Indian Institute of Information Technology Guwahati
Operating Systems Lab (CS232)
CG1 / Assignment 4

Floating date: 16th March, 2020
Deadline: 30th March, 2020

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.

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. In this assignment we will implement **Floyd-Warshall All-Pairs-Shortest-Path algorithm in a multi-threaded fashion** along with enforcing **the readers-writers problem.** The single-threaded version is given below:

Algorithm: Floyd-Warshall (FW) algorithm

```
1.  for k = 1 to n do
2.      for i = 1 to n do
3.          for j = 1 to n do
4.              if (dist[i][k] + dist[k][j] < dist[i][j])
5.                  dist[i][j] = dist[i][k] + dist[k][j]
6.          end for
7.      end for
8.  end for
```

Here the outer k loop denotes that we are including k as an intermediate vertex in this iteration. dist matrix represents the distance matrix (dist[i][j] = minimum distance between i and j in the graph till the current iteration). We need to maintain only 1 copy of the matrix. In iteration number k, we only need to

read values from the kth column and the kth row of the matrix and we can update values (i,j) in the same copy of the matrix without any inconsistency.

Multi-threaded implementation:

The k loop remains same. Here instead of the i loop, we create n threads that represent each iteration of the i loop. Each thread runs the j loop from 1 to n. There is a global matrix *Graph* that stores the adjacency matrix for the graph. All the threads can read it simultaneously. There is another global matrix *dist* that is used for updating the minimum distances between the vertices. **Only 1 copy** of dist will suffice.

At the start of every iteration inside the k loop, you need to create n threads. Each will run it's own j loop. You have to enforce readers-writers problem in the manner of how the dist matrix is accessed by the different threads. Any number of threads can read from the dist matrix provided that no other thread is writing to it. Only 1 thread can write to the dist matrix at a time. If you observe closely, line 4 has only read statements and line 5 is the only write statement present in the algorithm.

At the end of every iteration of the k loop, you need to wait for all the threads to finish. Then only can you start another iteration of the k loop. So you need to join the threads at the end of every iteration of the k loop.

Input Format:

Graph structure: The first line contains 2 integers N and M. N is the number of nodes. M is the number of undirected edges. Then each entry contains the link information (M entries). Line i+1 contains 3 integers ui, vi and wi which represents an undirected edge between nodes ui and vi and wi is the weight of that edge.

Constraints:

1<= N <= 100

All edge weights are positive.

Output Format: Print the final dist matrix. In case node j is not reachable from node i, then print INF as the distance between i and j.

Example:

Input:	**Output:**
4 4 1 2 1 2 3 1 3 4 1 4 1 1	0 1 2 1 1 0 1 2 2 1 0 1 1 2 1 0