Mutex

This lesson discusses Mutex as used in Ruby.

Mutex

`Mutex` is the most basic or primitive synchronization construct available in Ruby. It offers two methods: `lock()` and `unlock()`. A `Mutex` object can only be unlocked by a thread that locked it in the first place. Consider the snippet below, where the child and the main thread alternatively acquire the mutex. The main thread blocks for the period the child holds the mutex.

```
require 'date'

# Create a Mutex object
mutex = Mutex.new

# Spawn a child thread
Thread.new do
  puts("Child thread acquiring mutex")
  mutex.lock()
  sleep(3)
  puts("Child thread releasing mutex")
  mutex.unlock()
end

# Sleep main thread to give child thread a
# chance to acquire the mutex first
sleep(1)
puts("Main thread attempting to acquire mutex")
mutex.lock()
puts("Main thread acquires mutex")
mutex.unlock()
```

As mentioned in the previous introductory sections, a mutex has a notion of ownership. The `Thread` class has an instance method `owned?()` which can be used to know if the current thread has locked the mutex. Consider the code widget below; the main thread becomes an owner after it locks the mutex.

```
mutex = Mutex.new

# prints false
puts mutex.owned?()

# acquire the mutex
mutex.lock()

# prints true
puts mutex.owned?()

# unlock the mutex
mutex.unlock()
```

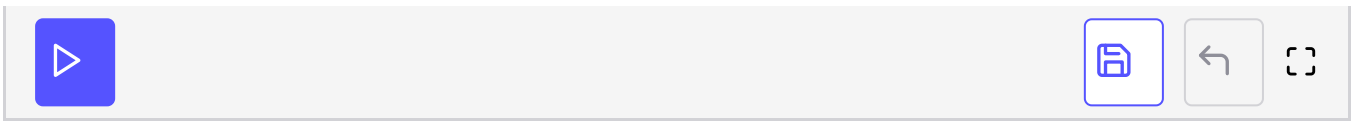
Ping Pong Example

Recall the ping pong example from the earlier section. We discussed that the busy-wait version of the ping pong program was thread-safe because it involved only two threads and the code was structured such that only one thread printed on the console at any time while the other thread waited in the loop.

Do you think the program will work correctly if we increased the number of threads printing pong and ping each by five? Let's try out in the code widget below. For easy reading of the output, we print the thread number for the threads that execute the ping code block.

```
flag = true
keepRunning = true
```

```
mutex = Mutex.new
pingThreads = []
pongThreads = []

pingThreads = 5.times.map do |i|

  Thread.new(i) do |id|

    while true

      while flag == true
        # busy-wait
      end

      puts "ping #{id}"
      sleep(0.01)
      flag = true
    end
  end
end

pongThreads = 5.times.map do

  Thread.new do

    while true

      while flag == false
        # busy-wait
      end

      puts "pong"
      flag = false
      sleep(0.1)

    end
  end
end


# run simulation for 10 seconds
sleep(10)
```



Note that in the program above, we add sleep statements to affect the faulty outcome. Without sleep, the code widget may not show two pings or pongs consecutively. Thread scheduling is unpredictable and dependent on underlying OS.

The program isn't thread-safe with more than one thread executing the ping or pong block. Consider the following sequence which can result in printing ping twice on the console

printing ping twice on the console.

1. Assume two threads **Thread A** and **Thread B** are waiting in the inner while loop waiting for `flag` to be set to false.
2. **Thread A** gets scheduled, finds the loop condition false and breaks out.
3. **Thread A** prints ping on the console and before it gets a chance to set `flag` to true, it gets switched out.
4. **Thread B** gets scheduled, finds `flag` set to false, breaks out of loop and prints ping on the console.
5. The two threads loop over to fall into the inner while loop again and wait for `flag` to be false.

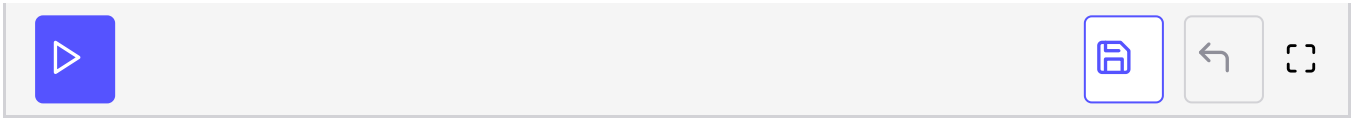
We can fix the above program by using a mutex. The trick is whenever a thread checks for the while loop's condition, it should do so while holding the mutex. If it finds the while loop condition to be true it continues to hold the mutex, prints on the console and releases the mutex for another thread to acquire.

```
    mutex.lock()
    while flag == true
      mutex.unlock()
      # busy-wait
      mutex.lock()
    end

    puts "ping #{arg}"
    flag = true
    mutex.unlock()
```

The fixed code appears in the code widget below:

```
flag = true
mutex = Mutex.new
pingThreads = []
pongThreads = []

pingThreads = 5.times.map do |i|
```

```
pingThreads = 5.times.map do |i|

  Thread.new(i) do |arg|

    while true

      mutex.lock()
      while flag == true
        mutex.unlock()
        # busy-wait
        mutex.lock()
      end

      puts "ping #{arg}"
      sleep(0.01)
      flag = true
      mutex.unlock()
    end
  end
end

pongThreads = 5.times.map do

  Thread.new do

    while true

      mutex.lock()
      while flag == false
        mutex.unlock()
        # busy-wait
        mutex.lock()
      end

      puts "pong"
      flag = false
      sleep(0.1)
      mutex.unlock()
    end
  end
end


# run simulation for 10 seconds
sleep(10)
```



> From the output we can verify that ping and pong are printed alternatively.