... continued

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

Using Semaphores for Producer-Consumer Problem

We can also implement the bounded buffer problem using a semaphore. For this problem, we'll use an instance of the `CountingSemaphore` that we implement in one of the later problems. A `CountingSemaphore` is initialized with a maximum number of permits to give out. A thread is blocked when it attempts to release the semaphore when none of the permits have been given out. Similarly, a thread blocks when attempting to acquire a semaphore that has all the permits given out. In contrast, Java's implementation of Semaphore can be signaled (released) even if none of the permits, the Java semaphore was initialized with, have been used. Java's semaphore has no upper bound and can be released as many times as desired to increase the number of permits. Before proceeding forward, it is suggested to complete the `CountingSemaphore` lesson.

We'll augment the `CountingSemaphore` class with a new constructor that takes in the maximum permits and also sets the number of permits already given out. We can use two semaphores, one `semConsumer` and the other `semProducer`. The trick is to initialize `semProducer` semaphore with a maximum number of permits equal to the size of the buffer and set all permits as available. Each permit allows a producer thread to enqueue an item in the buffer. Since the number of permits is equal to the size of the buffer, the producer threads can only enqueue items equal to the size of the buffer and then blocks. However, the `semProducer` is 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 becomes full. In case of the consumer threads, when the buffer is empty, we would want to block any consumer threads on a `dequeue()` call. This implies that we should initialize the `semConsumer`

semaphore with a maximum capacity equal to the size of the buffer and

set all the permits as currently given out. Let's look at the implementation of `enqueue()` method.

```
    public void enqueue(T item) throws InterruptedException {

        semProducer.acquire();

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

        array[tail] = item;
        size++;
        tail++;

        semConsumer.release();
    }
```

Suppose the size of the buffer is N. 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 `semConsumer` semaphore. However, the code is not yet complete. We have only solved the problem of coordinating between the producer and the consumer 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 `enqueue()` method. In our earlier implementations, we were able to guard the critical section by synchronizing on objects 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 to exercise mutual exclusion, however, any thread is free to signal the semaphore, not just the one that acquired it. We'll introduce a `semLock` semaphore that acts as a mutex. The complete version of the `enqueue()` method appears below:

```
    public void enqueue(T item) throws InterruptedException {

        semProducer.acquire();
        semLock.acquire();
```

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

        array[tail] = item;
        size++;
        tail++;

        semLock.release();
        semConsumer.release();
    }
```

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 perform `dequeue()` and when the buffer is empty the producer threads have N permits to perform `enqueue()`. The code for `dequeue()` is similar and appears below:

```
    public T dequeue() throws InterruptedException {

        T item = null;

        semConsumer.acquire();
        semLock.acquire();

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

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

        semLock.release();
        semProducer.release();

        return item;
    }
```

The complete code appears in the code widget below. We also include a simple test with one producer and two consumer threads.

main.java

BlockingQueueWithSemaphore

CountingSemaphore.java

```
public class BlockingQueueWithSemaphore<T> {
    T[] array;
    int size = 0;
    int capacity;
    int head = 0;
    int tail = 0;
    CountingSemaphore semLock = new CountingSemaphore(1, 1);
    CountingSemaphore semProducer;
    CountingSemaphore semConsumer;

    @SuppressWarnings("unchecked")
    public BlockingQueueWithSemaphore(int capacity) {
        // The casting results in a warning
        array = (T[]) new Object[capacity];
        this.capacity = capacity;
        this.semProducer = new CountingSemaphore(capacity, capacity);
        this.semConsumer = new CountingSemaphore(capacity, 0);
    }

    public T dequeue() throws InterruptedException {

        T item = null;

        semConsumer.acquire();
        semLock.acquire();

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

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

        semLock.release();
        semProducer.release();

        return item;
    }

    public void enqueue(T item) throws InterruptedException {

        semProducer.acquire();
        semLock.acquire();

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

```
        array[tail] = item;
        size++;
        tail++;

        semLock.release();
        semConsumer.release();
    }
}
```