Implementing Semaphore

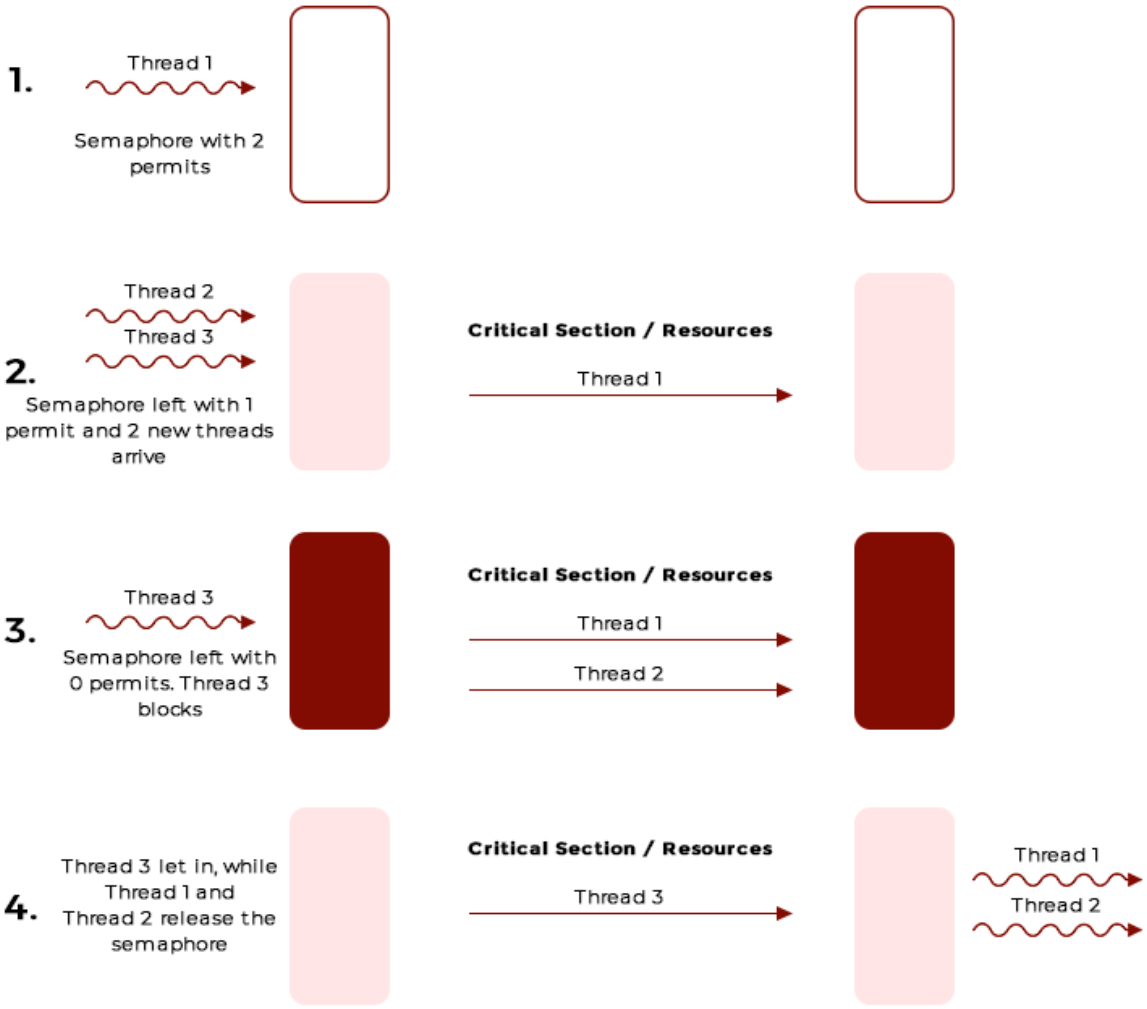
In this lesson, you will learn how to design and implement a simple semaphore class in Ruby.

Implementing Semaphore

Ruby's concurrent gem does provide its own implementation of `Semaphore`; however, it is an interesting exercise to learn to implement a semaphore using a monitor.

Briefly, a semaphore is a construct that allows some threads to access a fixed set of resources in parallel. Always think of a semaphore as having a fixed number of permits to give out. Once all the permits are given out, requesting threads need to wait for a permit to be returned before proceeding forward.

Your task is to implement a semaphore which takes in its constructor the maximum number of permits allowed and is also initialized with the same number of permits. Additionally, if all the permits have been given out, the semaphore blocks threads attempting to acquire it.



