Monitor

This lesson introduces the use of Monitor in Ruby.

Monitor

We have discussed the general concept of a monitor in a previous lesson and now we'll see Ruby's implementation of a monitor. A monitor provides mutual exclusion and the ability for a thread to wait on one or more condition variables.

We can create a monitor as follows:

```
monitor = Monitor.new
```

For mutual exclusion, we can use instance methods `enter()` and `exit()` to wrap a block of code. For instance:

```
monitor = Monitor.new

monitor.enter()
#... critical section
monitor.exit()
```

Or we can use `synchronize` alias on the monitor object, which automatically enters and exits the monitor for us. For instance:

```
monitor = Monitor.new

monitor.synchronize {
    #... critical section
}
```

It is preferable to use the alias with the monitor object because it will ensure that the monitor's `exit()` method is invoked even if the critical section raises an exception.

We can create condition variables associated with a monitor as follows:

```
monitor = Monitor.new
cv = monitor.new_cond()
```

With these building blocks, let's write a program with two threads. The main thread spawns a child that waits on the monitor's condition variable which is signaled by the main thread. The code appears in the code widget below:

```
require 'monitor'

monitor = Monitor.new
cv = monitor.new_cond()

child = Thread.new do
  monitor.enter()

  # susceptible to spurious wakeup
  cv.wait()
  monitor.exit()
  puts "Child thread exiting"
end

sleep(1)

monitor.enter()
cv.signal()
monitor.exit()

child.join()
  puts "Main thread exiting"
```

This example above may seem similar to the example we saw earlier which used an instance of the `Mutex` class and an instance of the `ConditionVariable` class to coordinate between two threads. One difference between using a monitor versus rolling your own using a combination of a mutex and a condition variable is that an exception is thrown when we attempt to signal a condition variable of a monitor without locking it first. This is demonstrated below:

```
require 'monitor'

monitor = Monitor.new
cv = monitor.new_cond()

# throws an exception
cv.signal()
```

In contrast, no exception is raised if we `signal()` an instance of `ConditionVariable` without locking a mutex that we intend to associate with the condition variable. Signaling in such a way is incorrect and shown below:

```
cv = ConditionVariable.new
mutex = Mutex.new

# doesn't throw exception but incorrect
cv.signal()
```

Ping Pong Example

Let's rewrite our ping pong example using the monitor. The logic is very similar to how we designed the solution when using a mutex and a condition variable. The complete code appears in the code widget below:

```
monitor = Monitor.new
cv = monitor.new_cond()

flag = true
keepRunning = true

pingThread = Thread.new do

  while keepRunning
```

```ruby
    # enter the monitor
    monitor.synchronize {

      while flag == true
        cv.wait()
      end

      puts "ping"
      flag = true

      # remember to wake up the other
      # waiting thread
      cv.signal()
    }
  end
end


pongThread = Thread.new do

  while keepRunning

    monitor.synchronize do

      while flag == false
        cv.wait()
      end

      puts "pong"
      flag = false

      cv.signal()
    end
  end
end


# run simulation for 10 seconds
sleep(10)
keepRunning = false
pingThread.join()
pongThread.join()
```