OS LAB ASSIGNMENT – 5

DEADLOCKS

Q1. Multiple Producers and Consumers

Problem Definition:

You have to implement a system which ensures synchronisation in a producer-consumer scenario. You also have to demonstrate deadlock condition and provide solutions for avoiding deadlock. In this system a main process creates 5 producer processes and 5 consumer processes who share 2 resources (queues). The producer's job is to generate a piece of data, put it into the queue and repeat. At the same time, the consumer process consumes the data i.e., removes it from the queue. In the implementation, you are asked to ensure synchronization and mutual exclusion. For instance, the producer should be stopped if the buffer is full and that the consumer should be blocked if the buffer is empty. You also have to enforce mutual exclusion while the processes are trying to acquire the resources.

Manager (manager.c):

It is the main process that creates the producers and consumer processes (5 each). After that it periodically checks whether the system is in deadlock. **Deadlock** refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain.

Implementation : The manager process (manager.c) does the following :

i. It creates a file matrix.txt which holds a matrix with 2 rows (number of resources) and 10 columns (ID of producer and consumer processes). Each entry (i, j) of that matrix can have three values:

- 0 => process i has not requested for queue j or released queue j
- 1 => process i requested for queue j
- 2 => process i acquired lock of queue j.

The manager initializes all entries to 0.

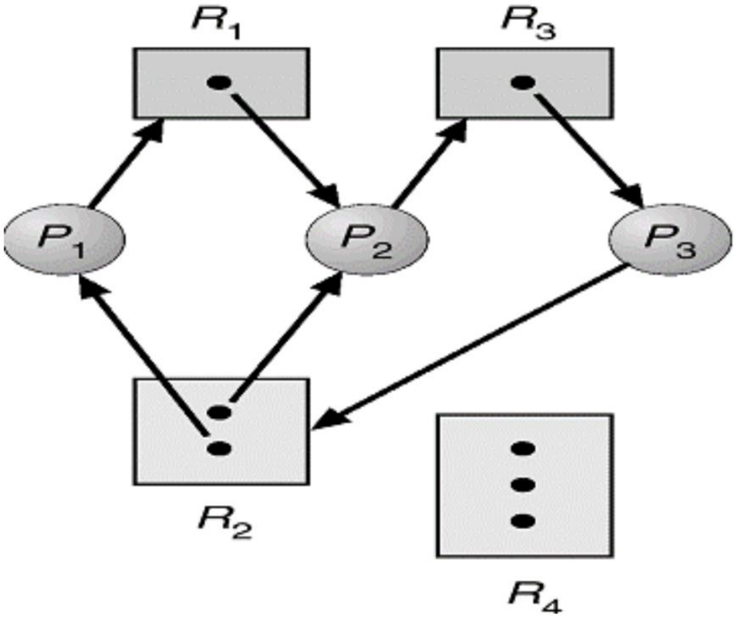
ii. It creates all the producer / consumer processes. It also creates two message queues (Buffer-size 10) and share them among those processes.

iii. Periodically, It generates a resource allocation graph from the matrix in matrix.txt and checks for cycles in that graph (after every 2 seconds) and prints the cycle if it has found a deadlock in the system.

Once the manager detects a deadlock, it prints the entire cycle. It then kills all the subprocesses (both producer and consumer) and exits.

Note: A resource allocation graph tracks which resource is held by which process and which process is waiting for a resource of a particular type. If a process is *using* a resource, a directed link is formed from the resource node to the process node. If a process is *requesting* a resource, a directed link is formed from the process node to the resource node. If there is a cycle in the Resource Allocation Graph then the processes will deadlock.

In the following Resource Allocation Graph R stands for a resource and P stands for a process. Here one of the deadlock cycles is R1 → P2 → R3 → P3 → R2 → P1 → R1.



Producer (producer.c):

The producer's job is to select one of the queues at random and to insert a random number between 1-50, if the queue is not full. It waits if any other process is currently holding the queue.

Before inserting, it prints "Trying to insert…" and after inserting "Successfully inserted.", with the process and queue numbers. It then goes to sleep for a random amount of time and repeats.

Mutual exclusion can be ensured using semaphores.

Consumer (consumer.c):

The consumer's job is to remove the element from the queue, if it's not empty. It can either consume from one queue with p probability or it can consume from both the queues with probability (1-p). If a consumer consumes from both queues, it follows the following algorithm:

- Request lock on a random queue, q.
- Consume item from q.
- Request lock on the remaining queue, q'.
- Consume item from q'.
- Release lock on q.
- Release lock on q'.

The locks can be implemented using semaphores.

All producer and consumer processes update the matrix (goes byte by byte to the proper location of the file and update the entry OR reads the entire file and replace it with the updated matrix) in the shared file, whenever they request for, acquire or release a queue. The mutual exclusion, while accessing this shared file, must be ensured (you might use one semaphore to ensure this).

