... continued

This lesson continues the discussion on the use of condition variables in Ruby.

Spurious Wakeups

Condition variables suffer from spurious wakeups - that is, wakeup of threads waiting on the condition variable without being signaled. This is specifically allowed by the POSIX standard because it allows more efficient implementations of condition variables under some circumstances. Even though the official documentation doesn't mention this peculiarity, it is a common pattern across all programming languages including Python, Java, etc.

Idiomatic Usage

The idiomatic usage of condition variables when waiting is as follows:

Waiting on Condition Variable

```
mutex.lock()

while predicate

    cv.wait()

end

mutex.unlock()
```

The while loop ensures that if a thread experiences a spurious wakeup it goes back to waiting since the predicate or the condition for the while loop hasn't changed.

A newbie mistake is to use an if-condition instead of a while loop shown as follows:

Incorrect Waiting on Condition Variable

```
mutex.lock()

# NEVER DO THIS !!!
if predicate

    cv.wait()

end

mutex.unlock()
```

For one, using the above incorrect pattern doesn't protect against spurious wakeups. Second, another thread can change the predicate in the time between the condition variable is signaled and a waiting thread is woken up. If the waking thread doesn't recheck the predicate after waking up, it might end up proceeding forward with the predicate being false.

Similarly, when invoking `signal()` or `broadcast()` the idiomatic usage requires us to wrap the call in a mutex synchronization block. This should be the same mutex that is passed into the corresponding `wait()` call.

```
mutex.lock()
cv.signal()
mutex.unlock()
```

Note that failing to follow the above pattern will not result in an exception but the resulting code will not be thread-safe.

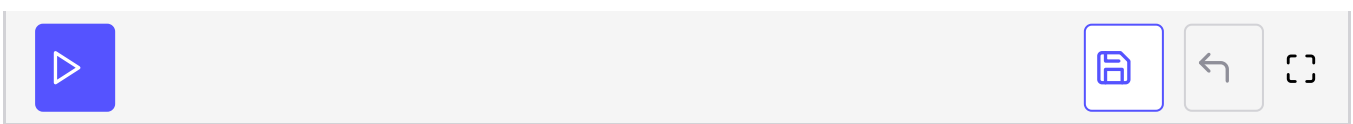
Ping Pong Example

Let's rewrite the ping pong example for two threads using condition variables. The predicate, in this case, would be the boolean `flag` variable, which is set to true when it's ping thread's turn to print and false when it's pong thread's turn to print.

Note that we always mutate and read the boolean `flag` while holding the mutex lock. This is important because we don't want to have a thread read the flag value and make a decision to move forward whilst it might get mutated by the other thread.

get initated by the other thread.

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

flag = true
keepRunning = true

pingThread = Thread.new do

  while keepRunning

    mutex.lock()

    while flag == true
      cv.wait(mutex)
    end

    puts "ping"
    flag = true

    # remember to wake up the other
    # waiting thread
    cv.signal()
    mutex.unlock()

  end
end


pongThread = Thread.new do

  while keepRunning

    mutex.lock()

    while flag == false
      cv.wait(mutex)
    end

    puts "pong"
    flag = false

    cv.signal()
    mutex.unlock()
  end
end


# run simulation for 10 seconds
sleep(5)
keepRunning = false
pingThread.join()
pongThread.join()
```



> Now let's rewrite the ping-pong example with five threads of each type. Note that even though we have five threads executing the ping block but only one of them is executing in the block at any given time because we are guarding the block using a mutex which ensures mutual exclusion. The `broadcast()` wakes up all waiting threads but only one of them is able to make progress while the rest go back to waiting if they find the predicate already false to make progress.

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

flag = true

pingThreads = 5.times do |i|
  Thread.new(i) do |arg|

    while true

      mutex.lock()

      while flag == true
        cv.wait(mutex)
      end

      puts "ping #{arg}"
      flag = true
      cv.broadcast()

      mutex.unlock()
    end
  end
end


pongThreads = 5.times.map do
  Thread.new do

    while true

      mutex.lock()

      while flag == false
        cv.wait(mutex)
      end

      puts "pong"
      flag = false
      cv.broadcast()
```

```
      mutex.unlock()
    end
  end

end


# run simulation for 5 seconds
sleep(5)
```