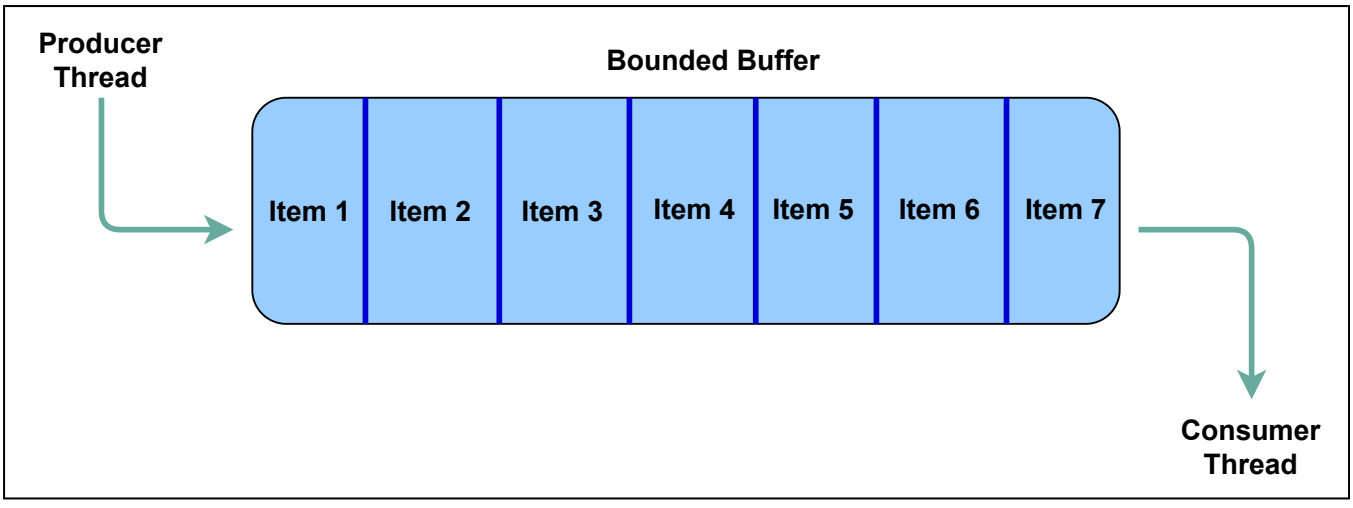
Blocking Queue | Bounded Buffer | Consumer Producer

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: blocking queue problem, bounded buffer problem or consumer producer problem.

Blocking Queue | Bounded Buffer | Consumer Producer

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.



Solution

The bounded buffer is a classic textbook concurrency problem. We'll look at the various ways we can implement such a buffer.

Our queue will have a finite size that is passed in via the constructor. Additionally, we'll use an integer array as the data structure for backing our queue. We choose to use a simple integer array to make the example easier to follow. However, a solution with a generic container would follow the same underlying logic.

Furthermore, we'll expose two APIs in our blocking queue class:

- `producer()`
- `consumer()`

We'll always add an element to the end of the array and remove an element from the first index of the array. Therefore, the end of the array is the tail of the queue and the start of the array is the head of the queue.

```
class BlockingQueue

  # this method declares a simple array as the data structure for blo
cking queue,
  # mutex for synchronization and a condition variable for waiting an
d notifying
  def initialize(capacity)
    @queue_capacity = capacity
    @queue = Array.new()
    @mutex = Mutex.new
    @cond_var = ConditionVariable.new
  end

  # producer method that populates the queue and notifies the waitin
g threads
  def producer(elem)
  end

  # consumer method that either consumes items from the queue or wait
s until something is present
  def consumer
  end

end
```

Let's start with the `producer()` method that adds items into the queue.

When should we block the producer? When the queue is at its maximum capacity. We can make the producer thread `wait()` on the condition variable. However, remember, whenever we wait on a condition variable we must wrap the `wait()` call in a while loop that tests for the predicate. This is to take care of spurious wakeups and ensure correct working in case of more than one producer threads. Moreover, since we check for the predicate in a while loop and also add items to the queue, both of which also get manipulated in the consumer code, we wrap the entire function body in a mutex lock. Lastly, whenever we add an item to the queue there may be consumer threads waiting on the condition variable, in case the queue was empty, so we must not forget to `signal()` the condition variable to wakeup any such threads. The code for the method appears below:

```
def producer(elem)
  @mutex.synchronize do
    while @queue.size == @queue_capacity
      puts "Oh no! The queue is full,   Please wait.."
      @cond_var.wait(@mutex)
    end
     # Adds an element to the end of the queue
    @queue << elem
    puts "Added #{elem} to the queue"
    @cond_var.signal
  end
end
```

Let's consider the `consumer()` method now. The logic is complementary to the `producer()` method. When should we block the consumer? When the queue is empty. That is the predicate we should check for in the while loop. Similarly, we should always `signal()` the condition variable whenever an item is consumed from the queue because there might be a producer thread that is waiting on the condition variable in case the queue was full. Finally, we wrap the function body in the same mutex as the other method since we mutate shared data-structures. The code for the `consumer()` method appears below:

```
def consumer
  @mutex.synchronize do
    while @queue.empty?
      puts "Oh no! The queue is empty. Please wait.."
```

```
      puts "Oh no! The queue is empty, Please wait.."
      @cond_var.wait(@mutex)
    end
      # Retrieves the element at the first index
    puts "Consumed #{@queue.shift} from the queue"
    @cond_var.signal
  end
end
```

Complete Code

The full code for the blocking queue appears below.

