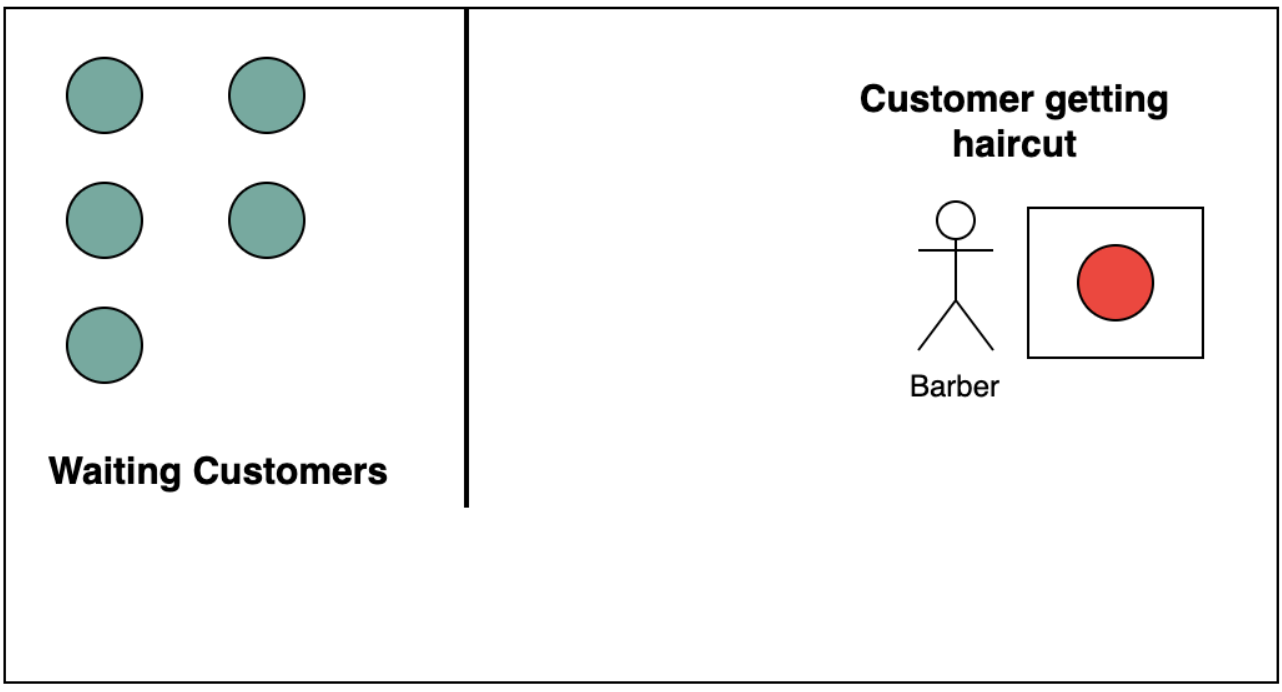
Barber Shop

This lesson visits the synchronization issues when programmatically modeling a hypothetical barber shop and how they can be solved using Ruby's concurrency primitives.

Barber Shop

A similar problem appears in Silberschatz and Galvin's OS book, and variations of this problem exist in the wild.

A barbershop consists of a waiting room with ***n*** chairs, and a barber chair for giving haircuts. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the interaction between the barber and the customers.



Solution

First of all, we need to understand the different state transitions for this problem before we devise a solution. Let's look at them piecemeal:

- A customer enters the shop and if all ***N*** chairs are occupied, he leaves. This hints at maintaining a count of the waiting customers.
- If any of the ***N*** chairs is free, the customer takes up the chair to wait for his turn. Note this translates to using a semaphore on which threads that have found a free chair wait on before being called in by the barber for a haircut.
- If a customer enters the shop and the barber is asleep , it implies there are no customers in the shop. The just-entered customer thread wakes up the barber thread. This sounds like using a signaling construct to wake up the barber thread.

We'll have a class that will expose two APIs, one for the barber thread to execute and the other for customers. The skeleton of the class would look like the following:

```
class BarberShop

  def initialize()
    @totalChairs = 3
    @waitingCustomers = 0
    @haircutsGiven = 0
    @lock = Mutex.new
  end

  def customerWalksIn()

  end

  def barber()
```

```
    while true
    end


  end
end
```

Now let's think about the customer thread. It enters the shop, acquires a lock to test the value of the counter `waitingCustomers`. We must test the value of this variable while no other thread can modify its value, hinting that we'll wrap the test under a mutex. If the value equals all the chairs available, then the customer thread gives up the mutex and returns from the method. If a chair is available the customer thread increments the variable `waitingCustomers`. Remember, the barber might be asleep which can be modeled as the barber thread waiting on a semaphore `waitForCustomerToEnter`. The customer thread must signal the semaphore `waitForCustomerToEnter` in case the barber is asleep.

