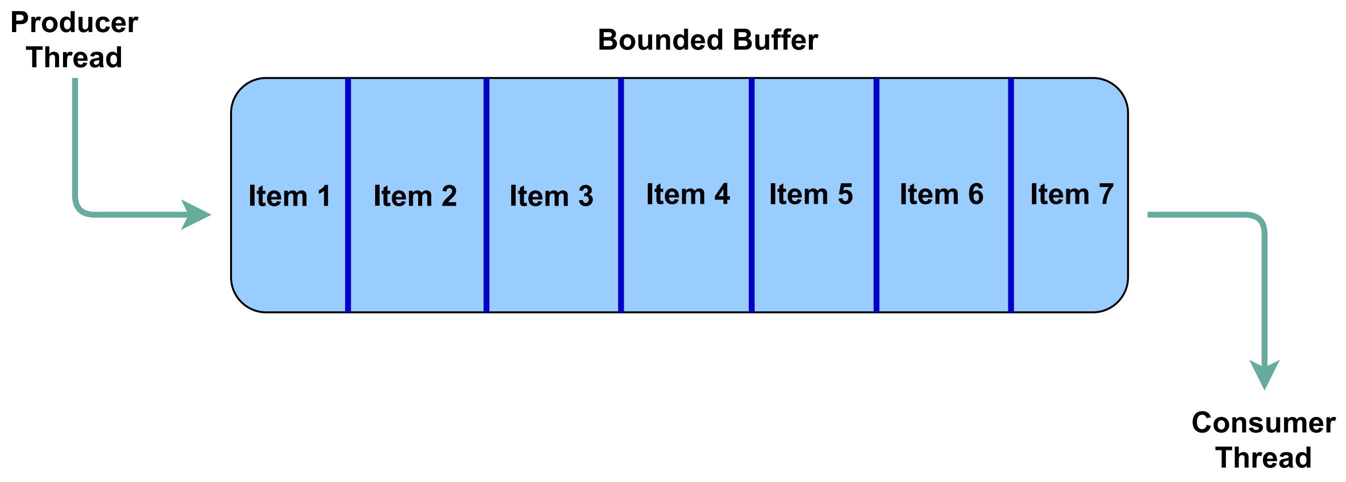
Blocking Queue | Bounded Buffer | Consumer Producer

Classical synchronization problem involving a limited size buffer which can have items added to it or removed from it by different producer and consumer threads. This problem is known by different names: consumer producer problem, bounded buffer problem or blocking queue problem.

Problem

A blocking queue is defined as a queue which blocks the caller of the enqueue method if there's no more capacity to add the new item being enqueued. Similarly, the queue blocks the dequeue caller if there are no items in the queue. Also, the queue notifies a blocked enqueuing thread when space becomes available and a blocked dequeuing thread when an item becomes available in the queue.



Bounder buffer in action

Solution

Our queue will have a finite size that is passed in via the constructor. Additionally, we'll use an array as the data structure for backing our queue. Furthermore, we'll expose the APIs `enqueue` and `dequeue` for our blocking queue class. We'll also need a **head** and a **tail** pointer to keep blocking queue class. We'll also need a **head** and a **tail** pointer to keep track of the front and back of the queue and a size variable to keep track

of the queue size at any given point in time. Given this, the skeleton of our blocking queue class would look something like below:

```
public class BlockingQueue<T> {

    T[] array;
    int size = 0;
    int capacity;
    int head = 0;
    int tail = 0;

    public BlockingQueue(int capacity) {
        array = (T[]) new Object[capacity];
        this.capacity = capacity;
    }

    public void enqueue(T item) {
    }

    public T dequeue() {
    }
}
```

Let's start with the **enqueue** method. If the current `size of the queue == capacity` then we know we'll need to block the caller of the method. We can do so by appropriately calling **wait()** method in a while loop. The while loop is conditioned on the size of the queue being equal to the max capacity. The loop's predicate would become false, as soon as, another thread performs a dequeue.

Note that whenever we test for the value of the **size** variable, we also need to make sure that no other thread is manipulating the size variable. This can be achieved by the synchronized keyword as it'll only allow a single thread to invoke the enqueue/dequeue methods on the queue object.

