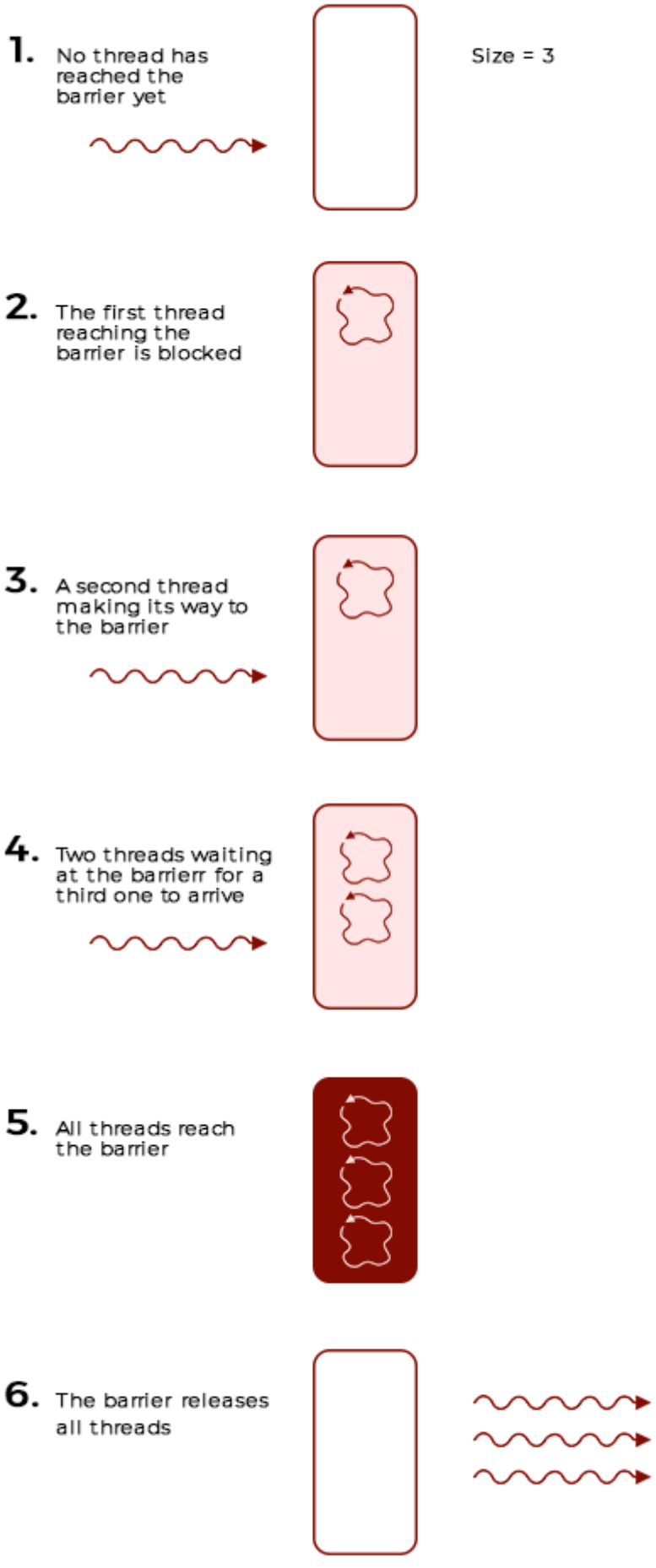
Implementing a Barrier

This lesson discusses how a barrier is implemented in Ruby.

Implementing a Barrier

A barrier can be thought of as a point in the program code, which all or some of the threads need to reach before any one of them is allowed to proceed further.

Working of a Barrier



Solution

A barrier allows multiple threads to congregate at a point in code before any one of the thread is allowed to move forward. Python and most other languages provide libraries that make barrier construct available for developer use. Even though we are re-inventing the wheel but this makes for a good interview question.

We can immediately realize that our solution will need a count variable to track the number of threads that have arrived at the barrier. If we have ***n*** threads, then ***n-1*** threads must wait for the **nth** thread to arrive. This suggests we have the **n-1** threads execute the wait method and the **nth** thread wakes up all the asleep **n-1** threads.

