Project 8 Ruby

The Ruby extension for this project is an explanation of threading/parallelism in the language.

Threads are implemented within the Ruby interpreter. This gives them the benefit of OS portability but the drawback of low-priority threads possibly not getting to run.
Ruby threads are also constrained to run on one processor at a time.

To run a create a new thread called t, use
t = Thread.new{#code to run goes here}
Thread.start can be used as well, to the same effect.

Here is an example with a one-line piece of code to execute, and one with multiple lines.

```
# create thread that will execute contained code right away
t2 = Thread.new {puts "Hello, I'm t1"}
t3 = Thread.new{
    puts "Hi!\n"
    puts "I'm t3\n"
}
```

We can also pass a function in:

```
def func()
    puts "hello from t4\n"
end

t4 = Thread.new {func()}
```

Threads are set to run automatically in Ruby. To make the thread not run until called, we can wrap it in a proc and call it later, like this:

```
# create thread that is not going to run right away
t1 = proc { Thread.new { puts "\nhi!!!!!!!!!!!" } }
# run the thread
t1.call
```

t.join can be used to join a thread of name *t*

Some useful methods in the thread class:
Thread.main returns a reference to the main thread
Thread.list returns an array of all thread objects that are runnable or stopped
Thread.current returns the thread currently executing
Thread.kill stops the current thread
Thread.pass passes execution to a thread other than the currently running one
Thread.exit exits the main thread

To use a mutex, first place *require 'thread'* at the top of the file
To create it, use *mutex* = Mutex.new
Mutex.synchronize{#insert code here} will lock and unlock the contained code.

Within the mutex section, in order to make one thread wait until another thread finishes, we can pass a signal between them, using a "condition variable".
Provided the condition variable has been declared before outside of both thread A and B's mutex sections, using
cv = ConditionVariable.new
we can then make thread A wait until thread B sends a signal regarding cv until it resumes, with
cv.wait(mutex)
in A and

cv.signal
in B.
Here is a clearer example:

```ruby
cv = ConditionVariable.new
a = Thread.new {
   mutex.synchronize {
      puts "a: I am critical but will wait for cv"
      cv.wait(mutex)
      puts "a: I am now critical again"
   }
}

b = Thread.new {
   mutex.synchronize {
      puts "b: now I am critical"
      cv.signal
      puts "b: I am still critical but done with cv"
   }
}
```

This prints:
a: I am critical but will wait for cv
b: now I am critical
b: I am still critical but done with cv
a: I am now critical again