Table of Contents

20. Multitasking and Multithreading

You can't concentrate on more than What's six times nine? one thing at once. You won't get very far reading this book if someone is interrupting you every five seconds asking you to do arithmetic problems. But any computer with a modern operating system can do many things at once. More precisely, it can simulate that ability by switching very quickly back and forth between tasks.

In a multitasking operating system, each program, or process, gets its own space in memory and a share of the CPU's time. Every time you start the Ruby interpreter, it runs in a new process. On Unix-based systems, your script can spawn subprocesses: this feature is very useful for running external command-line programs and using the results in your own scripts (see Recipes 20.8 and 20.9, for instance).

The main problem with processes is that they're expensive. It's hard to read while people are asking you to do arithmetic, not because either activity is particularly difficult, but because it takes time to switch from one to the other. An operating system spends a lot of its time as overhead, switching between processes, trying to make sure each one gets a fair share of the CPU's time.

The other problem with processes is that it's difficult to get them to communicate with each other. For simple cases, you can use techniques like those described in Recipe 20.8. You can implement more complex cases with Inter-Process Communication and named pipes, but we say, don't bother. If you want your Ruby program to do two things at once, you're better off writing your code with threads.

A thread is a sort of lightweight process that runs inside a real process. One Ruby process can host any number of threads, all running more or less simultaneously. It's faster to switch between threads than to switch between processes, and since all of a process's threads run in the same memory space, they can communicate simply by sharing variables.

Recipe 20.3 covers the basics of multithreaded programming. We use threads throughout this book, except when only a subprocess will work (see, for instance, Recipe 20.1). Some recipes in other chapters, like Recipes 3.12 and 14.4, show threads used in context.

Ruby implements its own threads, rather than using the operating system's implementation. This means that multithreaded code will work exactly the same way across platforms. Code that spawns subprocesses generally work only on Unix.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

If threads are faster and more portable, why would anyone write code that uses subprocesses? The main reason is that it's easy for one thread to stall all the others by tying up an entire process with an uninterruptible action. One such action is a system call. If you want to run a system call or an external program in the background, you should probably fork off a subprocess to do it. See Recipe 16.18 for a vivid example of this—a program that we need to spawn a subprocess instead of a subthread, because the subprocess is going to play a music file.

Recipe 20.1. Running a Daemon Process on Unix

Problem

You want to run a process in the background with minimal interference from users and the operating system.

Solution

In Ruby 1.9, you can simply call `Process.daemon` to turn the current process into a daemon. Otherwise, the most reliable way is to use the `Daemonize` module. It's not available as a gem, but it's worth downloading and installing, because it makes it easy and reliable to write a daemon:

```
#!/usr/bin/ruby -w
# daemonize_daemon.rb
require 'tempfile'
require 'daemonize'
include Daemonize            # Import Daemonize::daemonize into this namespace

puts 'About to daemonize.'
daemonize                    # Now you're a daemon process!
log = Tempfile.new('daemon.log')
loop do
  log.puts "I'm a daemon, doin' daemon things."
  log.flush
  sleep 5
end
```

If you run this code at the command line, you'll get back a new prompt almost immediately. But there will still be a Ruby process running in the background, writing to a temporary file every five seconds:

```
$ ./daemonize_daemon.rb
About to daemonize.
$ ps x | grep daemon
4472 ?         S       0:00 ruby daemonize_daemon.rb
4474 pts/2     S+      0:00 grep daemon

$ cat /tmp/daemon.log4472.0
I'm a daemon, doin' daemon things.
I'm a daemon, doin' daemon things.
I'm a daemon, doin' daemon things.
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Since it runs an infinite loop, this daemon process will run until you kill it:

```
$ kill 4472

$ ps x | grep daemon
 4569 pts/2    S+     0:00 grep daemon
```

A different daemon might run until some condition is met, or until it receives a Unix signal, or a "stop" message through some interface.

Discussion

A daemon process is one that runs in the background, without any direct user interface at all. Servers are usually daemon processes, but you might also write a daemon to do monitoring or task scheduling.

Rather than replacing your process with a daemon process, you may want to spawn a daemon while continuing with your original work. The best strategy for this is to spawn a subprocess with `Kernel#fork`.

Ruby's `fork` implementation takes a code block to be run by the subprocess. The code defined after the block is run in the original process. So pass your daemonizing code into `fork`, and continue with your work in the main body of the code:

```
#!/usr/bin/ruby -w
# daemon_spawn.rb
require 'tempfile'
require 'daemonize'
include Daemonize

puts "About to daemonize."
fork do
  daemonize
  log = Tempfile.new('daemon.log')
  loop do
    log.puts "I'm a daemon, doin' daemon things."
    log.flush
    sleep 5
  end
end

puts 'The subprocess has become a daemon.'
puts "But I'm going to stick around for a while."
sleep 10
puts "Okay, now I'm done."
```

The `Daemonize` code fits in a single file, and it's licensed under the same terms as Ruby. If you don't want to require your users to download and install it, you can just include it with your program. Because the code is short, you can even copy-and-paste the code into a file in your own program.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

However, there's also some (less fancy) daemonizing code in the Ruby 1.8 standard library. It's the `WEBrick::Daemon` class.

```
#!/usr/bin/ruby
# webrick_daemon.rb
require 'tempfile'
require 'webrick'

puts 'About to daemonize.'
WEBrick::Daemon.start do
  log = Tempfile.new('daemon.log')
  loop do
    log.puts "I'm a daemon, doin' daemon things."
    log.flush
    sleep 5
  end
end
```

It's worth examining the simpler daemonizing code in `WEBrick::Daemon` so that you can see what's going on. Here's the method in question:

```
def Daemon.start
  exit!(0) if fork
  Process::setsid
  exit!(0) if fork
  Dir::chdir("/")
  File::umask(0)
  STDIN.reopen("/dev/null")
  STDOUT.reopen("/dev/null", "w")
  STDERR.reopen("/dev/null", "w")
  yield if block_given?
end
```

A daemonizer works by forking a new process, letting the original one die, and closing off some of the resources that were available to the original.

`Process::setsid` disconnects the daemon from the terminal that spawned it. This is why, when your process becomes a daemon process, you get your command line back immediately. We close the original standard input, output, and error and replace them with null streams. We set the working directory and file umask to sensible defaults, regardless of what the daemon inherited from the parent. Then we run the daemon code.

`Daemonize::daemonize` also sets up signal handlers, calls `srand` so that the daemon process has a new random number seed, and (optionally) closes any open file handles left around by the original process. It can also retry the `fork` if it fails because the operating system is running too many processes to create another one.

The `fork` method, and methods like `daemonize` that depend on it, are only available on Unix-like systems. On Windows, the `win32-process` extension provides Windows implementations of methods like `fork`. The `win32-process` implementation of `fork` isn't perfect, but it's there if you need it. For cross-platform code, we recommend you spawn a thread and run your daemon code in the thread.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- The `Daemonize` package (http://grub.ath.cx/daemonize/)
- If you want to run an Internet server, you might want to use `gserver` from Ruby's standard library; see Recipe 14.14, "Writing an Internet Server"
- A service is the Windows equivalent of a daemon process; see Recipe 20.2, "Creating a Windows Service"
- Recipe 20.3, "Doing Two Things at Once with Threads"
- Both `win32-process` and `win32-service` were written by Daniel J. Berger; you can download them from his `win32utils` project at http://rubyforge.org/projects/win32utils/
- Get `win32-process` from http://rubyforge.org/projects/win32utils/

Recipe 20.2. Creating a Windows Service

Credit: Bill Froelich

Problem

You want to write a self-contained Ruby program for Windows that performs a task in the background.

Solution

Create a Windows service using the `win32-service` library, available as the `win32-service` gem.

Put all the service code below into a Ruby file called *rubysvc.rb*. It defines a service that watches for the creation of a file `c:\findme.txt`; if it ever finds that file, it immediately renames it.

The first step is to register the service with Windows. Running `ruby rubysrvc.rb register` will create the service.