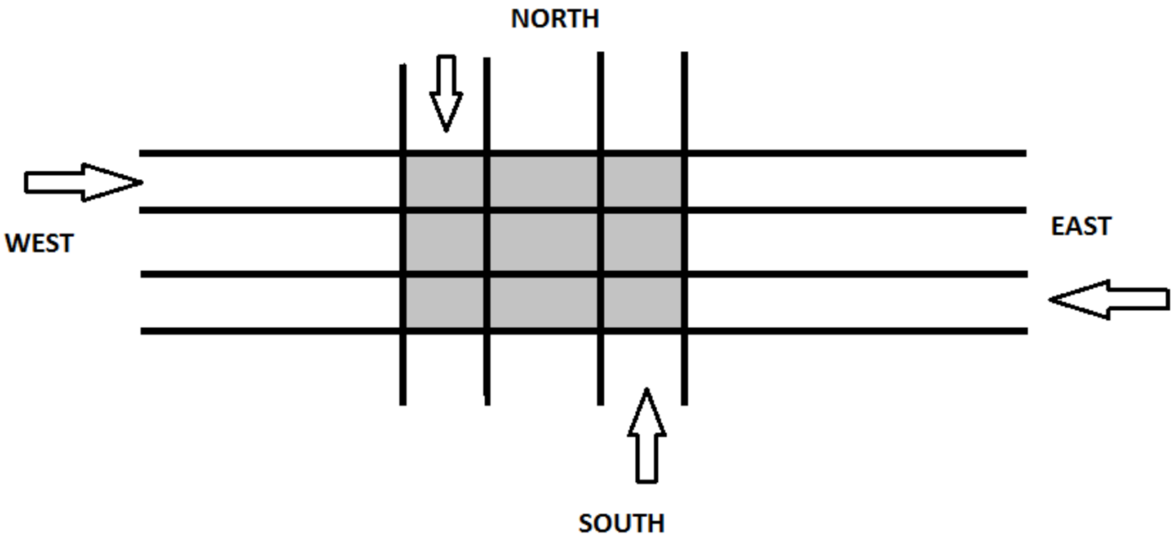
Appropriate messages should be printed as in the producer case.

Deadlock Avoidance Scheme :
You have to implement two variation of the solution. First one, allowing deadlock (case I), second one (case II), implement the following protocol to prevent deadlock. Suppose the queues have ids as Q0 and Q1. If any consumer decides to consume from both the queues, it always requests for Q0 first. In this case deadlock will never happen as the "circular wait" condition never satisfies.

Evaluation :
Case I : No deadlock avoidance. You have to print the element produced by the producer and consumed by the consumer. Also you have to give proper sleep to observe the deadlock. You should vary p from 0.2 to 0.8 with a step of 0.1. For each p value, count the total number of inserts and deletes for both the queues before the deadlock occurs and write it in a file (result.txt). You also have to print the cycle for which the deadlock has happened (as detected from the matrix.txt).

Case II : Use deadlock prevention protocol and show that the system runs without deadlock.

Q2. RAIL-MANAGER



Problem Definition:

Assume, you have the above rail-crossing scenario. Trains may come from all the four different directions. You have to implement a system where it never happens that more than one train crosses the ***junction (shaded region)*** at the same time. Every train coming to the junction, waits if there is already a train at the junction from **its own direction.** Each train also gives a precedence to the train coming from its **right side** (for example, right of North is West) and waits for it to pass. You also have to check for deadlock condition if there is any.

Each train is a separate process. Your task will be to create a manager which creates those processes and controls the synchronisation among them.

Semaphores:

You have total two types of semaphores. One for ensuring the mutual exclusion at the junction to cross and the other for synchronising the trains. There are four synchronising semaphores for trains coming from 4 directions- North, South, East and West.

Manager process (manager.c):

1. It takes as input a file sequence.txt with sequence of directions of trains coming to the junction. The input file format is the following - If it contains “NWEWS”, it means 5 train processes will be created in the order North-West-East-West-South.

