Concurrent and Distributed Systems
Synchronization Mechanisms

Mitre Flavia Antonia

December 12, 2019

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

1 Problem statement

1.1 First problem

Implement the producer consumer problem as follows:
a) using semaphores
b) using monitors
c) using locks

2 Implementation

2.1 First problem

2.1.1 Semaphores

For solving the first problem with **semaphores** I've created the classes:

- **Producer**

  This class extends the **Thread** class (overrides the **run** method from threads).
  It contains a private global variable of type **SemaphoreBasedQueue**.
  The **constructor** for this class receives a **SemaphoreBasedQueue** as a parameter and stores it in the global variable.
  In the run method with a for (0->5) items are put in the queue. The **Producer** produces items.

- **Consumer**

  This class extends the **Thread** class (overrides the **run** method from threads).
  It contains a private global variable of type **SemaphoreBasedQueue**.
  The **constructor** for this class receives a **SemaphoreBasedQueue** as a parameter and stores it in the global variable.
  In the run method with a for (0->5) items are gotten from the queue. The **Consumer** consumes items.

- **SemaphoreBasedQueue**

  This class contains an **item**, and two **semaphores**: one for the **Producer** (with one permit) and one for the **Consumer** (with 0 permits).
  We want the **put** action to be executed **first**. Before getting an item, it should be produced.

  The method **get** is implemented here.
  Try to **acquire** the **Consumer Semaphore**, necessary to consume an item.
  Display which **Consumer** consumes which item.
  Release the **Producer Semaphore** (the consuming is done, so the **Producer** must produce again).

  The method **put** is implemented here.
  It has an **item** as parameter.
  Try to **acquire** the **Producer Semaphore**, necessary to produce an

item.
Display which **Producer** produces which item.
Release the **Consumer Semaphore** (the producing is done, so the **Consumer** must consume again).

This class contains the **main** of the program.
A new **SemaphoreBasedQueue** is created and, with it's help, two **Producers**, and two **Consumers**.
All the **Producers** and **Consumers start** and try to **join**.

Pseudocode

- **Semaphores:**

1. Semaphore notEmpty ← (0,∅)
2. Semaphore notFull ← (N,∅)
3. finite queue of dataType buffer ← empty queue

Producer:
4. dataType d
5. **loop forever**
6. d ← produce
7. wait(notFull)
8. append(d,buffer)
9. signal(notEmpty)
10. **end loop**

Consumer:
11. dataType d
12. **loop forever**
13. wait(notEmpty)
14. d ← take(buffer)
15. signal(notFull)
16. consume(d)
17. **end loop**

2.1.2 Monitors

For solving the first problem with **monitors** I've created the classes:

- **Producer**

This class extends the **Thread** class (overrides the **run** method from threads).
It contains a private global variable of type **List** that plays the role of the queue.
The item that will be **added** to the queue is initialised with 0.

The **constructor** for this class receives a **List** of type Integer as a parameter and stores it in the global variable.
In the run method with a for (0 ->5) items are **added** to the queue.
The queue is **synchronized**.
The **Producer** produces items.
When an item was produced, the **Consumer** is **notified** to start consuming again.
The item is incremented.
The thread sleeps for 1000 milliseconds.

- **Consumer**

  This class extends the **Thread** class (overrides the **run** method from threads).
  It contains a private global variable of type **List** that plays the role of the queue.
  The **constructor** for this class receives a **List** of type Integer as a parameter and stores it in the global variable.
  In the run method with a for (0 ->5) items are removed from the queue.
  The queue is **synchronized**.
  If the queue is not emtpy, the **Consumer** consumes items.
  Else, the **Consumer waits** for the queue to not be empty anymore.
  The thread sleeps for 1000 milliseconds.

- **MonitorBasedQueue**

  This class contains the **main** of the program.
  Also, a variable of type **List** that plays the role of the queue.
  Four new **Threads** are created: two **Producers** and two **Consumers**.

