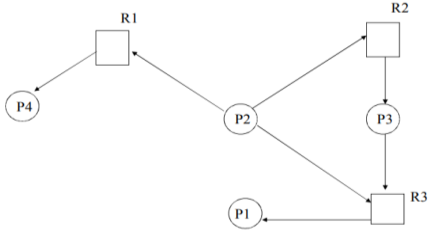
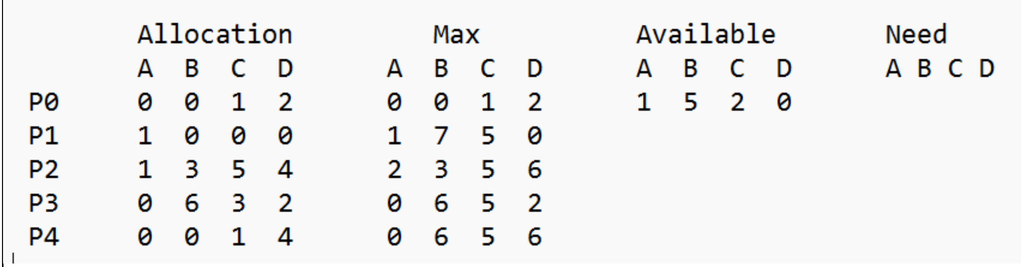
USN

Internal Assessment Test 2 – January 2024

<table>
<tr><td>Sub:</td><td colspan="4"><b>Operating Systems</b></td><td>Sub Code:</td><td>BCS303</td><td>Branch:</td><td colspan="2">ISE</td></tr>
<tr><td>Date:</td><td><b>18/1/2024</b></td><td>Duration:</td><td>90 min’s</td><td>Max Marks: 50</td><td>Sem/Sec:</td><td colspan="2">III A, B &amp; C</td><td colspan="2">OBE</td></tr>
<tr><td colspan="7"><b><u>Answer any FIVE FULL Questions</u></b></td><td>MARKS</td><td>CO</td><td>RBT</td></tr>
<tr><td>1</td><td colspan="6">Consider the following set of processes, with the length of the CPU burst is given below in milliseconds. The processes are assumed to have arrived in the order P1, P2,P3. Draw a Gantt Chart and calculate the average waiting time and average turnaround time for all processes using Round Robin (quantum = 2 ms) (6 marks) and SJF(4 marks).
<table>
<tr><td>Process</td><td>Arrival</td><td>Burst</td></tr>
<tr><td>P1</td><td>0</td><td>5</td></tr>
<tr><td>P2</td><td>1</td><td>10</td></tr>
<tr><td>P3</td><td>2</td><td>2</td></tr>
<tr><td>P4</td><td>3</td><td>1</td></tr>
</table></td><td>10</td><td>CO2</td><td>L3</td></tr>
<tr><td>2.a</td><td colspan="6">Differentiate Multilevel Queue Scheduling and Multilevel Feedback Queue Scheduling
5 differences – 5 marks</td><td>5</td><td>CO2</td><td>L2</td></tr>
<tr><td>2.b</td><td colspan="6">Explain the requirements to solve Critical Section Problem
Conditions – 5 marks</td><td>5</td><td>CO3</td><td>L2</td></tr>
<tr><td>3</td><td colspan="6">Outline a solution using Semaphores to solve Dining Philosopher’s Problem
Problem definition : 2 marks
Philosopher Code : 3 marks
Semaphore solution with explanation : 5 marks</td><td>10</td><td>CO3</td><td>L2</td></tr>
<tr><td>4.a</td><td colspan="6">Explain the several data structures maintained to encode the state of a resource with several instances.
Four data structures : 4</td><td>4</td><td>CO3</td><td>L2</td></tr>
<tr><td>4.b</td><td colspan="6">Derive the algorithm using test-and –set () instruction that satisfy all the critical section requirements.
Definition Test() and Set() :2 marks
Mutual Exclusion / Bounded wait Mutual Exclusion algorithm with explanation : 4 marks</td><td>6</td><td>CO3</td><td>L2</td></tr>
<tr><td>5.a</td><td colspan="6">Consider the below given resource- allocation graph in which all resources have only one instance. Determine whether the system has deadlock or not by using deadlock detection algorithm. Illustrate the process with a neat sketch.