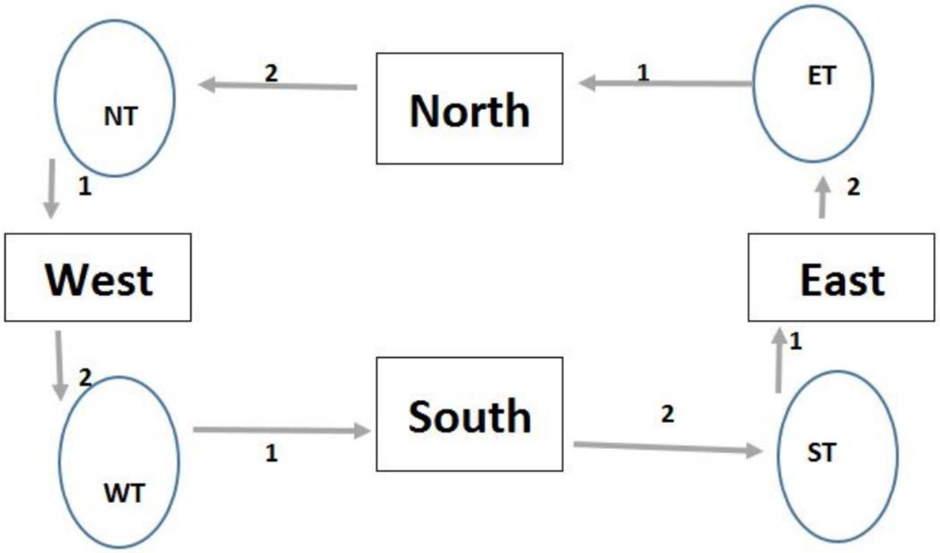
2. It also creates a file matrix.txt which contains a matrix of size n X m where n is the number of trains read from sequence.txt and m is the number of synchronising semaphores used (4 in our case).
   Each entry (i,j) of that matrix can have three values:
   - 0 => train i has not requested for semaphore j or released semaphore j
   - 1 => train i requested for semaphore j
   - 2 => train i acquired lock of semaphore j.

   Initially all entries of the matrix are set to 0. When any train process releases a semaphore the corresponding matrix entry goes back from 2 to 0.

3. At each iteration with p probability it checks for deadlock in the system and with 1-p probability it creates the next train process as per the direction specified in the input file. Once all the train processes specified in the input file are created, the manager only checks for deadlock periodically(at 1 second interval).
4. Deadlock detection:
   If every train (process) waits for a train coming from its right-side to pass, the system gets into a deadlock. Assume the semaphores to be the resources and the trains as the processes and generate the resource allocation graph (use the same procedure described in Q1."NOTE"). The manager process checks for cycle in the resource allocation graph built from the matrix present at matrix.txt.
   A possible deadlock scenario is the following. Assume there are four trains NT, WT, ST & ET coming from four different directions and four semaphores are East,West,North,South. Say at one instance the matrix looks as follows.

		Semaphores			
		North	West	South	East
Trains	NT	2	1	0	0
	WT	0	2	1	0
	ST	0	0	2	1
	ET	1	0	0	2

For the above matrix following is the Resource allocation graph. Rectangle is the resource ie semaphore and circle is the train ie process. Edge weight 1 means the process has requested for the resource and edge weight 2 means the process has acquired the resource. Now we can see there is a cycle in the following graph for the above matrix, hence deadlock exists.



After detecting the deadlock the manager prints the cycle.

Train process (train.c):

The steps followed by a train coming from North are the following:

1. It waits at the “North” semaphore.
2. Then it waits at the “West” semaphore.
3. Finally it checks for mutual exclusion at the junction.
4. It crosses the junction. The time taken to cross a junction is implemented by a sleep of 2 second.
5. When the train leaves the junction, it releases the mutual exclusion lock and the “North” semaphore.
6. Whenever the train requests for, acquires or releases a semaphore it updates the proper (i,j) th entry in the matrix.txt. The mutual exclusion while accessing this file should be ensured (you might use one more semaphore to ensure this).

Evaluation:

Vary p (taken as command line argument) from 0.2 to 0.7 in scale of 0.1 and print the following

i) (train) Process creations.
ii) The trains arriving, crossing & leaving the junction.
iii) Deadlock detection & corresponding cycle.

A sample output:

```
        ……..
        Train <pid1> North train started
        Train <pid2>: West train started
        Train <pid1>: requests for North-Lock
        Train <pid1>: Acquires North-Lock
        Train <pid2>: requests West-Lock
        Train <pid2>: acquires West-Lock
        Train <pid2>: requests South-Lock
        Train <pid2>: acquires South-Lock
        Train <pid1>: requests West-lock
        Train <pid2>: requests Junction-Lock
        Train <pid2>: acquires Junction-Lock; Passing Junction;
        Train <pid2>: releases junction-lock
        Train <pid2>: releases West-lock
        Train <pid1>: Acquires West-Lock
        Train <pid1>: requests Junction-Lock
        Train <pid1>: acquires Junction-Lock; Passing Junction;
        Train <pid1>: releases junction-lock
        Train <pid1>: releases North-lock
        ……...
        System Deadlocked
        <print the cycle >
                (for eg. - Train<pid10> from North  is waiting for Train<pid11> from West ------
        ---> Train<pid11> from West is Waiting for Train<pid12> from South --------->
        Train<pid12> from South is Waiting for Train<pid13> from East --------->
        Train<pid13> from East is Waiting for Train<pid10> from North  )
```

note: "North train started" means a train process is created and it is coming from North.

3 sample input sequences: i) NNSSNSS   ii) NWESNWESNWES iii) NESWNESWNESW