Below is the code:

```
class Barrier

    def initialize(totalThreads)
         @count = 0
         @totalThreads = totalThreads
         @mutex = Mutex.new
         @cv = ConditionVariable.new
    end

    def await
         @mutex.synchronize do
                 # increment the counter whenever a thread arrives at the barrier
                 @count = @count + 1

                 # if you are the nth thread
                 if (@count == @totalThreads)
                         # wake up all the threads
                         @cv.broadcast
                         # remember to reset count so that barrier can be reused
                         @count = 0
                 else
                         # wait if you aren't the nth thread
                         @cv.wait(@mutex)
                 end
         end
    end
end
```

Notice how we are resetting the `count` to zero on **line#20**. This is done so that we are able to re-use the barrier.

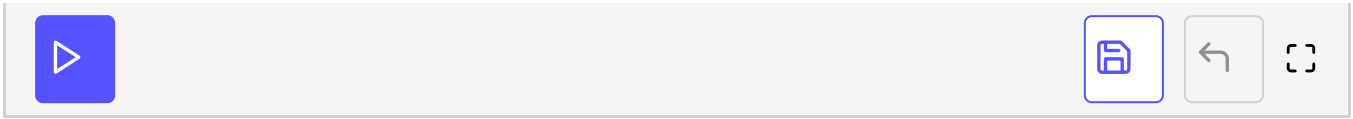
Below is the working code, along with a test case. The test-case creates three threads and has them synchronize on a barrier three times. We introduce sleeps accordingly so that, thread 1 reaches the barrier first, then thread 2 and finally thread 3. None of the threads is able to move forward until all the threads reach the barrier. This is verified by the order in which each thread prints itself in the output.

First Cut

Here's the first stab at the problem:

```
class Barrier

    def initialize(totalThreads)
         @count = 0
         @totalThreads = totalThreads
         @mutex = Mutex.new
         @cv = ConditionVariable.new
    end

    def await
         @mutex.synchronize do
                 # increment the counter whenever a thread arrives at the barrier
                 @count = @count + 1

                 # if you are the nth thread
                 if (@count == @totalThreads)
                         # wake up all the threads
                         @cv.broadcast
                         # remember to reset count so that barrier can be reused
                         @count = 0
                 else
                         # wait if you aren't the nth thread
                         @cv.wait(@mutex)
                 end
         end
    end
end

barrier = Barrier.new(3)

p1 = Thread.new do

  p "Thread 1"
  barrier.await
```

```
  p "Thread 1"
  barrier.await
  p "Thread 1"
  barrier.await

end

p2 = Thread.new do

  sleep(0.5)
  p "Thread 2"
  barrier.await
  sleep(0.5)
  p "Thread 2"
  barrier.await
  sleep(0.5)
  p "Thread 2"
  barrier.await

end

p3 = Thread.new do

  sleep(1.5)
  p "Thread 3"
  barrier.await
  sleep(1.5)
  p "Thread 3"
  barrier.await
  sleep(1.5)
  p "Thread 3"
  barrier.await

end

p1.join
p2.join
p3.join
```



When you run the above code, you'll see that the threads print themselves in order i.e., first thread 1, then thread 2 and finally thread 3 prints. Thread 1, after reaching the barrier, waits for the other two threads to reach the barrier before moving forward.

The above code has a subtle but very crucial bug! Can you spot the bug and try to fix it before reading on?

Second Cut

The previous code would have been hunky dory if we were guaranteed

that no spurious wake-ups could ever occur. The `wait()` method invocation on the mutex without the while loop is an error. We discussed in previous sections that `wait()` should always be used with a while loop that checks for a condition/predicate and if found false should make the thread `wait()` again.