Process Explanation : 3
Wait For Graph : 2
Deadlock / No deadlock : 1</td><td>6</td><td>CO2</td><td>L3</td></tr>
<tr><td>5.b</td><td colspan="6">Explain Process Termination to eliminate deadlocks
Causes : 4 marks</td><td>4</td><td>CO3</td><td>L2</td></tr>
<tr><td>6</td><td colspan="6">Consider the following system snapshot using data structures in the Banker’s algorithm with resources A, B, C and D and process P0 to P4:</td><td>10</td><td>CO3</td><td>L3</td></tr>
</table>



```
        Allocation          Max             Available        Need
        A  B  C  D      A  B  C  D      A  B  C  D       A B C D
P0      0  0  1  2      0  0  1  2      1  5  2  0
P1      1  0  0  0      1  7  5  0
P2      1  3  5  4      2  3  5  6
P3      0  6  3  2      0  6  5  2
P4      0  0  1  4      0  6  5  6
```

Using Banker's algorithm, answer the following questions:

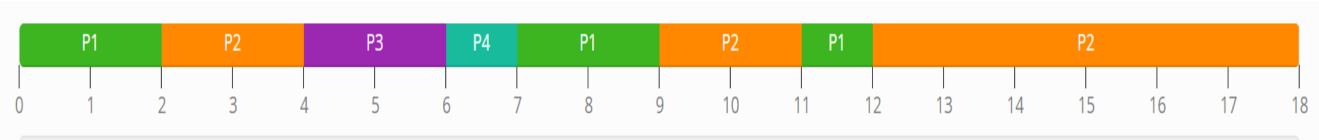
(i) What are the contents of Need Matrix? (2 marks)

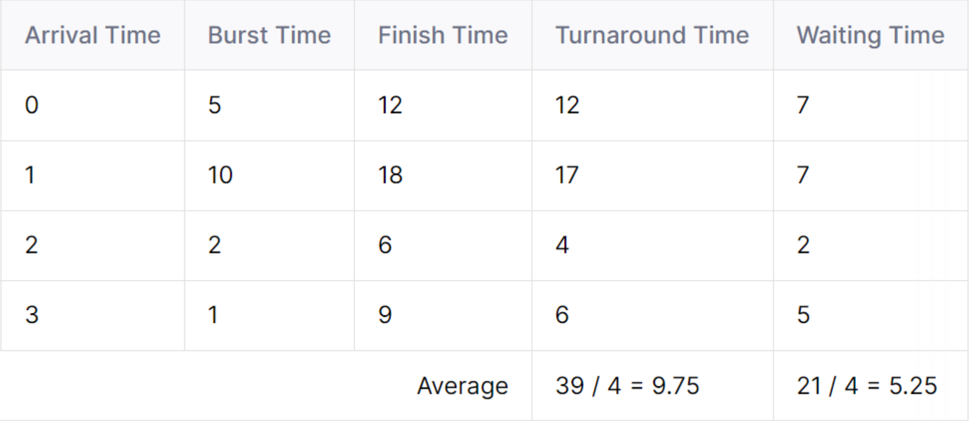
(ii) Is the in a safe state? (2 marks) Explain the algorithm and determine safe sequence order in which the processes completes its execution?(6 marks)

Solution

1,

Round Robin :

