Chapter 12

# Fibers, Threads, and Processes

Ruby gives you two basic ways to organize your program so that you can run different parts of it apparently "at the same time." Fibers let you suspend execution of one part of your program and run some other part. For more decoupled execution, you can split up cooperating tasks *within* the program, using multiple threads, or you can split up tasks between different programs, using multiple processes. Let's look at each in turn.

## Fibers

1.9

Ruby 1.9 introduced *fibers* to the language. Although the name suggests some kind of lightweight thread, in reality Ruby's fibers are really just a very simple coroutine mechanism. They allow you to write programs that look like you are manually scheduling threads without incurring any of the complexity inherent in threading. Let's look at a simple example. We'd like to analyze a text file, counting the occurrence of each word. We could do this (without using fibers) in a simple loop:

Download **samples/tutthreads_1.rb**

```
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
```

However, this code is messy because it conflates the concepts of finding words with the counting of the words.

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

Download samples/tutthreads_2.rb

```
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
```

The constructor for the Fiber class takes a block and returns a fiber object. For now, the code in the block is not executed.

Subsequently, we can call resume on the fiber object. This causes the block to start execution. The file is opened, and the scan method starts extracting individual words. However, at this point, Fiber.yield is invoked. This suspends execution of the block—the resume method that we called to run the block returns any value given to Fiber.yield.

Our main program enters the body of the loop and increments the count for the first word returned by the fiber. It then loops back up to the top of the while loop, which again calls words.resume while evaluating the condition. The resume call goes back into the block, continuing just after it left off (at the line after the Fiber.yield call).

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 are often used to generate values from infinite sequences on demand. Here's a fiber that returns successive integers divisible by 2 and not divisible by 3:

Download samples/tutthreads_3.rb

```
twos = Fiber.new do
   num = 2
   loop do
     Fiber.yield(num) unless num % 3 == 0
     num += 2
   end
end
10.times { print twos.resume, " " }
```

*produces:*

```
2 4 8 10 14 16 20 22 26 28
```

Because fibers are just objects, you can pass them around, store them in variables, and so on. Fibers can be resumed only in the thread that created them.

## Fibers, Coroutines, and Continuations

The basic fiber support in Ruby is limited—fibers can yield control only back to the code that resumed them. However, Ruby comes with two standard libraries that extend this behavior. The fiber library (described on page 754) adds full coroutine support. Once it is loaded, fibers gain a transfer method, allowing them to transfer control to arbitrary other fibers.

A related but more general mechanism is the *continuation*. A continuation is a way of recording the state of your running program (where it is, the current binding, and so on) and then resuming from that state at some point in the future. You can use continuations to implement coroutines (and other new control structures). Continuations have also been used to store the state of a running web application between requests—a continuation is created when the application sends a response to the browser; then, when the next request arrives from that browser, the continuation is invoked, and the application continues from where it left off. You enable continuations in Ruby by requiring the continuation library, described on page 738.

# Multithreading

1.9

Often the simplest way to do two things at once is by using *Ruby threads*. Prior to Ruby 1.9, these were implemented as so-called green threads—threads were switched totally within the interpreter. In Ruby 1.9, threading is now performed by the operating system. This is an improvement, but not quite as big an improvement as you might want. Although threads can now take advantage of multiple processors (and multiple cores in a single processor), there's a major catch. Many Ruby extension libraries are not thread safe (because they were written for the old threading model). So, Ruby compromises: it uses native operating system threads but operates only a single thread at a time. You'll never see two threads in the same application running Ruby code truly concurrently. (You will, however, see threads busy doing (say) I/O while another thread executes Ruby code. That's part of the point....)

## Creating Ruby Threads

Creating a new thread is pretty straightforward. The code that follows is a simple example. It downloads a set of web pages in parallel. For each URL that it is asked to download, the code creates a separate thread that handles the HTTP transaction.

Download **samples/tutthreads_4.rb**

```
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 }
```

*produces:*

```
Fetching: www.rubycentral.com
Fetching: slashdot.org
Fetching: www.google.com
Got www.google.com:  OK
Got www.rubycentral.com:  OK
Got slashdot.org:  OK
```

Let's look at this code in more detail, because a few subtle things are happening.

