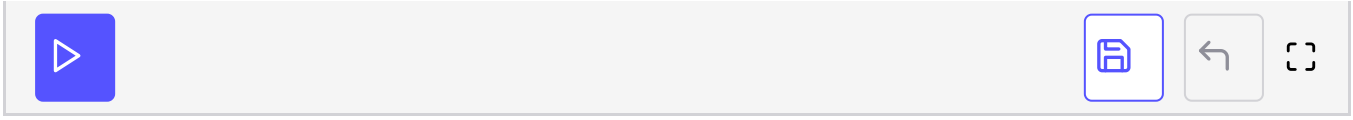
... continued

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

> As an illustration of the importance of the pattern to use when working with condition variables, we'll rewrite our ping-pong program to print ping and pong on the console alternatively. Instead of having two blocks, we'll have a single block of code that prints the two strings. That is we don't have separate code blocks, one to print each string. We'll create ten threads that execute the same code snippet and print ping or pong based on the value of the boolean `flag` variable. We also introduce an additional variable `active` that is used to gate entry into the critical section of the code implicitly.
>
> We have taken a departure from the idiomatic use of condition variables and used an `if` condition instead of a `while`.
>
> Take a minute to read the code in the snippet below:

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

flag = true
active = true

10.times.map do |i|

  Thread.new(i) do |arg|

    while true
      mutex.lock()
      if active == false
        cv.wait(mutex)
      end
      active = false
      mutex.unlock()

      if flag
        puts "ping by thread #{arg}"
```

```
      else
        puts "pong"
      end

      flag = !flag

      mutex.lock()
      active = true
      cv.signal()
      mutex.unlock()

      # Only to slow down the program
      sleep(0.01)

    end
  end
end


sleep(20)
```



In the snippet above **lines#19 - 25** form a critical section. Only a single thread should execute in this region. However, we aren't using a mutex to achieve that instead we are utilizing condition variables to ensure that a single thread executes in this critical section. The associated predicate is the `active` variable, which is mutated within mutex synchronized blocks. Note that **lines#12 - 17** let a single thread set the variable `active` to false and proceed forward, all subsequent threads get blocked on the condition variable's `wait()` method. Granted, that our code is unwieldy and can be simplified but the intent is to demonstrate the ill-effects of using condition variables without a corresponding while loop.

Now that you understand how the code is structured, can you think of a scenario where two pings or pongs get printed in succession?

1. Imagine **Thread A** starts, flips the variable `active` to false and gets switched out.
2. **Thread B** starts and enters the first mutex block but gets blocked on the condition variable's `wait()` method because `active` is false.
3. **Thread B** gets context switched and **Thread A** is rescheduled.

4. **Thread A** `signal()`s the condition variable but doesn't get context-switched.

5. **Thread B** is woken up but it is blocked on acquiring the mutex which is currently being held by **Thread A**.

6. **Thread A** luckily gets more time on the CPU and completes another loop. Say it prints ping on the console and before it gets a chance to flip the `flag` variable it gets context-switched.

7. Now **Thread B** which was woken up in step#5 get a chance to acquire the mutex and proceeds. The flag hasn't been flipped yet and it too prints ping on the console, thus violating the program constraint.

If we used `broadcast()` instead of `signal()`, the issue will be exacerbated.

If you run the code widget above you may or may not see the same string printed twice as there are many variables that affect thread scheduling. However, if you copy/paste the script in the code widget above and run it for a few seconds on your machine, you'll likely see the same string being printed in succession. The code widget, behind the scenes, runs on a VM with limited compute resources and the number of things that can go wrong in a multithreaded program is reduced. Thus race conditions may not be easily reproduced.

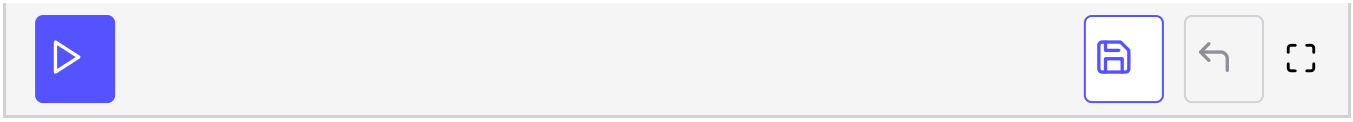
The fix for the program is to wrap the condition variable's `wait()` method in a `while` condition instead of an `if`.

Missed Signal

The last topic we'll discuss in the context of condition variables is the missed signal. When a condition variable is signaled, it wakes up one of the threads already waiting on the condition variable. If there no waiting threads, the signal is lost.

Consider the snippet below:

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


t1 = Thread.new do

  # listen for a signal after a second
  sleep(1)
  mutex.synchronize {
    puts "WAITING"
    cv.wait(mutex)
  }
  puts "EXITNG"
end


t2 = Thread.new do

  # broadcast the condition variable
  mutex.synchronize {
    cv.broadcast()
  }

end

# t1 will be stuck forever, unless a spurious wakeup
# occurs.
t1.join()

mutex.synchronize {
  cv.signal(mutex)
}

t2.join()
```



> Thread **t2** issues a `broadcast()` before thread **t1** gets a chance to `wait()` on the condition variable, causing the program to enter into a deadlock. The framework throws an exception warning us about a deadlock.
>
> We can associate an additional state in the form of a predicate with the condition variable or use a semaphore to capture the signaling by the second thread.