Solution

Given the above definition, we can now start to think of what functions our Semaphore class will need to expose. We need a function to "gain the permit" and a function to "return the permit".

1. `acquire()` function to simulate gaining a permit
2. `release()` function to simulate releasing a permit

The constructor accepts an integer parameter defining the number of permits available with the semaphore. Internally we need to store a count which keeps track of the permits given out so far.

The skeleton for our Semaphore class looks something like this so far.

```
class CountingSemaphore

  def initialize(maxPermits)
    @maxPermits = maxPermits
    @givenOut = 0
  end

  def acquire()

  end

  def release()

  end

end
```

Realize that the two methods `acquire()` and `release()` can be invoked at the same time by different threads and therefore we need to guard the logic inside of them by some kind of lock.

We can use the `Monitor` class for mutual exclusion as well as for signaling among threads.

Now let us fill in the implementation for our `acquire()` method. When can a thread not be allowed to acquire a semaphore? When all the permits are given out. This implies we'll need to wait for a permit to become available when `givenOut == maxPermits` If this condition isn't true we simply increment `givenOut` to simulate giving out a permit. The fact that we need to *wait* for a permit if none is available should ring a bell. We can create a condition variable using the monitor object and `wait()` on the condition variable.

The implementation of the `acquire()` method appears below. Note that we are also `broadcast()`-ing the condition variable `condVar` at the end of the method. We'll talk shortly, about why we broadcast instead of signal.

```
  def acquire()

    @monitor.enter()
```

```
    while @givenOut == @maxPermits

      @condVar.wait()
    end

    @givenOut += 1
    @condVar.broadcast()

    @monitor.exit()
  end
```

Implementing the `release()` method should require a simple decrement of the `givenOut` variable. However, when should we block a thread from proceeding forward like we did in `acquire()` method? If `givenOut == 0` then it won't make sense to decrement `givenOut` and we should block on this predicate.

This might seem counter-intuitive, you might ask why would someone call `release()` before calling `acquire()` - This is entirely possible since semaphore can also be used for signaling among threads. A thread can call `release()` on a semaphore object before another thread calls `acquire()` on the same semaphore object. There is no concept of ownership for a semaphore! Hence different threads can call acquire or release methods as they deem fit.

This also means that whenever we decrement or increment the `givenOut` variable we need to invoke `broadcast()` method on the condition variable so that any waiting thread in the other method is able to move forward. The full implementation appears below:

```
class CountingSemaphore

  def initialize(maxPermits)
    @maxPermits = maxPermits
    @givenOut = 0
    @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
```

Follow up Question#1

Note that in both the methods, we increment or decrement the variable `givenOut` before invoking `broadcast()` on the `padlock` variable. Does it matter if we switch the order of the two statements in the two methods? The answer is no. A thread inside either of the methods holds the lock associated with the monitor until the point the thread invokes `monitor.exit()`, which is the last statement in both the methods. A thread waiting on the condition variable when notified must acquire the monitor again, before resuming execution, therefore, the order of the two statements doesn't matter.

Follow up Question#2

Does it matter whether we use `broadcast()` or `signal()`? Reason about it this way: is it possible that two threads are blocked on the condition variable in both the `acquire()` and `release()` methods at the same time?

Let's assume they are then a `signal()` may wake-up the wrong thread whose predicate is still false. The woken-up thread will go back to waiting

and the program would hang since `signal()` wakes up a single thread. However, if threads can only be waiting on the condition variable in the method other than the one from which we signal the condition variable then it should not matter whether we use `broadcast()` or `signal()`. Using `PulseAll()` will allow only one thread out of the waiting ones to make progress and the rest would go back to waiting.

Test Case

In the example below, we have two threads, one always acquires the semaphore while the other always releases it. The execution of the two threads is staggered. We are using the semaphore as a signaling mechanism between the two threads.

```
class CountingSemaphore

  def initialize(maxPermits)
    @maxPermits = maxPermits
    @givenOut = 0
    @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

semaphore = CountingSemaphore.new(1)

t1 = Thread.new do

  sleep(2)
  puts "releasing"
  semaphore.release()

  sleep(2)
  puts "releasing"
  semaphore.release()

  sleep(2)
  puts "releasing"
  semaphore.release()

end

t2 = Thread.new do

  semaphore.acquire()

  puts "acquiring"
  semaphore.acquire()

  puts "acquiring"
  semaphore.acquire()

  puts "acquiring"
  semaphore.acquire()


end

t1.join()
t2.join()
```