```
# rubysrvc.rb
require 'rubygems'
require 'win32/service'
include Win32

SERVICE_NAME = "RubySvc"
SERVICE_DISPLAYNAME = "A Ruby Service"
if ARGV[0] == "register"
  # Start the service.
  svc = Service.new
  svc.create_service do |s|
    s.service_name = SERVICE_NAME
    s.display_name = SERVICE_DISPLAYNAME
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    s.binary_path_name = 'C:\InstantRails-1.3\ruby\bin\ruby ' +
      File.expand_path($0)
    s.dependencies = []
  end
  svc.close
  puts "Registered Service - " + SERVICE_DISPLAYNAME
```

When you're all done, you can run `rubysrvc.rb stop` to stop the service and remove it from Windows:

```
elsif ARGV[0] == "delete"
  # Stop the service.
  if Service.status(SERVICE_NAME).current_state == "running"
    Service.stop(SERVICE_NAME)
  end
  Service.delete(SERVICE_NAME)
  puts "Removed Service - " + SERVICE_DISPLAYNAME
else
```

If you run `rubysrvc.rb` with no arguments, nothing will happen, but it will remind you what parameters you can use:

```
if ENV["HOMEDRIVE"]!=nil
  # We are not running as a service, but the user didn't provide any
  # command line arguments. We've got nothing to do.
  puts "Usage: ruby rubysvc.rb [option]"
  puts "   Where option is one of the following:"
  puts "       register - To register the Service so it " +
       "appears in the control panel"
  puts "       delete   - To delete the Service from the control panel"
  exit
end
```

But when Windows runs `rubysrvc.rb` as a service, the real action starts:

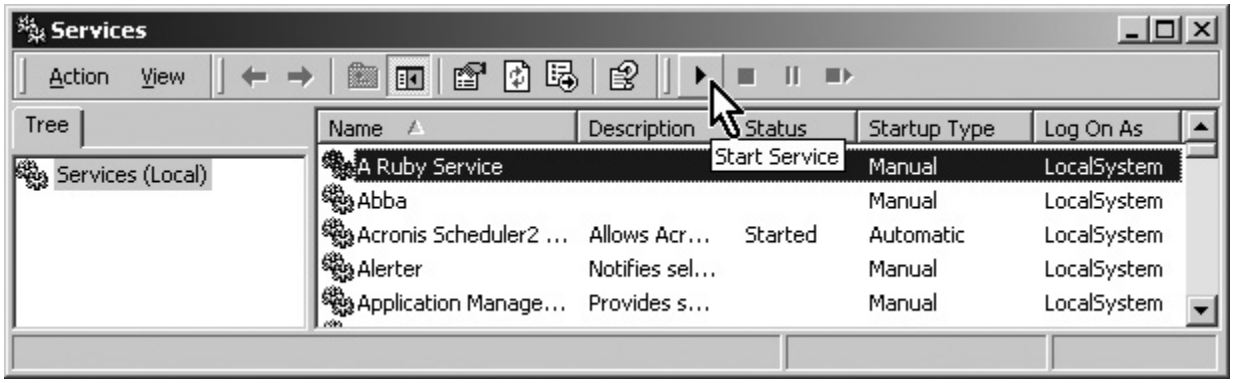
```
# If we got this far, we are running as a service.
class Daemon
  def service_init
    # Give the service time to get everything initialized and running,
    # before we enter the service_main function.
    sleep 10
  end

  def service_main
    fileCount = 0 # Initialize the file counter for the rename
    watchForFile = "c:\\findme.txt"
    while state == RUNNING
      sleep 5
      if File.exists? watchForFile
          fileCount += 1
          File.rename watchForFile, watchForFile + "." + fileCount.to_s
        end
      end
    end
  end
  d = Daemon.new
  d.mainloop
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Once you run `ruby rubysrvc.rb register`, the service will show up in the Services Control Panel as "A Ruby Service". To see it, go to Start ⟶ ControlPanel ⟶ Administrative Tools ⟶ Services (Figure 20-1). Start the service by clicking the service name in the list and clicking the start button.