  All the **Producers** and **Consumers start** and try to **join**.

Pseudocode

- **Monitors:**

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**

2.1.3 Locks

For solving the first problem with **locks** I've created the classes:

- **Producer**

  This class extends the **Thread** class (overrides the **run** method from threads).
  It contains a private global variable of type **LockBasedQueue**.
  The **constructor** for this class receives a **LockBasedQueue** of type Integer as a parameter and stores it in the global variable.
  In the run method with a for (0 ->5) items are enqueued. The **Producer** produces items.

- **Consumer**

  This class extends the **Thread** class (overrides the **run** method from threads).
  It contains a private global variable of type **LockBasedQueue**.
  The **constructor** for this class receives a **LockBasedQueue** as a parameter and stores it in the global variable.
  In the run method with a for (0 ->5) items are dequeued. The **Consumer** consumes items.

- **LockBasedQueue**

  This class contains: an **item** array of type T, the **head** and the **tail** of the queue, and a **lock**.
  The **constructor** for this class receives a **capacity** and creates as many items of type T as this capacity.
  The queue is initialised (**head** is empty, **tail** is empty), also the lock is initialised as a new **reentrant lock**.

  The **enque** method (enq) is implemented here.
  It has an **item** of type T as parameter.
  The **lock** is locked. Try to **put** an **item** in the queue (produce an item). Increment the tail.
  Finally, display which **Producer** produces which item.
  Unlock the **lock** (the producing is done, so the **Consumer** must consume again).

  The **deque** method (deq) is implemented here.
  The **lock** is locked. Try to **get** an **item** from the queue (consume an item). Increment the head.

Finally, display which **Consumer** consumes which item.
Unlock the **lock** (the consuming is done, so the **Producer** must produce again).

This class contains the **main** of the program.
A new **LockBasedQueue** (type Integer, capacity 6) is created, and with it's help, two **Producers**, and two **Consumers**.
All the **Producers** and **Consumers start** and try to **join**.

Pseudocode

- **Locks:**

1. Lock lock
2. finite queue of dataType buffer ← empty queue

Producer:

3. dataType d
4. **loop forever**
5. d ← produce
6. lock(notFull)
7. append(d,buffer)
8. unlock(notEmpty)
9. **end loop**

Consumer:

10. dataType d
11. **loop forever**
12. lock(notEmpty)
13. d ← take(buffer)
14. unlock(notFull)
15. consume(d)
16. **end loop**

3 Experimental Data

3.1 First problem

Semaphores:

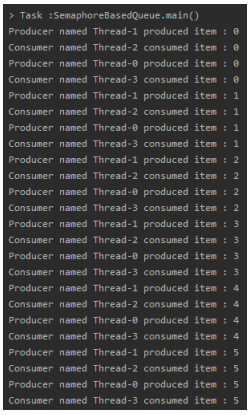

Figure 1: First output

Producer 1 produces. Consumer 2 consumes. Producer 0 produces. Consumer 3 consumes. This happens for all the values in the interval given, in the same order(0, 1 ... 5). The producing action happens first and the consuming, last.

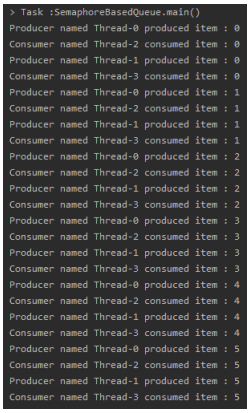

Figure 2: Second output

Producer 0 produces. Consumer 2 consumes. Producer 1 produces. Consumer 3 consumes. This happens for all the values in the interval given, in the same order(0, 1 ... 5). The only thing that can change is the order for the producers(0 can be first, or 1 can be first) or consumers(2 can be first, or 3 can be first). The producing action happens first and the consuming, last. 0, 1, 2, 3 are the numbers of the threads used.

