Concurrent and Distributed Systems
Concurrent Problem Solving

Mitre Flavia Antonia

January 18, 2020

Third Year
Group: C.E.N. 3.1
Lecturer: Costin Bădică
Teaching assistant: Cristinel Ungureanu

1 Problem statement

1.1 First problem

Implement a program to solve the producer-consumer problem using:
- coarse synchronization
- fine synchronization

The code must be tested with at least 4 producers and the number of consumers will be given by the number of available virtual CPU's.

2 Implementation

2.1 First problem

For solving the first problem with fine synchronization I've created the classes:

- **QueueMain**

  This class contains the main of the program.

  Here new PCQueue, producer and consumer instances are created.
  The number of producers and consumers equals the number of available processors.
  The threads for producer and consumer are started and joined.

- **Consumer**

  This class contains two private variables: a PCQueue and an integer number.
  The consumer constructor has an int and a PCQueue as parameters.
  This class extends Thread.
  In the implementation of the run method, for every available processor, an element is consumed.

- **Producer**

  This class contains a private variable PCQueue.
  The producer constructor has an int and a PCQueue as parameters.
  This class extends Thread.
  In the implementation of the run method, for every available processor, an element is produced.

- **PCQueue**

  This class contains 2 private Semaphores: one for producer and one for consumer and two locks (only one lock makes the program go in a deadlock).
  The semaphore for consumer has 0 permits, to assure that the consumer does not try to consume an item before it is produced.
  The method put is synchronised with the first lock, the producer semaphore is acquired, and the consumer semaphore is released.
  The method get is synchronised with second lock, the consumer semaphore

is acquired, and the producer semaphore is released.

For solving the first problem with coarse synchronization I've created the classes:

- **QueueMain**

  This class contains the main of the program.

  Here new PCQueue, producer and consumer instances are created.
  The number of producers and consumers equals the number of available processors.
  The threads for producer and consumer are started and joined.

- **Consumer**

  This class contains two private variables: a PCQueue and an integer number.
  The consumer constructor has an int and a PCQueue as parameters.
  This class extends Thread.
  In the implementation of the run method, for every available processor, an element is consumed.

- **Producer**

  This class contains a private variable PCQueue.
  The producer constructor has an int and a PCQueue as parameters.
  This class extends Thread.
  In the implementation of the run method, for every available processor, an element is produced.

- **PCQueue**

  This class contains 2 private Semaphores: one for producer and one for consumer a boolean flag.
  The semaphore for consumer has 0 permits, to assure that the consumer does not try to consume an item before it is produced.
  The method put is synchronised. The producer semaphore is acquired, the flag becomes false, and it notifies all the threads about it.
  The consumer semaphore is released.
  This method waits for the flag to be true to end it's execution.
  The method get is synchronised. The consumer semaphore is acquired,

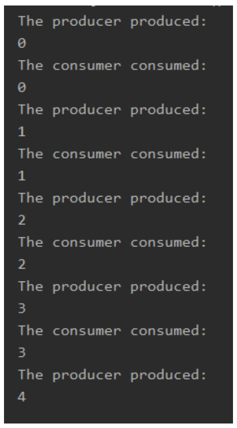
the flag becomes true, and it notifies all the threads about it.
The producer semaphore is released.
This method waits for the flag to be false to end it's execution.

Pseudocode

```
1. item = 0
2. finite queue of dataType buffer ← empty queue
Producer:
3. dataType d
4. loop forever
5. queue(notFull)
6. add(item)
7. notifyAll()
8. item ← item + 1
9. end loop
Consumer:
10. dataType d
11. loop forever
12. queue(notEmpty)
13. d ← take(buffer)
14. remove(d)
15. wait()
16. end loop
```

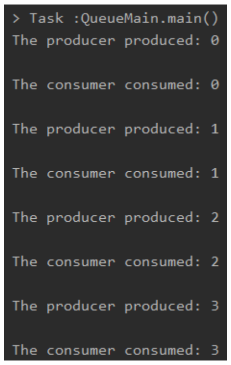
3 Experimental Data

3.1 First problem



```
The producer produced:
0
The consumer consumed:
0
The producer produced:
1
The consumer consumed:
1
The producer produced:
2
The consumer consumed:
2
The producer produced:
3
The consumer consumed:
3
The producer produced:
4
```

Figure 1: Output for fine synchronisation



```
> Task :QueueMain.main()
The producer produced: 0

The consumer consumed: 0

The producer produced: 1

The consumer consumed: 1

The producer produced: 2

The consumer consumed: 2

The producer produced: 3

The consumer consumed: 3
```

Figure 2: Output for coarse synchronisation

4 Conclusions

For fine synchronisation:
The producer–consumer problem is a classic example of a multi-process synchronization problem.
The problem is to make sure that the producer won't try to add data into the queue if it's full and that the consumer won't try to remove data from an empty queue.
A bad solution could result in a deadlock where both processes are waiting to be awakened.
The semaphores used in the solution of the problem assure executing the ”put” operation first, using a semaphore with 0 permits. The producer will produce an item, and right after the consumer will consume it.
The lock locks the production of items until the queue is full, and then consumes the all items from the queue.
For coarse synchronisation:
This solution is based on using a flag instead of locks, that becomes true when a producer produces and false when a consumer consumes.