Threads in Ruby

A multithreaded program has more than one thread of execution. Within each thread, statements are executed sequentially, but the threads themselves may be executed in parallel on a multicore CPU, for example. Often on a single CPU machine, multiple threads are not actually executed in parallel, but parallelism is simulated by interleaving the execution of the threads.

Ruby makes it easy to write multithreaded programs with the Thread class. Ruby threads are a lightweight and efficient way to achieve concurrency in your code.

Creating Ruby Threads

To start a new thread, just associate a block with a call to Thread.new. A new thread will be created to execute the code in the block, and the original thread will return from Thread.new immediately and resume execution with the next statement.

```ruby
# Thread #1 is running here
Thread.new {
   # Thread #2 runs this code
}
# Thread #1 resumes this code
def method1
   index = 0
   while index<5
      puts "method1 at: #{Time.now}"
      sleep(5)
      index = index+1
   end
end
```

```
def method2
   index = 0
   while index<5
      puts "method2 at: #{Time.now}"
      sleep(5)
      index = index+1
   end
end

puts "Main Thread Started At #{Time.now}"
thread1 = Thread.new{method1()} # A new threads are created with
Thread.new
# There is no need to start a thread after creating it, it begins
running automatically when CPU resources become available.
# A thread runs the code in the block associated with the call to
Thread.new and then it stops running.

thread2 = Thread.new{method2()}
thread1.join
thread2.join
#  You can wait for a particular thread to finish by calling that thread's
#Thread.join method. The calling thread will block until the given #thread is
#finished.
puts "End of main thread at #{Time.now}"

output:
Main Thread Started At 2017-09-26 08:03:21 +0530
method2 at: 2017-09-26 08:03:21 +0530
method1 at: 2017-09-26 08:03:21 +0530
method2 at: 2017-09-26 08:03:26 +0530
method1 at: 2017-09-26 08:03:26 +0530
method1 at: 2017-09-26 08:03:31 +0530
method2 at: 2017-09-26 08:03:31 +0530
method2 at: 2017-09-26 08:03:36 +0530
method1 at: 2017-09-26 08:03:36 +0530
method1 at: 2017-09-26 08:03:41 +0530
method2 at: 2017-09-26 08:03:41 +0530
End of main thread at 2017-09-26 08:03:46 +0530
```

Race Conditions

```
class Product
  class << self; attr_accessor :price end
  @price = 0
end

(1..5).each { Product.price += 10 }

puts "Product.price = #{Product.price}"

output:
Product.price = 50
```

Multithreaded version of this code:

```
class Product
  class << self; attr_accessor :price end
  @price = 0
end

threads = (1..5).map do |i|
  Thread.new(i) do |i|
    prod_price = Product.price # Reading value
    sleep(rand(0..7))
    prod_price += 10          # Updating value
    sleep(rand(0..7))
    Product.price = prod_price # Writing value
  end
end

threads.each {|t| t.join}
puts "Product.price = #{Product.price}"
```

```
output:
Product.price = 10
```

This is what a race condition does.

To fix race conditions, we have to control the program so that when one thread is doing work another should wait until the working thread finishes. This is called Mutual Exclusion and we use this concept to remove race conditions in our programs.

```
class Product
  class << self; attr_accessor :price end
  @price = 0
end

mutex = Mutex.new

threads = (1..5).map do |i|
  Thread.new(i) do |i|
    mutex.synchronize do  # One and only one thread can access the block
#wrapped in mutex.synchronize at any time. Other threads have to wait until the current
#thread that is processing completes.
      prod_price = Product.price # Reading value
      sleep(rand(0..7))
      prod_price += 10        # Updating value
      sleep(rand(0..7))
      Product.price = prod_price # Writing value
    end
  end
end

threads.each {|t| t.join}

puts "Product.price = #{Product.price}"
```

```
output:
Product.price = 100
```

Avoiding the Deadlock

Deadlock is the condition that occurs when all threads are waiting to acquire a resource held by another thread. Because all threads are blocked, they cannot release the locks they hold. And because they cannot release the locks, no other thread can acquire those locks.

This is where *condition variables* come into picture. A *condition variable* is simply a semaphore that is associated with a resource and is used within the protection of a particular *mutex*. When you need a resource that's unavailable, you wait on a condition variable. That action releases the lock on the corresponding *mutex*. When some other thread signals that the resource is available, the original thread comes off the wait and simultaneously regains the lock on the critical region.

```
require 'thread'
mutex = Mutex.new

var = ConditionVariable.new
thread1 = Thread.new {
   mutex.synchronize {
      puts "Entered in the critical secton for thread 1 and I was
asked to wait on a condition !!"
      var.wait(mutex)
      puts "A: I am executing the critical section finally !!"
   }
}

puts "In the main thread !!"
```

```
thread2 = Thread.new {
   mutex.synchronize {
      puts "Entered in the critical secton for thread 2 !!"
      var.signal
      puts "B: I am still there in the critical, finally wrapping up
!!"
   }
}
thread1.join
thread2.join

puts "Main thread finishing up !!"
```

```
output:
In the main thread !!
Entered in the critical secton for thread 1 and I was asked to wait on
a condition !!
Entered in the critical secton for thread 2 !!
B: I am still there in the critical, finally wrapping up !!
A: I am executing the critical section finally !!
Main thread finishing up !!
```