New threads are created with the Thread.new call. It is given a block that contains the code to be run in a new thread. In our case, the block uses the net/http library to fetch the top page from each of our nominated sites. Our tracing clearly shows that these fetches are going on in parallel.

When we create the thread, we pass the required URL as a parameter. This parameter is passed to the block as url. Why do we do this, rather than simply using the value of the variable page_to_fetch within the block?

A thread shares all global, instance, and local variables that are in existence at the time the thread starts. As anyone with a kid brother can tell you, sharing isn't always a good thing. In this case, all three threads would share the variable page_to_fetch. The first thread gets started, and page_to_fetch is set to "www.rubycentral.com". In the meantime, the loop creating the threads is still running. The second time around, page_to_fetch gets set to "slashdot.org". If the first thread has not yet finished using the page_to_fetch variable, it will suddenly start using this new value. These kinds of bugs are difficult to track down.

However, local variables created within a thread's block are truly local to that thread—each thread will have its own copy of these variables. In our case, the variable url will be set at the time the thread is created, and each thread will have its own copy of the page address. You can pass any number of arguments into the block via Thread.new.

This code also illustrates a gotcha. Inside the loop, the threads use print to write out the messages, rather than puts. Why? Because behind the scenes, puts splits its work into two chunks: it writes its argument, and then it writes a newline. Between these two, a thread could get scheduled, and the output would be interleaved. Calling print with a single string that already contains the newline gets around the problem.

## Manipulating Threads

Another subtlety occurs on the last line in our download program. Why do we call join on each of the threads we created?

When a Ruby program terminates, all threads are killed, regardless of their states. However, 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. By calling join on each of the requester threads, you can make sure that all three requests have completed before you terminate the main program. If you don't want to block forever, you can give join a timeout parameter—if the timeout expires before the thread terminates, the join call returns nil. Another variant of join, the method Thread#value, returns the value of the last statement executed by the thread.

In addition to join, a few other handy routines are used to manipulate threads. The current thread is always accessible using Thread.current. You can obtain a list of all threads using Thread.list, which returns a list of all Thread objects that are runnable or stopped. To determine the status of a particular thread, you can use Thread#status and Thread#alive?.

In addition, you can adjust the priority of a thread using Thread#priority= . Higher-priority threads will run before lower-priority threads. We'll talk more about thread scheduling, and stopping and starting threads, in just a bit.

## Thread Variables

A thread can normally access any variables that are in scope when the thread is created. Variables local to the block containing the thread code are local to the thread and are not shared.

But what if you need per-thread variables that can be accessed by other threads—including the main thread? Class Thread features a special facility that allows thread-local variables to be created and accessed by name. You simply treat the thread object as if it were a Hash, writing to elements using []= and reading them back using []. In the example that follows, each thread records the current value of the variable count in a thread-local variable with the key mycount. To do this, the code uses the string "mycount" when indexing thread objects. (A *race condition*[1] exists in this code, but we haven't talked about synchronization yet, so we'll just quietly ignore it for now.)

Download **samples/tutthreads_6.rb**

```
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}"
```

1. A race condition occurs when two or more pieces of code (or hardware) both try to access some shared resource, and the outcome changes depending on the order in which they do so. In the example here, it is possible for one thread to set the value of its mycount variable to count, but before it gets a chance to increment count, the thread gets descheduled and another thread reuses the same value of count. These issues are fixed by synchronizing the access to shared resources (such as the count variable).

*produces:*

```
7, 0, 8, 6, 5, 4, 1, 9, 3, 2, count = 10
```

The main thread waits for the subthreads to finish and then prints out the value of count captured by each. Just to make it more interesting, we have each thread wait a random time before recording the value.

## Threads and Exceptions

What happens if a thread raises an unhandled exception? It depends on the setting of the abort_on_exception flag (documented on pages 705 and 707) and on the setting of the interpreter's *debug* flag (described on page 234).