Monitors:

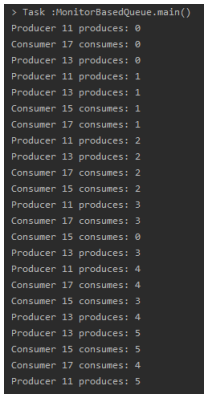

Figure 3: First output
Producer 1 produces 0, 1 ... 5.
Producer 0 produces 0, 1 ... 5.
Consumer 2 consumes 0, 1 ... 5.
Consumer 3 consumes 0, 1 ... 5.
The threads don't make actions in order, but the Consumer never tries to consume a value before it was produced. The value 0 produced by one of the producers is consumed later. The last produced value is never consumed.

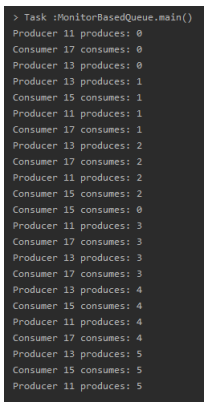

Figure 4: Second output
Producer 0 produces 0, 1 ... 5.
Producer 1 produces 0, 1 ... 5.
Consumer 3 consumes 0, 1 ... 5.
Consumer 2 consumes 0, 1 ... 5.
The value 0 produced by one of the producers is consumed later. The last produced value is never consumed.

Locks:

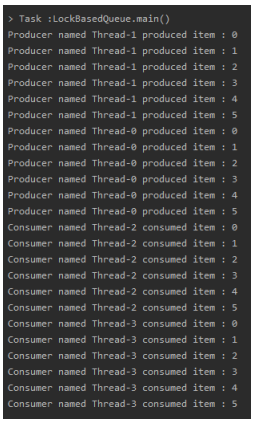

Figure 5: First output
Producer 1 produces 0, 1 ... 5.
Producer 0 produces 0, 1 ... 5.
Consumer 2 consumes 0, 1 ... 5.
Consumer 3 consumes 0, 1 ... 5.
The only thing that can change is the order for the producers (0 can be first, or 1 can be first) or consumers (2 can be first, or 3 can be first). The producing action happens first and the consuming, last (possible cases: P, P, C, C or P, C, P, C). 0, 1, 2, 3 are the numbers of the threads used.

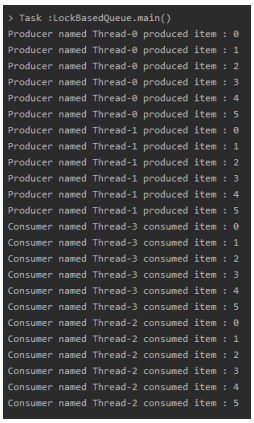

Figure 6: Second output
Producer 0 produces 0, 1 ... 5.
Producer 1 produces 0, 1 ... 5.
Consumer 3 consumes 0, 1 ... 5.
Consumer 2 consumes 0, 1 ... 5.

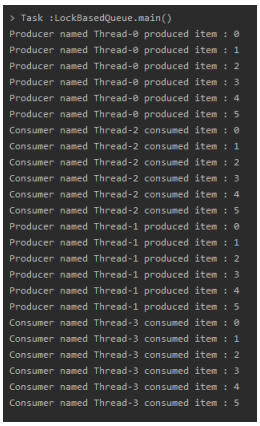

Figure 7: Third output
Producer 0 produces 0, 1 ... 5.
Consumer 2 consumes 0, 1 ... 5.
Producer 1 produces 0, 1 ... 5.
Consumer 3 consumes 0, 1 ... 5.

4 Conclusions

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.
The solution with monitors is that the producer will go to sleep or will discard data if the queue is full. The next time the consumer removes an item from the queue, it notifies the producer, who starts to fill the queue again. In the same way, the consumer can go to sleep if it finds the queue to be empty. The next time the producer puts data into the queue, it wakes up the sleeping consumer. An bad solution could result in a deadlock where both processes are waiting to be awakened. The difference for the semaphore solution of the problem is 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 solution basically locks the production of items until the queue is full, and then consumes the all items from the queue.