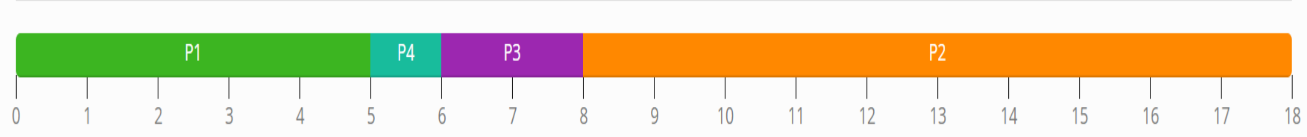


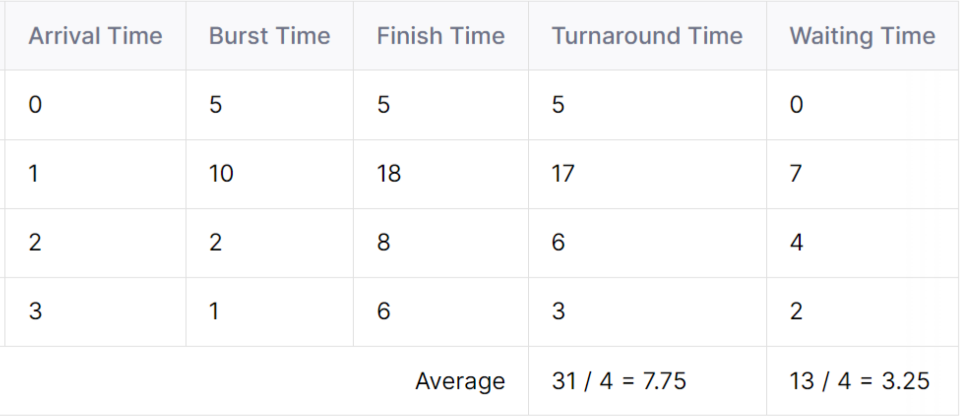


Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
0	5	12	12	7
1	10	18	17	7
2	2	6	4	2
3	1	9	6	5
		Average	39 / 4 = 9.75	21 / 4 = 5.25

SJF:

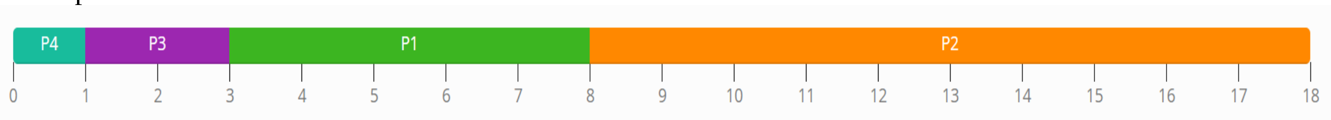
Non Preemptive

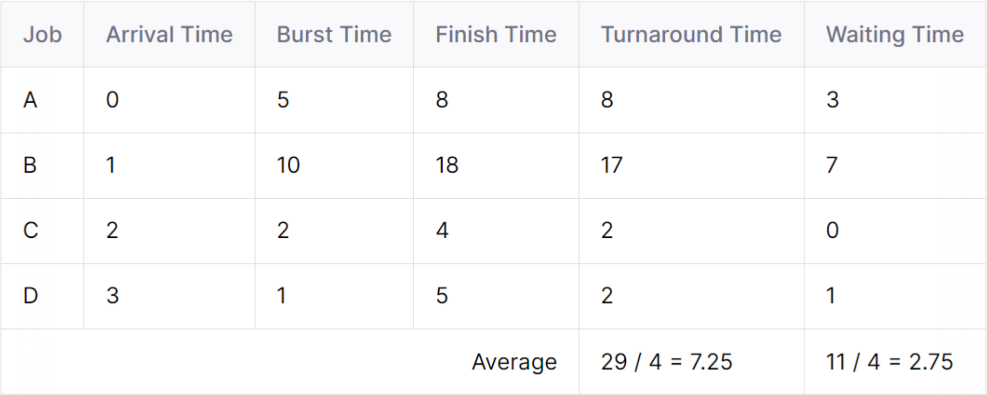




Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
0	5	5	5	0
1	10	18	17	7
2	2	8	6	4
3	1	6	3	2
		Average	31 / 4 = 7.75	13 / 4 = 3.25

Preemptive

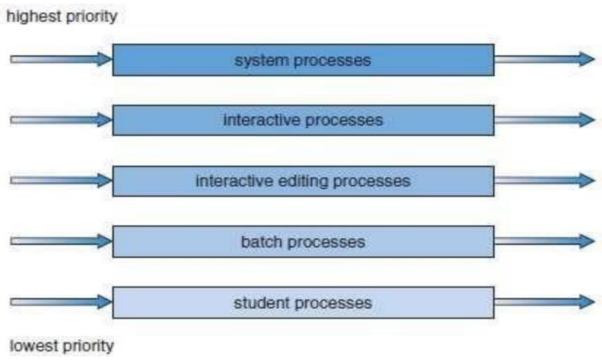
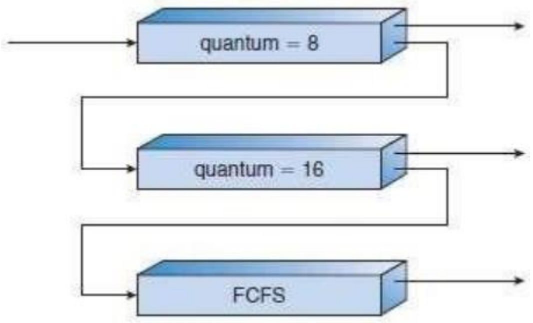




Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	5	8	8	3
B	1	10	18	17	7
C	2	2	4	2	0
D	3	1	5	2	1
			Average	29 / 4 = 7.25	11 / 4 = 2.75

2a.

Multilevel queue scheduling (MLQ)	**Multilevel feedback queue scheduling (MLFQ)**
It is queue scheduling algorithm in which ready queue is partitioned into several smaller queues and processes are assigned permanently into these queues.	In this algorithm, ready queue is partitioned into smaller queues on basis of CPU burst characteristics.
The processes are divided on basis of their intrinsic characteristics such as memory size, priority etc.	The processes are not permanently allocated to one queue and are allowed to move between queues.
In this algorithm queue are classified into two groups, first containing background processes and second containing foreground processes.	Here, queues are classified as higher priority queue and lower priority queues. If process takes longer time in execution it is moved to lower priority queue.

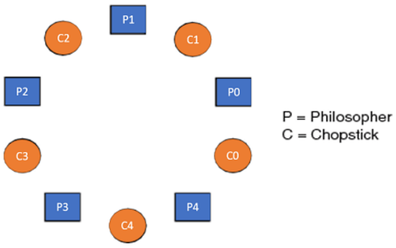
Multilevel queue scheduling (MLQ)	**Multilevel feedback queue scheduling (MLFQ)**
The priority is fixed in this algorithm.	The priority for process is dynamic as process is allowed to move between queue.
Since, processes do not move between queues, it has low scheduling overhead and is inflexible.	Since, processes are allowed to move between queues, it has high scheduling overhead and is flexible.
 highest priority system processes interactive processes interactive editing processes batch processes student processes lowest priority	

2b.

Critical section problem may be resolved by satisfying the following three requirements:

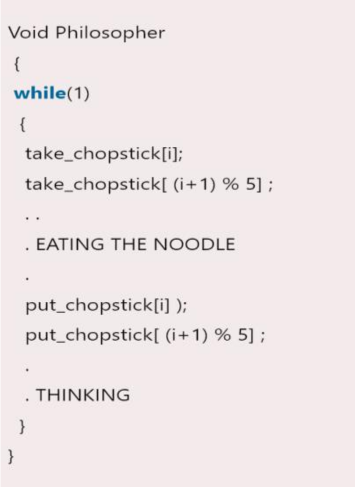
- **Mutual Exclusion:** If a process is executing in its critical section, then **no other process is allowed** to execute in the critical section.
- **Progress:** ***If no process is executing in the critical section*** and other processes are waiting outside the critical section, ***then only those processes that are not executing in their remainder section*** **can participate in deciding** which will enter the critical section next, and **the selection cannot be postponed** indefinitely.
- **Bounded Waiting:** A **bound must exist** on **the number of times** that **other processes are allowed to enter** their **critical sections** after a process has made a request to enter its critical section and before that request is granted.

3.**Dining Philosopher Problem:**



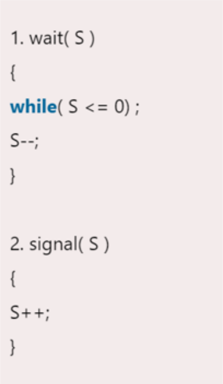
- The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively.
- A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick.

- A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.
- The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.