If abort_on_exception is false and the debug flag is not enabled (the default condition), an unhandled exception simply kills the current thread—all the rest continue to run. In fact, you don't even hear about the exception until you issue a join on the thread that raised it. In the following example, thread 2 blows up and fails to produce any output. However, you can still see the trace from the other threads.

Download **samples/tutthreads_7.rb**

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

*produces:*

```
0
1
3
```

You normally don't use sleep to wait for threads to terminate. Instead, you'll use the join method. If you join to a thread that has raised an exception, then that exception will be raised in the thread that does the joining:

Download **samples/tutthreads_8.rb**

```
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
```

*produces:*

```
0
1
3
Failed: Boom!
```

However, set abort_on_exception to true or use -d to turn on the debug flag, and an unhandled exception kills all running threads. Once thread 2 dies, no more output is produced.

Download **samples/tutthreads_9.rb**

```
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 }
```

*produces:*

```
0
1
3
prog.rb:5:in `block (2 levels) in <main>': Boom! (RuntimeError)
```

# Controlling the Thread Scheduler

In a well-designed application, you'll normally just let threads do their thing; building timing dependencies into a multithreaded application is generally considered to be bad form, because it makes the code far more complex and also prevents the thread scheduler from optimizing the execution of your program.

Class Thread provides a number of methods that control the scheduler. Invoking Thread.stop stops the current thread, and invoking Thread#run arranges for a particular thread to be run. Thread.pass deschedules the current thread, allowing others to run, and Thread#join and Thread#value suspend the calling thread until a given thread finishes. These last two are the only low-level thread control methods that the average program should use. In fact, I now consider most of the other low-level thread control methods too dangerous to use correctly in programs I write.[2] Fortunately, Ruby has support for higher-level thread synchronization.

2. And, worse, some of these primitives are unsafe in use. Charles Nutter of JRuby fame has a blog post that illustrates one problem:
http://headius.blogspot.com/2008/02/rubys-threadraise-threadkill-timeoutrb.html

# Mutual Exclusion

Let's start by looking at a simple example of a race condition—two threads updating a shared variable:

Download samples/tutthreads_10.rb

```
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
```

*produces:*

```
17335
```

We create 10 threads, and each increments the shared sum variable 10,000 times. And yet, when the threads all finish, the final value in sum is considerably less than 100,000. Clearly we have a race condition. In one thread, we call inc, passing it the current value in sum—let's say that value is 99. It returns the new value 100, which we assign back into sum. But what happens if, during that sequence, another thread gets scheduled? It also passes the value 99 to inc. Let's say the second thread finishes the call to inc first. It assigns 100 back into sum. Then the first thread gets rescheduled and finishes its call to inc. That call returns 100 as well, which gets assigned into sum. So, we had two calls, in two threads, but the overall effect was that sum changed only from 99 to 100. We lost data.

Fortunately, that's easy to fix. We can use the built-in class Mutex to create synchronized regions—areas of code that only one thread may enter at a time.

Some schools coordinate students' access to the bathrooms during class time using a system of bathroom passes. Each room has two passes, one for girls and one for boys. To visit the bathroom, you have to take the appropriate pass with you. If someone else already has that pass, you have to cross your legs and wait for them to return. The bathroom pass controls access to the critical resource—you have to own the pass to use the resource, and only one person can own it at a time.

A mutex is like that bathroom pass. You create a mutex to control access to a resource and then lock it when you want to use that resource. If no one else has it locked, your thread continues to run. If someone else has already locked that particular mutex, your thread suspends until they unlock it.

Here's a version of our counting code that uses a mutex to ensure that only one thread updates the count at a time:

Download samples/tutthreads_11.rb

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

*produces:*

```
100000
```

This pattern is so common that the Mutex class provides Mutex#synchronize, which locks the mutex, runs the code in a block, then unlocks the mutex. This also ensures that the mutex will get unlocked even if an exception is thrown while it is locked.