Figure 20-1. The Services Control Panel



To test the service, create a file in `c:\` called *findme.txt*.

```
$ echo "test" > findme.txt
```

Within seconds, the file you just created will be renamed to *findme.txt*:

```
$ dir findme*
# Volume in drive C has no label.
# Volume Serial Number is 7C61-E72E
# Directory of c:\
# 04/14/2006  02:29 PM               9 findme.txt.1
```

To remove the service, run `ruby rubysrvc.rb` delete.

Discussion

There's no reason why the code that registers `rubysrvc.rb` as a Windows service has to be in `rubysrvc.rb` itself, but it makes things much simpler. When you run `ruby rubysrvc.rb register`, the script tells Windows to run `rubysrvc.rb` again, only as a service. The key is the `binary_path_name` defined on the `Service` object: this is the command for Windows to run as a service. In this case, it's an invocation of the `ruby` interpreter with the service script passed as an input. But you could have run the same code from an `irb` session: then, `rubysrvc.rb` would only have been invoked once, by Windows, when running it as a service.

The code above assumes that your Ruby interpreter is located in `c:InstantRails-1.3\ruby\bin\ruby`. Of course, you can change this to point to

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

your Ruby interpreter if it's somewhere else: perhaps `c:\ruby\bin\ruby`. If you've got the Ruby interpreter in your path, you just do this:

```
s.binary_path_name = 'ruby ' + File.expand_path($0)
```

When you create a service, you specify both a service name and a display name. The service name is shorter, and is used when referring to the service from within Ruby code. The display name is the one shown in the Services Control Panel.

Our example service checks every five seconds for a file with a certain name. Whenever it finds that file, it renames it by appending a number to the filename. To keep things simple, it does no error checking to see if the new filename already exists; nor does it do any file locking to ensure that the file is completely written before renaming it. Real services should include at least some basic high-level error handling:

```
def service_main
 begin
  while state == RUNNING
    # Do my work
  end
  # Finish my work
  rescue StandardError, Interrupt => e
    # Handle the error
  end
end
```

In addition to the `service_main` method, your service can define additional methods to handle the other service events (`stop`, `pause`, and `restart`). The `win32-service` gem comes with a useful example script, `daemon_test.rb`, which provides sample implementations of these methods.

See Also

- The win32-service library was written by Daniel J. Berger, and is part of the win32utils project (http://rubyforge.org/projects/win32utils/)
- Recipe 6.13, "Locking a File," and Recipe 6.14, "Backing Up to Versioned Filenames," demonstrate more robust renaming and file locking strategies
- Recipe 20.1, "Running a Daemon Process on Unix," for similar functionality on Unix
- Recipe 23.2, "Managing Windows Services"

Recipe 20.3. Doing Two Things at Once with Threads

Problem

You want your program to run two or more pieces of code in parallel.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

Create a new thread by passing a code block into `Thread.new`. That block will run simultaneously with any code you write after the call to `Thread.new`.

The following code features two competing threads. One continually decrements a variable by one, while the main program's thread busily incrementing the same variable by three. The decrementing thread starts its work earlier, but the incrementing thread always wins in the end, because it increments the counter by a larger number:

```
x = 0
Thread.new do
  while x < 5
    x -= 1
    puts "DEC: I decremented x to #{x}\n"
  end
  puts "DEC: x is too high; I give up!\n"
end

while x < 5
  x += 3
  puts "INC: I incremented x to #{x}\n"
end
# DEC: I decremented x to -1
# DEC: I decremented x to -2
# DEC: I decremented x to -3
# DEC: I decremented x to -4
# INC: I incremented x to -1
# DEC: I decremented x to -2
# INC: I incremented x to 1
# DEC: I decremented x to 0
# INC: I incremented x to 3
# DEC: I decremented x to 2
# INC: I incremented x to 5
# DEC: x is too high; I give up!

x                               # => 5
```

Discussion

A Ruby process starts out running only one thread: the main thread. When you call `Thread#new`, Ruby spawns another thread and starts running it alongside the main thread. The operating system divides CPU time among all the running processes, and the Ruby interpreter further divides its alotted CPU time among all of its threads.