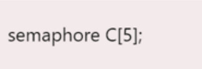


```
Void Philosopher
 {
 while(1)
  {
   take_chopstick[i];
   take_chopstick[ (i+1) % 5] ;
   . .
   . EATING THE NOODLE
   .
   put_chopstick[i] );
   put_chopstick[ (i+1) % 5] ;
   .
   . THINKING
  }
}
```

Semaphore code:

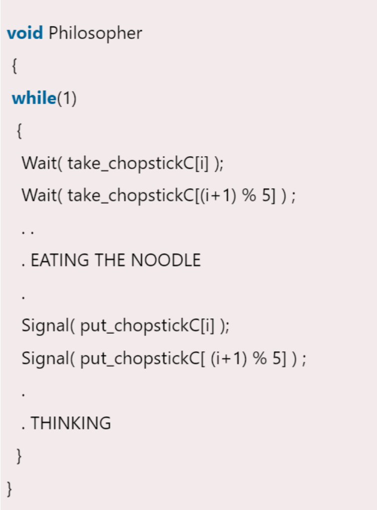


```
1. wait( S )
{
while( S <= 0) ;
S--;
}

2. signal( S )
{
S++;
}
```



```
semaphore C[5];
```

Solution :



```
void Philosopher
 {
 while(1)
  {
   Wait( take_chopstickC[i] );
   Wait( take_chopstickC[(i+1) % 5] ) ;
   . .
   . EATING THE NOODLE
   .
   Signal( put_chopstickC[i] );
   Signal( put_chopstickC[ (i+1) % 5] ) ;
   .
   . THINKING
  }
}
```

1. Initialize the semaphores for each fork to 1 (indicating that they are available).
2. Initialize a binary semaphore (mutex) to 1 to ensure that only one philosopher can attempt to pick up a fork at a time.
3. For each philosopher process, create a separate thread that executes the following code:
   - While true:
     - Think for a random amount of time.
     - Acquire the mutex semaphore to ensure that only one philosopher can attempt to pick up a fork at a time.
     - Attempt to acquire the semaphore for the fork to the left.
   - If successful, attempt to acquire the semaphore for the fork to the right.
   - If both forks are acquired successfully, eat for a random amount of time and then release both semaphores.
   - If not successful in acquiring both forks, release the semaphore for the fork to the left (if acquired) and then release the mutex semaphore and go back to thinking.
4. Run the philosopher threads concurrently.

4a)

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Ri are available.

Max: An n x m matrix defines the maximum demand of each process. If Max[i] [j] equals k, then process P; may request at most k instances of resource type Ri.

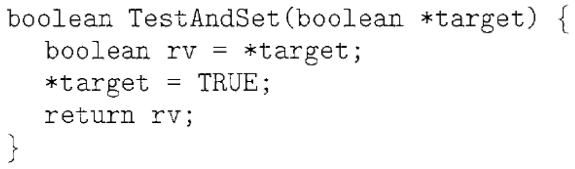
Allocation: An 11 x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals lc, then process P; is currently allocated lc instances of resource type Rj.

Need: An n x m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process P; may need k more instances of resource type Ri to complete its task. Note that Need[i][j] equals Max[i][j] - Allocation [i][j].

These data structures vary over time in both size and value.

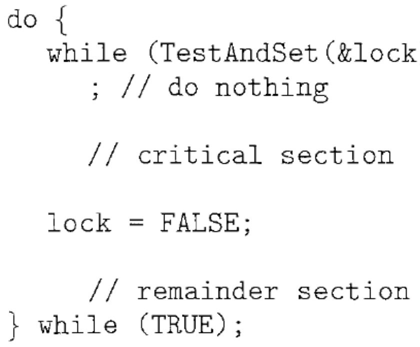
4b.

Definition:



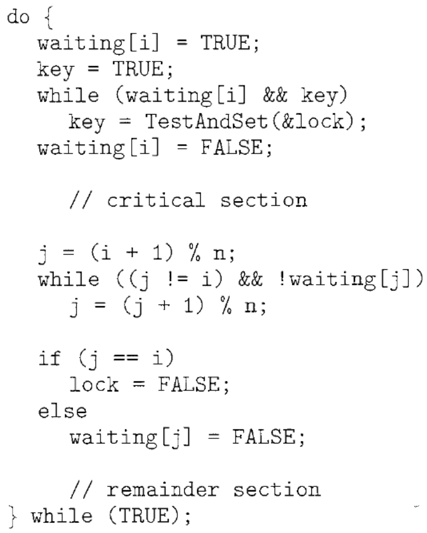
```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Mutual Exclusion



```
do {
   while (TestAndSet(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

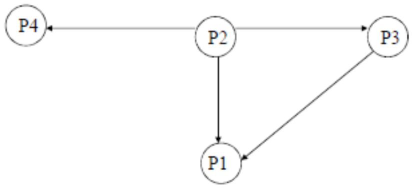
Bounded – Waiting Mutual Exclusion:



```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
} while (TRUE);
```

5a.