Finally, as the queue grows, it'll reach the end of our backing array, so we need to reset the tail of the queue back to zero. Notice that since we only

proceed to enqueue an item when `size of queue < capacity` we are guaranteed that tail would not be overwriting an existing item.

```
    public synchronized void enqueue(T item) throws InterruptedExcept
ion {

        // wait for queue to have space
        while (size == capacity) {
            wait();
        }

        // reset tail to the beginning if the tail is already
        // at the end of the backing array
        if (tail == capacity) {
            tail = 0;
        }

        // place the item in the array
        array[tail] = item;
        size++;
        tail++;

        // don't forget to notify any other threads waiting on
        // a change in value of size. There might be consumers
        // waiting for the queue to have atleast one element
        notifyAll();
    }
```

Note that in the end we are calling `notifyAll()` method. Since we just added an item to the queue, it is possible that a consumer thread is blocked in the dequeue method of the queue class waiting for an item to become available so it's necessary we send a signal to wake up any waiting threads.

If no thread is waiting, then the signal will simply go unnoticed and be ignored, which wouldn't affect the correct working of our class. This would be an instance of **missed signal** that we have talked about earlier.

Now let's design the `dequeue` method. Similar to the enqueue method, we need to block the caller of the dequeue method if there's nothing to dequeue i.e. `size == 0`

We need to 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.

Finally, we remember to call `notifyAll()` since if the queue were full then there might be producer threads blocked in the enqueue method. This logic in code appears as below:

```java
    public synchronized T dequeue() throws InterruptedException {

        T item = null;

        // wait for atleast one item to be enqueued
        while (size == 0) {
            wait();
        }

        // reset head to start of array if its past the array
        if (head == capacity) {
            head = 0;
        }

        // store the reference to the object being dequeued
        // and overwrite with null
        item = array[head];
        array[head] = null;
        head++;
        size--;

        // don't forget to call notify, there might be another thread
        // blocked in the enqueue method.
        notifyAll();

        return item;
    }
```

We see the dequeue method is analogous to enqueue method. Note that we could have eliminated lines 17 & 18 and instead just returned the following:

`return array[head-1];`

but for better readability we choose to expand this operation into two lines.

Complete Code

The full code for the blocking queue appears below.

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

        Thread t1 = new Thread(new Runnable() {

            @Override
            public void run() {
                try {
                    for (int i = 0; i < 50; i++) {
                        q.enqueue(new Integer(i));
                        System.out.println("enqueued " + i);
                    }
                } catch (InterruptedException ie) {

                }
            }
        });

        Thread t2 = new Thread(new Runnable() {

            @Override
            public void run() {
                try {
                    for (int i = 0; i < 25; i++) {
                        System.out.println("Thread 2 dequeued: " + q.dequeue());
                    }
                } catch (InterruptedException ie) {

                }
            }
        });

        Thread t3 = new Thread(new Runnable() {

            @Override
            public void run() {
                try {
                    for (int i = 0; i < 25; i++) {
                        System.out.println("Thread 3 dequeued: " + q.dequeue());
                    }
                } catch (InterruptedException ie) {

                }
            }
        });

        t1.start();
        Thread.sleep(4000);
        t2.start();

        t2.join();

        t3.start();
        t1.join();
        t3.join();
    }
}

// The blocking queue class
class BlockingQueue<T> {

    T[] array;
    Object lock = new Object();
    int size = 0;
    int capacity;
    int head = 0;
    int tail = 0;

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

    public void enqueue(T item) throws InterruptedException {

        synchronized (lock) {

            while (size == capacity) {
                lock.wait();
            }

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

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

    public T dequeue() throws InterruptedException {

        T item = null;
        synchronized (lock) {

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

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

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

            lock.notifyAll();
        }

        return item;
    }
}
```

The test case in our example creates two dequeuer threads and one enqueuer thread. The enqueue-er thread initially fills up the queue and gets blocked, till the dequeuer threads start off and remove elements from the queue. The output would show enqueuing and dequeuing activity interleaved after the first 5 enqueues.

Follow Up Question

In both the `enqueue()` and `dequeue()` methods we use the `notifyAll()` method instead of the `notify()` method. The reason behind the choice is very crucial to understand. Consider a situation with two producer threads and one consumer thread all working with a queue of size one. It's possible that when an item is added to the queue by one of the producer threads, the other two threads are blocked waiting on the condition variable. If the producer thread after adding an item invokes `notify()` it is possible that the other producer thread is chosen by the system to resume execution. The woken-up producer thread would find the queue full and go back to waiting on the condition variable, causing a deadlock. Invoking `notifyAll()` assures that the consumer thread also gets a chance to wake up and resume execution.