Download samples/tutthreads_12.rb

```
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
```

*produces:*

```
100000
```

There are times when you want to claim a mutex lock if the mutex is currently unlocked, but you don't want to suspend the current thread if it isn't. The Mutex#try_lock method does just

that, taking the lock if it can, but returning false if the lock is already taken. The following code illustrates a hypothetical currency converter. The ExchangeRates class caches rates from an online feed, and a background thread updates that cache once an hour. This update takes a minute or so. In the main thread, we interact with our user. However, rather than just go dead if we can't claim the mutex that protects the rate object, we use try_lock and print a status message if the update is in process.

```
rate_mutex = Mutex.new
exchange_rates = ExchangeRates.new
exchange_rates.update_from_online_feed
Thread.new do
  loop do
    sleep 3600
    rate_mutex.synchronize do
      exchange_rates.update_from_online_feed
    end
  end
end
loop do
  print "Enter currency code and amount: "
  line = gets
  if rate_mutex.try_lock
    begin
      puts exchange_rates.convert(line)
    ensure
      rate_mutex.unlock
    end
  else
    puts "Sorry, rates being updated. Try again in a minute"
  end
end
```

If you are holding the lock on a mutex and you want to temporarily unlock it, allowing others to use it, you can call Mutex#sleep. We could use this to rewrite the previous example:

```
rate_mutex = Mutex.new
exchange_rates = ExchangeRates.new
exchange_rates.update_from_online_feed
Thread.new do
  rate_mutex.lock
  loop do
    rate_mutex.sleep 3600
    exchange_rates.update_from_online_feed
  end
end
loop do
  print "Enter currency code and amount: "
  line = gets
  if rate_mutex.try_lock
    begin
      puts exchange_rates.convert(line)
```

```
      ensure
        rate_mutex.unlock
      end
    else
      puts "Sorry, rates being updated. Try again in a minute"
    end
  end
```

## Queues and Condition Variables

Most of the examples in this chapter use the Mutex class for synchronization. However, another technique is useful, particularly when you need to synchronize work between producers and consumers. The Queue class, located in the thread library, implements a thread-safe queuing mechanism. Multiple threads can add and remove objects from each queue, and each addition and removal is guaranteed to be atomic. For an example, see the description of the thread library on page 817.

A condition variable is a controlled way of communicating an event (or a condition) between two threads. One thread can wait on the condition, and the other can signal it. The thread library extends threads with condition variables. Again, see the library description for an example.

# Running Multiple Processes

Sometimes you may want to split a task into several process-sized chunks—maybe to take advantage of all those cores in your shiny new processor. Or perhaps you need to run a separate process that was not written in Ruby. Not a problem: Ruby has a number of methods by which you may spawn and manage separate processes.

## Spawning New Processes

You have several ways to spawn a separate process; the easiest is to run some command and wait for it to complete. You may find yourself doing this to run some separate command or retrieve data from the host system. Ruby does this for you with the system and backquote (or backtick) methods:

```
system("tar xzf test.tgz")   # =>   true
result = `date`
result                       # =>   "Mon Apr 13 13:26:03 CDT 2009\n"
```

1.9

The method Kernel.system executes the given command in a subprocess; it returns true if the command was found and executed properly. It raises an exception if the command cannot be found. It returns false if the command ran but returned an error. In case of failure, you'll find the subprocess's exit code in the global variable $?.

One problem with system is that the command's output will simply go to the same destination as your program's output, which may not be what you want. To capture the standard

output of a subprocess, you can use the backquote characters, as with `date` in the previous example. Remember that you may need to use String#chomp to remove the line-ending characters from the result.

OK, this is fine for simple cases—we can run some other process and get the return status. But many times we need a bit more control than that. We'd like to carry on a conversation with the subprocess, possibly sending it data and possibly getting some back. The method IO.popen does just this. The popen method runs a command as a subprocess and connects that subprocess's standard input and standard output to a Ruby IO object. Write to the IO object, and the subprocess can read it on standard input. Whatever the subprocess writes is available in the Ruby program by reading from the IO object.

