Asynchronous to Synchronous Problem

In this lesson, we will study a real-life interview question asking to convert asynchronous execution to synchronous execution.

Asynchronous to Synchronous Problem

This is an actual interview question asked at Netflix.

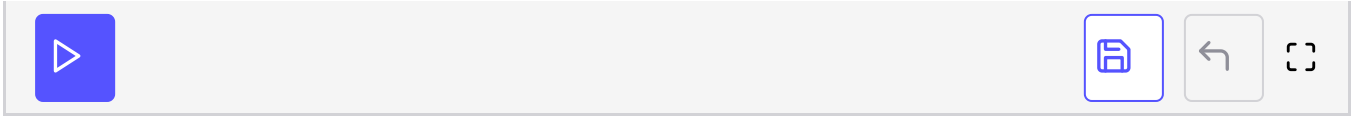
Imagine we have an `AsyncExecutor` class that performs some useful task asynchronously via the method `execute()`. In addition, the method accepts a function object that acts as a callback and gets invoked after the asynchronous execution is done. The definition of the involved classes is below. The asynchronous work is simulated using sleep. A passed-in call is invoked to let the invoker take any desired action after the asynchronous processing is complete.

Executor Class

```
class AsyncExecutor

  def work(callback)
    sleep(5)
    callback.call()
  end

  def execute_async(callback)
    Thread.new do
      work(callback)
    end
  end
end
```

An example run would be as follows:

```
class AsyncExecutor

  def work(callback)
    # simulate work by sleeping for 5 seconds
    sleep(5)
    callback.call()
  end

  def execute(callback)
    Thread.new do
      work(callback)
    end
  end
end


def sayHi()
  puts "hi"
end

asyncExecutor = AsyncExecutor.new
cb = Proc.new { sayHi() }
asyncExecutor.execute(cb)

puts "main program exiting"

# sleep for the async task to complete
sleep(6)
```



Note how the main thread exits before the asynchronous execution is completed. The message "Hi" is printed after the main thread has exited.

Your task is to make the execution synchronous without changing the original classes (imagine that you are given the binaries and not the source code) so that the main thread waits till the asynchronous execution is complete. In other words, the highlighted **line#25** only executes once the asynchronous task is complete.

Solution

The problem here asks us to convert asynchronous code to synchronous code without modifying the original code. The requirement that the main

thread should block until the asynchronous execution is complete hints at

using some kind of notification/signaling mechanism. The main thread *waits* on something, which is then *signaled* by the asynchronous execution thread. Semaphore is the first thought that may come to your mind for solving this problem and is one of the right approaches. However, since we don't have a semaphore class available in Ruby's standard library, we can use a condition variable and a mutex pair to achieve the same functionality.

Since we can't modify the original code, we'll extend a new class `SynchronousExecutor` from the given `AsyncExecutor` class and override the `execute()` method. The trick here is to invoke the original asynchronous implementation using `super()` inside the overridden method. The class would look as follows:

```
class SyncExecutor < AsyncExecutor

  def initialize()
    @mutex = Mutex.new
    @cv = ConditionVariable.new
    @is_done = false
  end

  def work(callback)
    super(callback)

    # notify after returning from the above call
  end

  def execute(callback)
    super(callback)

    # wait for async callback to complete execution
    # before returning
  end
end
```

We create a `ConditionVariable` and a `Mutex` object for signaling purposes. Inside the `execute` method, rather than returning immediately like the base class method, we wait on the condition variable that is only signaled in the `work()` method, once the callback has completed execution.

The complete code appears below:

```
class SyncExecutor < AsyncExecutor

  def initialize()
    @mutex = Mutex.new
    @cv = ConditionVariable.new
    @is_done = false
  end

  def work(callback)
    super(callback)

    puts "#{Thread.current.__id__} thread notifying"
    @mutex.lock()
    @cv.broadcast()
    @is_done = true
    @mutex.unlock()
  end

  def execute(callback)
    super(callback)

    @mutex.lock()
    while @is_done == false
      @cv.wait(@mutex)
    end
    puts "#{Thread.current.__id__} thread woken-up"
    @mutex.unlock()
  end
end
```

The main thread that invokes `execute()` gets blocked on the condition variable object's `wait()` method on **line#24**. The base class `AsynExecutor` spawns another thread that simulates the work being done and then invokes the `SyncExecutor` class's `work()` method. The `SyncExecutor` class overrides the `work()` method from the base class. It invokes the base class's implementation and then signals the condition variable allowing the main thread to resume execution. All in all, the main thread remains blocked until the asynchronous execution is complete and the passed in callback has been invoked.

The complete code appears in the code widget below. If you run the program, the message from the main thread prints last.

```
class AsyncExecutor

  def work(callback)
    sleep(5)
    callback.call()
  end

  def execute(callback)
    Thread.new do
      work(callback)
    end
  end
end


class SyncExecutor < AsyncExecutor

  def initialize()
    @mutex = Mutex.new
    @cv = ConditionVariable.new
    @is_done = false
  end

  def work(callback)
    super(callback)

    puts "#{Thread.current.__id__} thread notifying"
    @mutex.lock()
    @cv.broadcast()
    @is_done = true
    @mutex.unlock()
  end

  def execute(callback)
    super(callback)

    @mutex.lock()
    while @is_done == false
      @cv.wait(@mutex)
    end
    puts "#{Thread.current.__id__} thread woken-up"
    @mutex.unlock()
  end
end

def sayHi()
  puts "hi"
end

exec = SyncExecutor.new
cb = Proc.new { sayHi() }
exec.execute(cb)

puts "main program exiting"

sleep(6)
```