... continued

This lesson demonstrates how to implement consumer-producer problem using mutexes.

Mutex Implementation

In the previous section we implemented the consumer-producer problem using a mutex and condition variable pair and later using a monitor and condition variable pair. Both of them are the right approaches to solve the problem but for instructional purposes, let's see how we can solve the problem just using a mutex which includes busy waiting.

Let's consider the `producer()` method. We need to check for the size of the queue while holding the mutex lock since queue is a shared data-structure. Say the queue is full and the producer thread needs to wait, but we don't have a condition variable to wait on. The only solution is to continuously check for the predicate to become true in a loop. However, if we don't let go of the mutex, the consumer thread will never get a chance to change the predicate. The trick is to unlock and then immediately lock the mutex again before spinning in the while loop. In between the two actions, the consumer thread should get a chance to acquire the lock and change the predicate. Note that we must lock the mutex every time we check the predicate in the while loop. The complete code for the `producer()` method appears below:

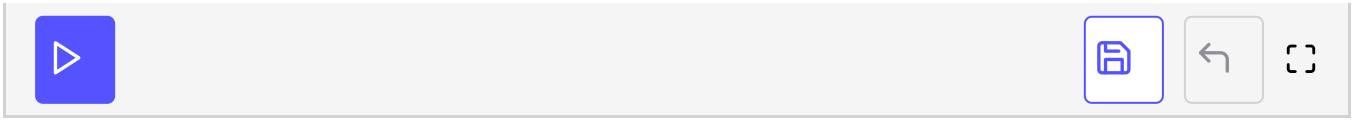
```
def producer(elem)
  @mutex.lock()

  while @queue.size == @queue_capacity
    # puts "Oh no! The queue is full, Please wait.."
    # give chance to consumer to acquire lock
    @mutex.unlock()
    # make sure to acquire lock before checking
    # predicate
    @mutex.lock()
```

```
    end


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

The complete code for the producer-consumer problem using only a mutex appears 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 s
  def initialize(capacity)
    @queue_capacity = capacity
    @queue = Array.new()
    @mutex = Mutex.new
  end

  # producer method that populates the queue and notifies the waiting threads
  def producer(elem)
    @mutex.lock()

    while @queue.size == @queue_capacit
      # give chance to consumer to acquire lock
      @mutex.unlock()
      # make sure to acquire lock before checking
      # the predicate
      @mutex.lock()
    end

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

  # consumer method that either consumes items from the queue or waits until something is pre
  def consumer
    @mutex.lock()
    while @queue.empty?
      # give chance to producer to acquire the lock
      @mutex.unlock()
      # make sure to acquire lock before checking
      # the predicate
      @mutex.lock()
    end
    # Retrieves the element at the first index
    puts "Consumed #{@queue.shift} from the queue"
    @mutex.unlock()
  end
end
```

```
blocking_queue = BlockingQueue.new(5)
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
  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')
  blocking_queue.producer('item 4')
  blocking_queue.producer('item 5')
  blocking_queue.producer('item 6')
  blocking_queue.producer('item 7')
end

threads.each(&:join)
```



> The highlighted lines in the code widget above, show where we consecutively acquire and release the mutex.