Condition Variables

This lesson explores the use of condition variables in Ruby.

Condition Variables

Synchronization mechanisms need more than just mutual exclusion; a general need is to be able to wait for another thread to do something. Condition variables provide mutual exclusion and the ability for threads to wait for a predicate to become true.

Condition variables is always associated with a lock and in case of Ruby that lock is an instance of the `Mutex` class. Using condition variables allows us to rid ourselves of writing busy-wait loops that constantly poll for a condition to become true.

The `Condition` class can be used to create an instance of a condition variable. Note that we'll also need to create a mutex object to work in conjunction with the condition variable.

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

wait

The `wait()` method accepts a mutex object as an argument. The idiomatic usage of a condition variable requires us to first synchronize on the mutex object and then invoke `wait()` on the condition variable while passing in the same mutex object that we synchronized on. The idiomatic usages are as follows:

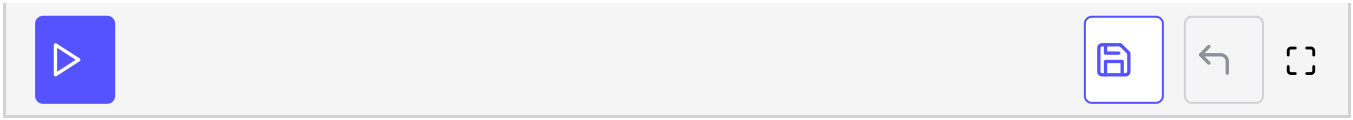
```
mutex.lock()
cv.wait(mutex)
puts "statement executed when cv is signaled"
```

```
mutex.unlock()
```

A thread gives up the mutex associated with the condition variable when it invokes `wait()` on the condition variable. The mutex lock is reacquired by the thread when the condition variable is signaled.

Consider the code widget below, where we try to wait on the condition variable without locking the mutex first. When the code widget is run, it throws an exception.

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

# Attempting to wait on the condition variable
# without locking the mutex first
cv.wait(mutex)
```



Signal

A thread can call `signal()` method to wake up one of the threads waiting on the condition variable. The woken up thread will immediately try to reacquire the mutex that was passed-in at the time `wait()` was invoked.

Broadcast

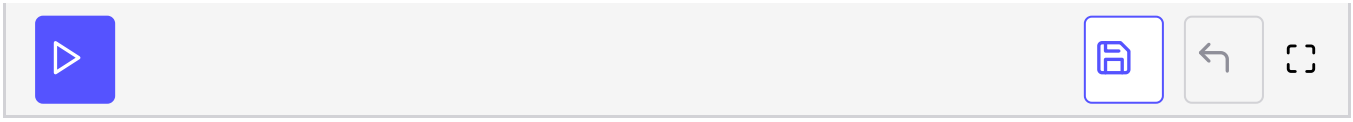
The `broadcast()` method wakes up all the threads waiting on the condition variable. However, only one thread will be able to acquire the mutex lock, while the rest will be blocked until they can acquire the lock.

Example

In the example presented in the code widget below, the main thread spawns a child thread. The child thread waits on the condition variable, which the main thread signals after a second.

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

```
thread = Thread.new do

  mutex.lock()
  cv.wait(mutex)
  mutex.unlock()

  puts "Child thread exiting"
end


# give a chance to the child thread to
# wait on the condition variable
sleep(1)

# signal the condition variable
cv.signal()

# wait for child thread to exit
sleep(1)

puts "Main thread exiting"
```



In the second example, we spawn three child threads and each of them waits on the condition variable. When we `broadcast()` the condition variable, all waiting threads are woken up but only one gets to acquire the mutex and proceed forward. Once the woken up thread releases the mutex, other blocked threads are able to make progress.

```
cv = ConditionVariable.new
mutex = Mutex.new
threads = []

threads = 3.times.map do |i|
  thread = Thread.new(i) do |arg|

    mutex.lock()
    cv.wait(mutex)

    sleep(1)
    puts "Child thread exiting #{arg}"
    mutex.unlock()
  end
end

# give a chance to all the child threads to
# wait on the condition variable
sleep(1)

# signal the condition variable
```

```
# signal the condition variable
cv.broadcast()

# wait for child thread to exit
sleep(7)

puts "Main thread exiting"
```