Algorithm:
Step 1: Take the first process (Pi) from the resource allocation graph and check the path in which it is acquiring resource (R_i), and start a wait-for-graph with that particular process.
Step 2: Make a path for the Wait-for-Graph in which there will be no Resource included from the current process (P_i) to next process (P_j), from that next process (P_j) find a resource (R_j) that will be acquired by next Process (P_k) which is released from Process (P_j).
Step 3: Repeat Step 2 for all the processes.
Step 4: After completion of all processes, if we find a closed-loop cycle then the system is in a deadlock state, and deadlock is detected.



No cycle, therefore No Deadlock.

5b.
To eliminate the deadlock, we can simply kill one or more processes. For this, we use two methods:

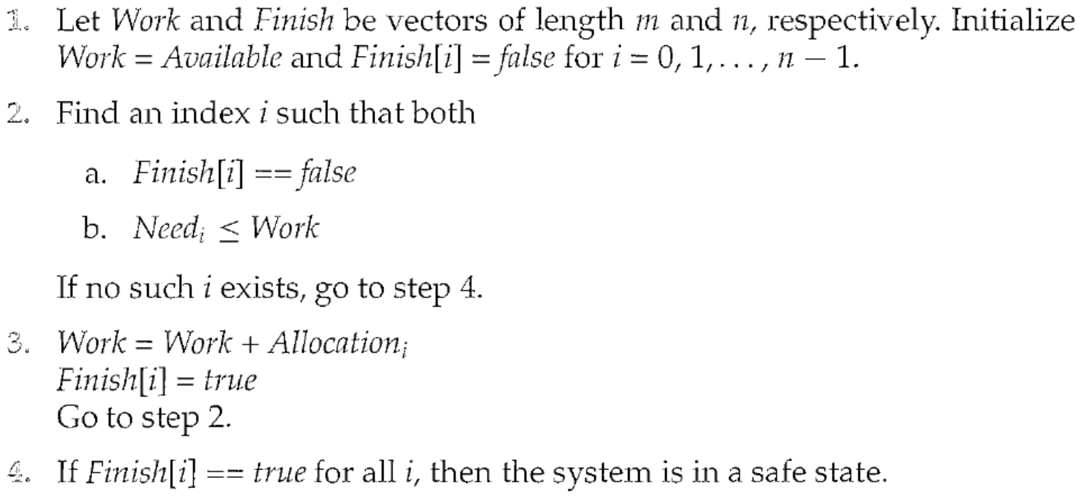
1. **Abort all the Deadlocked Processes**: Aborting all the processes will certainly break the deadlock but at a great expense. The deadlocked processes may have been computed for a long time, and the result of those partial computations must be discarded and there is a probability of recalculating them later.

2. **Abort one process at a time until the deadlock is eliminated**: Abort one deadlocked process at a time, until the deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead, because, after aborting each process, we have to run a deadlock detection algorithm to check whether any processes are still deadlocked.

6)
(i) Need Matrix

A	B	C	D
0	0	0	0
0	7	5	0
1	0	0	2
0	0	2	0
0	6	4	2

(ii) The system is not in safe state.
Algorithm :



1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available* and *Finish*[i] = *false* for $i = 0, 1, \ldots, n-1$.

2. Find an index i such that both

   a. *Finish*[i] == *false*

   b. $Need_i \leq Work$

   If no such i exists, go to step 4.

3. $Work = Work + Allocation_i$
   Finish[i] = *true*
   Go to step 2.

4. If *Finish*[i] == *true* for all i, then the system is in a safe state.

Safety Sequence order:
P0→P2→P3→P4→P1