Next, the customer thread itself needs to wait on a semaphore before the barber comes over, greets the customer, and leads him to the salon chair. Let's call this semaphore `waitForBarberToGetReady`. This is the same semaphore the barber signals as soon as it wakes up. All customer threads waiting for a haircut will block on this `waitForBarberToGetReady` semaphore. The barber thread signaling this semaphore is akin to letting one customer come through and sit on the barber chair for a haircut. This logic when coded looks like the following:

```
  def customerWalksIn()
    @lock.lock()
    if @waitingCustomers == @totalChairs
      puts "Customer walks out, all chairs occupied"
      @lock.unlock()
      return
    end

    @waitingCustomers += 1
    @lock.unlock()

    @waitForCustomerToEnter.release()
    @waitForBarberToGetReady.acquire()

    # TODO: complete the rest of the logic.
  end
```

```
end
```

Now let's work with the barber code. This should be a perpetual loop, where the barber initially waits on the semaphore `waitForCustomerToEnter` to simulate no customers in the shop. If woken up, then it implies that there's at least one customer in the shop who needs a haircut and the barber gets up, greets the customer and leads him to his chair before starting the haircut. This sequence is translated into code as the barber thread signaling the `waitForBarberToGetReady` semaphore. Next, the barber simulates a haircut by sleeping for 50 milliseconds

Once the haircut is done. The barber needs to inform the customer thread too; it does so by signaling the `waitForBarberToCutHair` semaphore. The customer thread should already be waiting on this semaphore.

Finally, to let the barber thread know that the current customer thread has left the barber chair and the barber can bring in the next customer, we make the barber thread wait on yet another semaphore `waitForCustomerToLeave`. This is the same semaphore the customer thread needs to signal before exiting. The barber thread's implementation appears below:

```
def barber()

  while true
    @waitForCustomerToEnter.acquire()
    @waitForBarberToGetReady.release()

    @haircutsGiven += 1

    puts "Barber cutting hair ... #{@haircutsGiven}"
    sleep(0.05)

    @waitForBarberToCutHair.release()
    @waitForCustomerToLeave.acquire()
  end
end
```

The complete customer thread code appears below:

```
def customerWalksIn()
  @lock.lock()

  if @waitingCustomers == @totalChairs
    puts "Customer walks out, all chairs occupied"
    @lock.unlock()
    return
  end

  @waitingCustomers += 1
  @lock.unlock()

  @waitForCustomerToEnter.release()
  @waitForBarberToGetReady.acquire()

  @waitForBarberToCutHair.acquire()
  @waitForCustomerToLeave.release()

  @lock.lock()
  @waitingCustomers -= 1
  @lock.unlock()
end
```

The complete code appears below in the code widget. The test case starts with ten customers trying to get a haircut followed by a batch of another five customers.

main.rb

CountingSemaphore.rb

```
class CountingSemaphore

  def initialize(maxPermits, initialAvailablePermits)
    @maxPermits = maxPermits
    @givenOut = maxPermits - initialAvailablePermits
    @monitor = Monitor.new
    @condVar = @monitor.new_cond()
  end

  def acquire()
    @monitor.enter()

    while @givenOut == @maxPermits
      @condVar.wait()
    end

    @givenOut += 1
    @condVar.broadcast()

    @monitor.exit()
  end

  def release()
```

```
    @monitor.enter()

    while @givenOut == 0
      @condVar.wait()
    end

    @givenOut -= 1
    @condVar.broadcast()

    @monitor.exit()
  end
end
```




The execution output would show 6 customers getting a haircut and the rest walking out since there are only three chairs available at the barbershop.

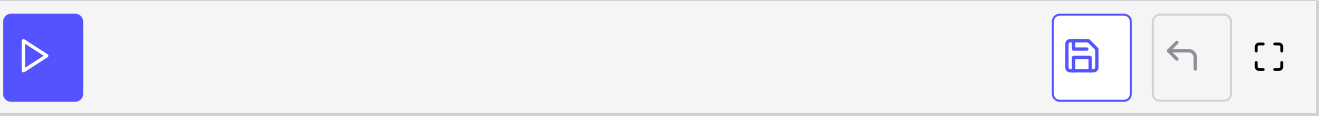
Note we only decrement `waitingCustomers` ***after*** the customer thread has received a haircut. However, you may argue that since the customer getting the haircut has left one spot open in the waiting area, it should be possible to have one more thread come in the shop and wait for a total of four threads. Three threads wait on chairs in the waiting area and one thread occupies the barber chair where it undergoes a haircut. If we tweak our implementation to compensate for this change (**lines 31 - 33**, in the below widget), then we'll see the above test give more than six haircuts. The change entails we decrement the `waitingCustomers` variable right after the barber seats a customer. The code with the change appears below. If you run the widget, you'll see eight threads getting a haircut.

main.rb

CountingSemaphore.rb

```
class CountingSemaphore

  def initialize(maxPermits, initialAvailablePermits)
    @maxPermits = maxPermits
    @givenOut = maxPermits - initialAvailablePermits
    @monitor = Monitor.new
    @condVar = @monitor.new_cond()
  end

  def acquire()
    @monitor.enter()

    while @givenOut == @maxPermits
```

```
      @condVar.wait()
    end

    @givenOut += 1
    @condVar.broadcast()

    @monitor.exit()
  end

  def release()
    @monitor.enter()

    while @givenOut == 0
      @condVar.wait()
    end

    @givenOut -= 1
    @condVar.broadcast()

    @monitor.exit()
  end
end
```



> Note that even with the change, it is possible to see a run with only six haircuts. Consider that threads selected for haircuts get context-switched on **line#26** for each of the two customer batches, then only six customers will get haircuts as none of the threads would have been seated by the barber fast enough on the haircut-chair to decrement the `waitingCustomers` count.