safe state.**

C] **Yes, the request to P1 with (0,4,2,0) can be granted immediately.**   
As Available (1,5,2,0) is still greater than the requested, P1 can be processed immediately. Once P1 has completed its processing, Available becomes (1,9,4,0). Then any one of the P0 or P2 can be processed. And system will continue to remain in a safe state.



1] The most best and optimum solution to solve this problem is by using priority queue so that we can check lock on both accounts. The object creates a priority queue which has the time of transfer request initiated by any process and transfer amount.

2] Priority should be given to accounts which has higher amounts to transfer. Also, the request was queued at the earliest.

3] We firstly check if both the accounts are available or not whenever a process is requesting to transfer money between two accounts, i.e. no account has lock. We don’t proceed for transfer if any of the account has lock present on it rather we change the request time to future & sort the queue again by request time and amount to transfer.

But please note that if both of the accounts are free, then we can acquire the lock and transfer money.

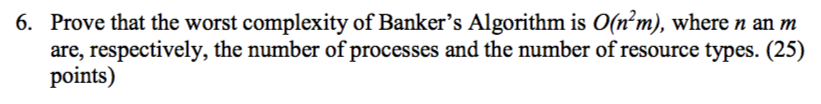
**The Algorithm we use**:

* Add process to queue with its time of request.
* Sort the queue by request time and transfer amount.
* Choose the top most process.
* Obtain account numbers & check if any of the account has lock or no.
* If any of the account has lock present on it, then increase the request time by ‘x’.
* Once more rearrange the queue by request time and transfer amount.
* If both accounts are free, then acquire the lock and then transfer the amount.
* Once amount is transferred, remove the process from queue.

4] Using this method, any process won’t be able to block the accounts unless both the accounts are unlocked thus avoiding deadlock.

Both accounts are checked before acquiring lock, locking & releasing of the lock is avoided on any of the account.

5] All the processes will be executed one by one **since we sort the queue**. We can ensure that, periodic check on accounts is done to acquire the lock by changing the request time in the future.



As said before, The Banker's algorithm, sometimes referred to as the detection algorithm, is a resource allocation and deadlock avoidance algorithm.

As there is no need to hardcode the entire program, I have given and explained the most important part of the pseudocode and why it will have a worst complexity of O(n2m)

Pseudocode for Banker’s Algo:

do

{

look for an unmarked process Pi such that

if (found)

{

mark Pi; /\* Consider it finished \*/

A = A + Ci; Ci = 0; j++; /\*Take its resources\*/

}

else

loop = FALSE;

} while loop && (j<n);

Elaborated Code in C Sharp:

safe = false;

for (int IndexCount = 0; IndexCount < NumberOfProcesses; IndexCount++) **//#1//**

{

if (running[IndexCount].Equals(1))

{

exec = 1;

for (int InIndex = 0; InIndex < NumberOfResources; InIndex++) **//#2//**

{

if (max\_claim[IndexCount, InIndex] - \_curr[IndexCount, InIndex] > avl[InIndex])

{

exec = 0;

break;

}

}

if (exec.Equals(0))

{

Console.WriteLine("Process 0 is executing.", IndexCount++);

running[IndexCount] = 0;

count--;

safe = true;

for (int InIndex = 0; InIndex < NumberOfResources; InIndex++)

avl[InIndex] += \_curr[IndexCount, InIndex]; **//#3//**

break;

}

}

}

Explanation for solution:

When the program executes **#1**, the loop runs **n** times for process 1,   
**n-1** times for process 2, n-2 times for process 3, and so on.

The total it will be run for **(n(n+1))/2** times. i.e. **(n^2)/2 + n/2**.

Hence complexity generated within the loop is **O**(**n^2**).

This looping will execute for ‘m’ resources as indicated in **#2**.

Thus, total complexity is **O(n2m).**