The condition the while loop can check for is simply how many threads have incremented the `count` variable so far. A thread that wakes up spuriously should go back to sleep if the `count` is less than the total number of threads. We can check for this condition as follows:

```
while(@count < @totalThreads)
  @cv.wait(@mutex)
end
```

The while loop introduces another problem. When the last thread does a `broadcast()` it also resets the `count` to 0, which means the threads that are legitimately woken up will always be stuck in the while loop because `count` is immediately set to zero. What we really want is not to reset the `count` variable to zero until all the threads escape the while condition when `count` becomes `totalThreads`. Below is the improved version:

```
class Barrier
    def initialize(totalThreads)
        @count = 0
        @released = 0
        @totalThreads = totalThreads
        @mutex = Mutex.new
        @cv = ConditionVariable.new
    end

    def await
        @mutex.synchronize do
                # increment the counter whenever a thread arrives at the barrier
                @count = @count + 1

                # if you are the nth thread
                if (@count == @totalThreads)
                        # wake up all the threads
                        @cv.broadcast
                        # remember to reset count so that barrier can be reused
                        @released = @totalThreads;
                else
                        # wait till all threads reach barrier
                        while (@count < @totalThreads)
                                @cv.wait(@mutex)
                        end
```

```
                end
                @released = @released - 1
                if (@released == 0)
                        @count = 0
                end
        end
    end
end
```

The above code introduces a new variable `released` that keeps tracks of how many threads exit the barrier and when the last thread exits the barrier it resets `count` to zero, so that the barrier object can be reused in the future. There is still a bug in the above code! Can you guess what it is?

Final Cut

To understand why the above code is broken, consider three threads t1, t2, and t3 trying to `await()` on a barrier object in an infinite loop. Note the following sequence of events:

1. Threads t1 and t2 invoke `await()` and end up waiting at line#24. The `count` variable is set to 2 and any spurious wakeups will cause t1 and t2 to go back to waiting.
2. Thread t3 comes along, executes the if block on line#16 and finds `count` == `totalThreads`. Thread t3 doesn't wait, notifies threads t1 and t2 to wakeup and exits.
3. **If thread t3 attempts to invoke `await()` immediately after exiting it and is also granted the monitor before threads t1 or t2 get a chance to acquire the monitor then the `count` variable will be incremented to 4.**
4. With `count` equal to 4, t3 will not block at the barrier and exit which breaks the contract for the barrier.
5. The invocation order of the `await()` method was t1, t2, t3, and t3 again. The right behavior would have been to release t1, t2, or t3 in any order and then block t3 on its second invocation of the `await()` method.
6. Another flaw with the above code is, it can cause a deadlock. Suppose

we wanted the three threads t1, t2, and t3 to congregate at a barrier

twice. The first invocation was in the order [t1, t2, t3] and the second was in the order [t3, t2, t1]. If t3 immediately invoked `await()` after the first barrier, it would go past the second barrier without stopping while t2 and t1 would become stranded at the second barrier, since `count` would never equal `totalThreads`.

The fix requires us to block any new threads from proceeding until all the threads that have reached the previous barrier are released. The code with the fix appears below:

```
class Barrier

        def initialize(totalThreads)
        @count = 0
         @released = 0
         @totalThreads = totalThreads
         @mutex = Mutex.new
         @cv = ConditionVariable.new
        end

        def await
         @mutex.synchronize do
                 while (@count == @totalThreads)
                         @cv.wait(@mutex)
                 end

                 @count = @count + 1

                 if (@count == @totalThreads)
                         @cv.broadcast
                         @released = @totalThreads
                 else
                         while(@count < @totalThreads)
                                 @cv.wait(@mutex)
                         end
                 end
                 @released = @released - 1
                 if (@released == 0)
                         @count = 0
                         @cv.broadcast
                 end
         end
        end
end

barrier = Barrier.new(3)

p1 = Thread.new do
```

```
        p "Thread 1"
        barrier.await
        p "Thread 1"

        barrier.await
        p "Thread 1"
        barrier.await

        end

p2 = Thread.new do

        sleep(0.5)
        p "Thread 2"
        barrier.await
        sleep(0.5)
        p "Thread 2"
        barrier.await
        sleep(0.5)
        p "Thread 2"
        barrier.await

        end

p3 = Thread.new do

        sleep(1.5)
        p "Thread 3"
        barrier.await
        sleep(1.5)
        p "Thread 3"
        barrier.await
        sleep(1.5)
        p "Thread 3"
        barrier.await

        end

p1.join
p2.join
p3.join
```