... continued

This lesson explains implementing a bounded buffer using Semaphores.

Semaphore Implementation

We can also implement the bounded buffer problem using a semaphore. Since Ruby's core library doesn't have a semaphore, we'll use the semaphore implementation from the previous lesson. Let's revisit our CountingSemaphore's constructor. It takes in the maximum number of permits. We can use two semaphores, one semConsumer and the other semProducer. The trick is to initialize semProducer semaphore with the maximum queue capacity of N. This allows producer threads to enqueue the However, the items in queue. semProducere only released/incremented by a consumer thread whenever it consumes an item. If there are no consumer threads, the producer threads will block when the buffer size becomes N. For the consumer thread code, initially, the buffer is empty. We would want to block any consumer threads on a consumer() method call. This implies that we should initialize the semConsumer with zero permits and a maximum size of N, since the producers can't put more than N items in the buffer. Given our implementation of the CountingSemaphore we can do that by simply acquiring all the consumer permits in the initialize() method of the BlockingQueue class. Let's look at the implementation of producer() method.

```
def producer(elem)

@semProducer.acquire()

# Adds an element to the end of the queue
@queue << elem
puts "Added #{elem} to the queue"</pre>
```

end

If you study the code above, it should be evident that only N items can be enqueued in the items buffer. At the end of the method, we signal any consumer threads waiting on the <code>semConsumer</code> semaphore. However, the code is not yet complete. We have only solved the problem of coordinating between the threads. The astute reader would immediately realize that multiple producer threads can manipulate the code lines between the first and the last semaphore statements in the above <code>producer()</code> method. In our earlier implementations, we were able to guard the critical section by locking a mutex or entering a monitor that ensured only a single thread is active in the critical section at a time. We need similar functionality using semaphores. Recall that we can use a binary semaphore in place of a mutex, however, any thread is free to signal the semaphore, not just the one that acquired it. We'll introduce a <code>semLock</code> semaphore that acts as a mutex. The complete version of the enqueue() method appears below:

```
def producer(elem)

  @semProducer.acquire()

  @semLock.acquire()

  # Adds an element to the end of the queue
  @queue << elem
  puts "Added #{elem} to the queue"
  @semLock.release()

  @semConsumer.release()
end</pre>
```

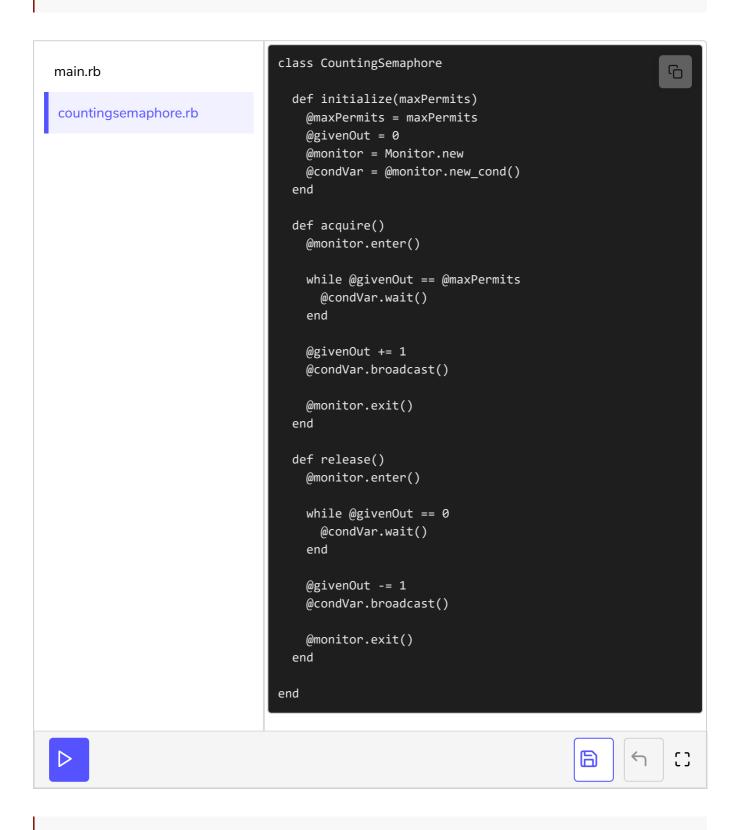
Realize that we have modeled each item in the buffer as a permit. When the buffer is full, the consumer threads have N permits to invoke consumer() method and when the buffer is empty the producer threads have N permits to invoke producer() method. The code for consumer() is similar and appears below:

```
def consumer
  @semConsumer.acquire()
  @semLock.acquire()

# Potnioves the element at the first index
```

```
puts "Consumed #{@queue.shift} from the queue"
  @semLock.release()
  @semProducer.release()
end
```

The complete code appears below.



In the test case above, we intentionally keep the size of the blocking queue to one. There are two consumer threads and five producer threads.

A total of fifteen items got anguoued. Since the gueue canacity is one, the

A total of inteen items get enqueued. Since the queue capacity is one, the

log lines from the producer and consumer threads must alternate, as shown in the output from running the above code widget.