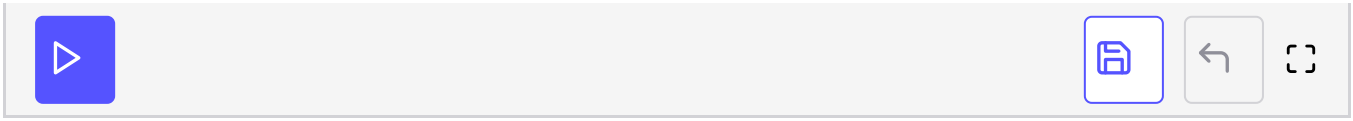
```
#A blocking queue class with initialize, producer and consumer method
s
class BlockingQueue

  # this method declares a simple array as the data structure for blo
cking queue, mutex for synchronization and a condition variable for w
aiting and notifying
  def initialize(capacity)
    @queue_capacity = capacity
    @queue = Array.new()
    @mutex = Mutex.new
    @cond_var = ConditionVariable.new
  end

  # producer method that populates the queue and notifies the waitin
g threads
  def producer(elem)
    @mutex.synchronize do
      while @queue.size == @queue_capacity
        puts "Oh no! The queue is full, Please wait.."
        @cond_var.wait(@mutex)
      end
      # Adds an element to the end of the queue
      @queue << elem
      puts "Added #{elem} to the queue"
      @cond_var.signal
    end
  end

  # consumer method that either consumes items from the queue or wait
s until something is present
  def consumer
```

```
    @mutex.synchronize do
      while @queue.empty?

        puts "Oh no! The queue is empty, Please wait.."
        @cond_var.wait(@mutex)
      end
      # Retrieves the element at the first index
      puts "Consumed #{@queue.shift} from the queue"
      @cond_var.signal

    end
  end
end
```

The test case, in our example, creates a thread that tries to consume from an empty queue. The consumer thread is blocked while the queue gets populated by the producer thread. The consumer thread is unblocked the moment an item is inserted into the queue. Another scenario tested is the queue reaching full capacity. The producer thread gets suspended unless one of the items is taken off the queue by the consumer thread.

```
#A blocking queue class with initialize, producer and consumer methods
class BlockingQueue

  # this method declares a simple array as the data structure for blocking queue,
  # mutex for synchronization and a condition variable for waiting and notifying
  def initialize(capacity)
    @queue_capacity = capacity
    @queue = Array.new()
    @mutex = Mutex.new
    @cond_var = ConditionVariable.new
  end

  # producer method that populates the queue and notifies the waiting threads
  def produce(elem)
    @mutex.synchronize do
      while @queue.size == @queue_capacity
        puts "Oh no! The queue is full, Please wait.."
        @cond_var.wait(@mutex)
      end
        @queue << elem
        puts "Added #{elem} to the queue"
        @cond_var.signal
    end
  end

  # consumer method that either consumes items from the queue or waits until something is pre
  def consume
    @mutex.synchronize do
        while @queue.empty?
          puts "Oh no! The queue is empty, Please wait.."
```

```
          @cond_var.wait(@mutex)
        end
      puts "Consumed #{@queue.shift} from the queue"
      @cond_var.signal

    end
  end
end
blocking_queue = BlockingQueue.new(2)
threads = []
# the first thread will go to sleep until the second
# thread adds an element to the queue, causing the first thread
# to be woken up again
threads << Thread.new do
  puts "Going to consume"
  blocking_queue.consume
end

threads << Thread.new do
  sleep 2
  blocking_queue.produce('item 1')
  blocking_queue.produce('item 2')
  blocking_queue.produce('item 3')
end
threads.each(&:join)
```



Using Monitor for Implementation

We implemented the producer-consumer problem using a mutex and condition variable pair. However, we can also use a `Monitor` object to do the same. The code is almost identical. The changes are highlighted in the code widget below:

```
#A blocking queue class with initialize, producer and consumer methods
class BlockingQueue

  # this method declares a simple array as the data structure for blocking queue,
  # mutex for synchronization and a condition variable for waiting and notifying
  def initialize(capacity)
    @queue_capacity = capacity
    @queue = Array.new()
    @monitor = Monitor.new
    @cond_var = @monitor.new_cond()
  end

  # producer method that populates the queue and notifies the waiting threads
  def producer(elem)
    @monitor.synchronize do
      while @queue.size == @queue_capacity
```

```
      while @queue.size == @queue_capacity
        puts "Oh no! The queue is full, Please wait.."
        @cond_var.wait()
      end
      # Adds an element to the end of the queue
      @queue << elem
      puts "Added #{elem} to the queue"
      @cond_var.signal
    end
  end

  # consumer method that either consumes items from the queue or waits until something is pre
  def consumer
    @monitor.synchronize do
      while @queue.empty?
        puts "Oh no! The queue is empty, Please wait.."
        @cond_var.wait()
      end
      # Retrieves the element at the first index
      puts "Consumed #{@queue.shift} from the queue"
      @cond_var.signal

    end
  end
end

blocking_queue = BlockingQueue.new(2)
threads = []
# the first thread will go to sleep until the second
# thread adds an element to the queue, causing the first thread
# to be woken up again
threads << Thread.new do
  puts "Going to consume"
  blocking_queue.consumer
end

threads << Thread.new do
  sleep 2
  blocking_queue.producer('item 1')
  blocking_queue.producer('item 2')
  blocking_queue.producer('item 3')
end

threads.each(&:join)
```