For example, on our systems one of the more useful utilities is pig, a program that reads words from standard input and prints them in pig latin (or igpay atinlay). We can use this when our Ruby programs need to send us output that our five-year-olds shouldn't be able to understand:

```
pig = IO.popen("/usr/local/rubybook/bin/pig", "w+")
pig.puts "ice cream after they go to bed"
pig.close_write
puts pig.gets
```

*produces:*

```
iceway eamcray afterway eythay ogay otay edbay
```

This example illustrates both the apparent simplicity and the more subtle real-world complexities involved in driving subprocesses through pipes. The code certainly looks simple enough: open the pipe, write a phrase, and read back the response. But it turns out that the pig program doesn't flush the output it writes. Our original attempt at this example, which had a pig.puts followed by a pig.gets, hung forever. The pig program processed our input, but its response was never written to the pipe. We had to insert the pig.close_write line. This sends an end-of-file to pig's standard input, and the output we're looking for gets flushed as pig terminates.

popen has one more twist. If the command you pass it is a single minus sign (–), popen will fork a new Ruby interpreter. Both this and the original interpreter will continue running by returning from the popen. The original process will receive an IO object back, and the child will receive nil. This works only on operating systems that support the fork(2) call (and for now this excludes Windows).

Download **samples/tutthreads_17.rb**

```
pipe = IO.popen("-","w+")
if pipe
  pipe.puts "Get a job!"
  STDERR.puts "Child says '#{pipe.gets.chomp}'"
else
  STDERR.puts "Dad says '#{gets.chomp}'"
  puts "OK"
end
```

*produces:*

```
Dad says 'Get a job!'
Child says 'OK'
```

In addition to the popen method, some platforms support the methods Kernel.fork, Kernel.exec, and IO.pipe. The filenaming convention of many IO methods and Kernel.open will also spawn subprocesses if you put a | as the first character of the filename (see the introduction to class IO on page 546 for details). Note that you *cannot* create pipes using File.new; it's just for files.

## Independent Children

Sometimes we don't need to be quite so hands-on; we'd like to give the subprocess its assignment and then go on about our business. Sometime later, we'll check to see whether it has finished. For instance, we may want to kick off a long-running external sort:

```
exec("sort testfile > output.txt") if fork.nil?
# The sort is now running in a child process
# carry on processing in the main program
# ... dum di dum ...
# then wait for the sort to finish
Process.wait
```

The call to Kernel.fork returns a process ID in the parent, and nil in the child, so the child process will perform the Kernel.exec call and run sort. Sometime later, we issue a Process.wait call, which waits for the sort to complete (and returns its process ID).

If you'd rather be notified when a child exits (instead of just waiting around), you can set up a signal handler using Kernel.trap (described on page 579). Here we set up a trap on SIGCLD, which is the signal sent on "death of child process":

```
trap("CLD") do
  pid = Process.wait
  puts "Child pid #{pid}: terminated"
end
fork { exec("sort testfile > output.txt") }
# Do other stuff...
```

*produces:*

```
Child pid 83170: terminated
```

For more information on using and controlling external processes, see the documentation for Kernel.open, IO.popen, and the section on the Process module on page 641.

## Blocks and Subprocesses

IO.popen works with a block in pretty much the same way as File.open does. If you pass it a command, such as date, the block will be passed an IO object as a parameter:

Download **samples/tutthreads_20.rb**

```
IO.popen("date") {|f| puts "Date is #{f.gets}" }
```

*produces:*

```
Date is Mon Apr 13 13:26:03 CDT 2009
```

The IO object will be closed automatically when the code block exits, just as it is with File.open.

If you associate a block with Kernel.fork, the code in the block will be run in a Ruby subprocess, and the parent will continue after the block:

Download **samples/tutthreads_21.rb**

```
fork do
  puts "In child, pid = #$$"
  exit 99
end
pid = Process.wait
puts "Child terminated, pid = #{pid}, status = #{$?.exitstatus}"
```

*produces:*

```
In child, pid = 83177
Child terminated, pid = 83177, status = 99
```

$? is a global variable that contains information on the termination of a subprocess. See the section on Process::Status beginning on page 650 for more information.