MI-RUB Fibers, Threads, and Processes

Pavel Strnad
pavel.strnad@fel.cvut.cz

Dept. of Computer Science, FEE CTU Prague,
Karlovo nám. 13, 121 35
Praha, Czech Republic

MI-RUB, WS 2011/12

Contents

1 Fibers

2 Multithreading

3 Mutual Exclusion

Fibers

Fibers

- Fibers are a coroutine mechanism.
- Fibers are not a lightweight threads.
- Fibers were introduced in Ruby 1.9.

Fibers

Fibers

- Fibers are a coroutine mechanism.
- Fibers are not a lightweight threads.
- Fibers were introduced in Ruby 1.9.

Fibers

Fibers

- Fibers are a coroutine mechanism.
- Fibers are not a lightweight threads.
- Fibers were introduced in Ruby 1.9.

Fiber Example

Simple task

We'd like to analyze a text file, counting the occurrence of each word.

Fiber Example

Solution without Fibers

```
counts = Hash.new(0)
File.foreach("testfile") do |line|
    line.scan(/\w+/) do |word|
        word = word.downcase
        counts[word] += 1
    end
end
counts.keys.sort.each {|k| print "#{k}:#{counts[k]} "}

produces:
and:1 is:3 line:3 on:1 one:1 so:1 this:3 three:1 two:1
```

What is the problem?

Fiber Example

Problem

This code combines the concepts of finding words with the counting of the words.

Solution?

We could fix this by writing a method that reads the file and yields each successive word. But *fibers* give us a simpler solution.

Handled Exception

Solution with Fibers

```
words = Fiber.new do
    File.foreach("testfile") do |line|
        line.scan(/\w+/) do |word|
            Fiber.yield word.downcase
        end
    end
end

counts = Hash.new(0)
while word = words.resume
    counts[word] += 1
end
counts.keys.sort.each {|k| print "#{k}:#{counts[k]} "}

produces:
and:1 is:3 line:3 on:1 one:1 so:1 this:3 three:1 two:1
```

Fibers

Example Continue

When the fiber runs out of words in the file, the block exits. The next time resume is called, it returns nil (because the block has exited). (You'll get a **FiberError** if you attempt to call resume again after this.)

Fibers

Fibers Properties

- Fibers are often used to generate values from infinite sequences.
- Fibers can be resumed only in the thread which created them, but Fiber library adds full coroutine support (adds a *transfer* method allowing to transfer control flow to other fiber).

Fibers Exercise

Exercise 1

Write a fiber that implements Fibonacci sequence. Example of usage:

```
for i in (0..100) do
    value = fibonacci.resume
    puts value
end
```

Threads

Threads

- Often the simplest way to do two things at once.
- Prior to Ruby 1.9, these were implemented as so-called green threads—threads were switched totally within the interpreter.
- Now, it uses native operating system threads but operates only a **single thread at a time**.
- Ruby extension libraries are not thread safe (because they were written for old model).

Threads Example

```
require 'net/http'
pages = %w( www.rubycentral.com slashdot.org www.google.com )
threads = []
for page_to_fetch in pages
    threads << Thread.new(page_to_fetch) do |url|
        h = Net::HTTP.new(url, 80)
        print "Fetching: #{url}\n"
        resp = h.get('/')
        print "Got #{url}: #{resp.message}\n"
    end
end
threads.each {|thr| thr.join }
```

Threads

```
Thread.new(page_to_fetch) do |url|
```

Thread.new

Threads are created using Thread.new. We can pass any number of arguments.

Threads

```
print "Got #{url}: #{resp.message}\n"
```

raise

It is better to use print rather puts because puts can be interleaved. Puts do its work in two chunks and between them other thread can be scheduled.

Threads

```
threads.each {|thr| thr.join }
```

raise

We need to join running threads. When a Ruby program terminates, all threads are killed, regardless of their states.

Thread Variables

Thread Variables

- Local variables are local in a thread. No one can access them.
- Threads can be treated as hashes using [].

Thread Variables Example

```
count = 0
threads = []
10.times do |i|
    threads[i] = Thread.new do
        sleep(rand(0.1))
        Thread.current["mycount"] = count
        count += 1
    end
end
threads.each {|t| t.join; print t["mycount"], ", " }
puts "count = #{count}"
```

Threads and Exceptions

What happens if a thread raises an unhandled exception?

It depends on the setting of the abort_on_exception flag and on the setting of the interpreter's debug flag. The abort_on_exception is false and the debug flag is not enabled be default, then *unhandled exception* **kills the current thread** and **all the rest continue to run**!

Threads and Exceptions Example

```
threads = []
4.times do |number|
    threads << Thread.new(number) do |i|
        raise "Boom!" if i == 2
        print "#{i}\n"
    end
end
sleep 1
```

Threads and Exceptions Example

If you join to a thread that has raised an exception, then that exception will be raised in the thread that does the joining

```
threads = []
4.times do |number|
    threads << Thread.new(number) do |i|
        raise "Boom!" if i == 2
        print "#{i}\n"
    end
end
threads.each do |t|
    begin
         t.join
    rescue RuntimeError => e
         puts "Failed: #{e.message}"
    end
end
```

Threads and abort_on_exception Example

Once thread 2 dies, no more output is produced.

```
Thread.abort_on_exception = true
threads = []
4.times do |number|
    threads << Thread.new(number) do |i|
         raise "Boom!" if i == 2
         print "#{i}\n"
    end
end
threads.each {|t| t.join }
```

Mutual Exclusion Example Not Correct

```
def inc(n)
    n + 1
end
sum = 0
threads = (1..10).map do
    Thread.new do
        10_000.times do
            sum = inc(sum)
        end
    end
end
threads.each(&:join)
p sum
```

What is the problem? What is desired output?

Mutual Exclusion

Mutual Exclusion Primitive

- Controls access to a shared resource.
- Schedules threads.
- One thread at a time.
- Provide by **Mutex** class.

Mutual Exclusion Example Correct

```
def inc(n)
    n + 1
end
sum = 0
mutex = Mutex.new
threads = (1..10).map do
    Thread.new do
         10_000.times do
               mutex.synchronize do ####
                    sum = inc(sum)      # one at a time, please
               end                               ####
          end
    end
end
threads.each(&:join)
p sum
```

Queue

The Queue class, located in the thread library, implements a threadsafe queuing mechanism. Multiple threads can add and remove objects from each queue, and each addition and removal is guaranteed to be atomic.

Exercise 2

Rewrite the following code using Queue and Thread.

```
words = Fiber.new do
    File.foreach("testfile") do |line|
        line.scan(/\w+/) do |word|
            Fiber.yield word.downcase
        end
    end
end

counts = Hash.new(0)
while word = words.resume
    counts[word] += 1
end
counts.keys.sort.each {|k| print "#{k}:#{counts[k]} "}
```