

## ... continued

This lesson explains how to solve the producer-consumer problem using a mutex.

In the previous lesson, we solved the consumer producer problem using the `synchronized` keyword, which is equivalent of a monitor in Java. Let's see how the implementation would look like, if we were restricted to using a mutex. There's no direct equivalent of a theoretical mutex in Java as each object has an implicit monitor associated with it. For this question, we'll use an object of the `Lock` class and pretend it doesn't expose the `wait()` and `notify()` methods and only provides mutual exclusion similar to a theoretical mutex. Without the ability to wait or signal the implication is, a blocked thread will constantly poll in a loop for a predicate/condition to become true before making progress. This is an example of a busy-wait solution.

Let's start with the `enqueue()` method. If the current `size of the queue == capacity` then we know we need to block the caller of the method until the queue has space for a new item. Since a mutex only allows locking, we give up the mutex at this point. The logic is shown below.

```
lock.lock();
while (size == capacity) {
    // Release the mutex to give other threads
    lock.unlock();
    // Reacquire the mutex before checking the
    // condition
    lock.lock();
}

if (tail == capacity) {
    tail = 0;
}

array[tail] = item;
```

```

size++;

tail++;
lock.unlock();

```

The most important point to realize in the above code is the weird-looking while loop construct, where we release the lock and then immediately attempt to reacquire it. Convince yourself that whenever we test the while loop condition `size == capacity`, we do so while holding the mutex! Also, it may not be immediately obvious but a different thread can acquire the mutex just when a thread releases the mutex and attempts to reacquire it within the while loop. Lastly, we modify the `array` variable only when holding the mutex.

We also need to manage the `tail` as the queue grows. Once it reaches the end of our backing array, we reset it to zero. Realize that since we only proceed to add an item when `size of queue < maxSize` we are guaranteed that `tail` will never overwrite an existing item.

Now let us see the code for the `dequeue()` method which is analogous to the `enqueue()` one.

```

T item = null;

lock.lock();
while (size == 0) {
    lock.unlock();
    lock.lock();
}

if (head == capacity) {
    head = 0;
}

item = array[head];
array[head] = null;
head++;
size--;

lock.unlock();
return item;

```

Again note that we always test for the condition `size == 0` when holding

Again note that we always test for the condition `size == 0` when holding the lock. Additionally, all shared state is manipulated in mutual exclusion.

Additionally, we reset `head` of the queue back to zero in case it's pointing past the end of the array. We need to decrement the `size` variable too since the queue will now have one less item. The complete code appears in the widget below. It also runs a simulation of several producers and consumers that constantly write and retrieve from an instance of the blocking queue, for one second.

main.java

BlockingQueueWithMutex.java



```
class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        final BlockingQueueWithMutex<Integer> q = new BlockingQueueWithMutex<Integer>(5);

        Thread producer1 = new Thread(new Runnable() {
            public void run() {
                try {
                    int i = 1;
                    while (true) {
                        q.enqueue(i);
                        System.out.println("Producer thread 1 enqueued " + i);
                        i++;
                    }
                } catch (InterruptedException ie) {
                }
            }
        });

        Thread producer2 = new Thread(new Runnable() {
            public void run() {
                try {
                    int i = 5000;
                    while (true) {
                        q.enqueue(i);
                        System.out.println("Producer thread 2 enqueued " + i);
                        i++;
                    }
                } catch (InterruptedException ie) {
                }
            }
        });

        Thread producer3 = new Thread(new Runnable() {
            public void run() {
                try {
                    int i = 100000;
                    while (true) {
                        q.enqueue(i);
```

```

        System.out.println("Producer thread 3 enqueued " + i);
        i++;
    }
} catch (InterruptedException ie) {
}
}
});

Thread consumer1 = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                System.out.println("Consumer thread 1 dequeued " + q.dequeue());
            }
        } catch (InterruptedException ie) {
        }
    }
});

Thread consumer2 = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                System.out.println("Consumer thread 2 dequeued " + q.dequeue());
            }
        } catch (InterruptedException ie) {
        }
    }
});

Thread consumer3 = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                System.out.println("Consumer thread 3 dequeued " + q.dequeue());
            }
        } catch (InterruptedException ie) {
        }
    }
});

producer1.setDaemon(true);
producer2.setDaemon(true);
producer3.setDaemon(true);
consumer1.setDaemon(true);
consumer2.setDaemon(true);
consumer3.setDaemon(true);

producer1.start();
producer2.start();
producer3.start();

consumer1.start();
consumer2.start();
consumer3.start();

Thread.sleep(1000);
}

```



## Faulty Implementation

As an exercise, we reproduce the two `enqueue()` and `dequeue()` methods, without locking the mutex object when checking for the while-loop conditions. If you run the code in the widget below multiple times, some of the runs would display a dequeue value of null. We set an array index to null whenever we remove its content to indicate the index is now empty. A race condition is introduced when we check for while-loop predicate without holding a mutex.

Incorrect dequeue() implementation

```
public T dequeue() {  
  
    T item = null;  
  
    while (size == 0) { }  
  
    lock.lock();  
    if (head == capacity) {  
        head = 0;  
    }  
  
    item = array[head];  
    array[head] = null;  
    head++;  
    size--;  
  
    lock.unlock();  
    return item;  
}
```

and,

Incorrect enqueue() implementation

```
public void enqueue(T item) {  
  
    while (size == capacity) { }
```

```

        lock.lock();

        if (tail == capacity) {
            tail = 0;
        }

        array[tail] = item;
        size++;
        tail++;
        lock.unlock();
    }

```

main.java



FaultyBlockingQueueWithMute<

```

class Demonstration {

    static final FaultyBlockingQueueWithMutex<Integer> q = new FaultyBlockingQueueWithMutex<

    static void producerThread(int start, int id ) {
        while (true) {
            try {
                q.enqueue(start);
                System.out.println("Producer thread " + id + " enqueued " + start);
                start++;
                Thread.sleep(1);
            } catch (InterruptedException ie){
                // swallow exception
            }
        }
    }

    static void consumerThread(int id) {
        while (true) {
            try {
                System.out.println("Consumer thread " + id + " dequeued " + q.dequeue());
                Thread.sleep(1);
            } catch (InterruptedException ie){
                // swallow exception
            }
        }
    }

    public static void main( String args[] ) throws InterruptedException {

        Thread producer1 = new Thread(new Runnable() {
            public void run() {
                producerThread(1, 1);
            }
        });
    }

```

```
Thread producer2 = new Thread(new Runnable() {
    public void run() {
        producerThread(5000, 2);
    }
});

Thread producer3 = new Thread(new Runnable() {
    public void run() {
        producerThread(100000, 3);
    }
});

Thread consumer1 = new Thread(new Runnable() {
    public void run() {
        consumerThread(1);
    }
});

Thread consumer2 = new Thread(new Runnable() {
    public void run() {
        consumerThread(2);
    }
});

Thread consumer3 = new Thread(new Runnable() {
    public void run() {
        consumerThread(3);
    }
});

producer1.setDaemon(true);
producer2.setDaemon(true);
producer3.setDaemon(true);
consumer1.setDaemon(true);
consumer2.setDaemon(true);
consumer3.setDaemon(true);

producer1.start();
producer2.start();
producer3.start();

consumer1.start();
consumer2.start();
consumer3.start();

Thread.sleep(20000);
}
```