The block you pass into `Thread.new` is a closure (see Recipe 7.4), so it has access to all the variables that were in scope at the time you instantiated the thread. This means that threads can share variables; as a result, you don't need complex communication schemes the way you do to communicate between processes. However, it also means that your threads can step on each other's toes unless you're careful to synchronize any shared objects. In the example above, the threads were *designed* to step on each other's toes, providing head-to-head competition, but usually you don't want that.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Once a thread's execution reaches the end of its code block, the thread dies. If your main thread reaches the end of *its* code block, the process will exit and all your other threads will die prematurely. If you want your main thread to stall and wait for some other thread to finish, you can call `Thread#join` on the thread in question.

This code spawns a subthread to count to one million. Without the call to `Thread#join`, the counter only gets up to a couple hundred thousand before the process exits:

```
#!/usr/bin/ruby -w
# counter_thread.rb
counter = 0
counter_thread = Thread.new do
  1.upto(1000000) { counter += 1; }
end

counter_thread.join unless ARGV[0]
puts "The counter was able to count up to #{counter}."
$ ./counter_thread.rb
The counter was able to count up to 1000000.

$ ./counter_thread.rb dont_call_join
The counter was able to count up to 172315.
```

You can get a list of the currently active thread objects with `Thread.list`:

```
Thread.new { sleep 10 }
Thread.new { x = 0; 10000000.times { x += 1 } }
Thread.new { sleep 100 }
Thread.list
# => [#<Thread:0xb7d19ae0 sleep>, #<Thread:0xb7d24cec run>,
#         #<Thread:0xb7d31cf8 sleep>, #<Thread:0xb7d68748 run>]
```

Here, the two running threads are the main `irb` thread and the thread running the counter loop. The two sleeping threads are the ones currently running `sleep` calls.

Recipe 20.4. Synchronizing Access to an Object

Problem

You want to make an object accessible from only one thread at a time.

Solution

Give the object a `Mutex` member (a semaphore that controls whose turn it is to use the object). You can then use this to synchronize activity on the object.

This code gives every object a `synchronize` method. This simulates the behavior of Java, in which `synchronize` is a keyword that can be applied to any object:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'thread'
class Object
  def synchronize
    mutex.synchronize { yield self }
  end

  def mutex
    @mutex ||= Mutex.new
  end
end
```

Here's an example. The first thread gets a lock on the list and then dawdles for a while. The second thread is ready from the start to add to the list, but it doesn't get a chance until the first thread releases the lock.

```
list = []
Thread.new { list.synchronize { |l| sleep(5); 3.times { l.push "Thread 1" } } }
Thread.new { list.synchronize { |l| 3.times { l.push "Thread 2" } } }
sleep(6)
list
# => ["Thread 1", "Thread 1", "Thread 1", "Thread 2", "Thread 2", "Thread 2"]
```

`Object#synchronize` only prevents two synchronized code blocks from running at the same time. Nothing prevents a wayward thread from modifying the object without calling `synchronize` first:

```
list = []
Thread.new { list.synchronize { |l| sleep(5); 3.times { l.push "Thread 1" } } }
Thread.new { 3.times { list.push "Thread 2" } }
sleep(6)
list
# => ["Thread 2", "Thread 2", "Thread 2", "Thread 1", "Thread 1", "Thread 1"]
```

Discussion

One of the big advantages of multithreaded programs is that different threads can share data. But where there is data sharing, there is the possibility for corruption. When two threads operate on the same object at the same time, the results can vary wildly depending on when the Ruby interpreter decides to switch between threads. To get predictable behavior, you need to have one thread lock the object, so other threads can't use it.

When every object has a `synchronize` method, it's easier to share an object between threads: if you want to work alone with the object, you put that code within a `synchronize` block. Of course, you may find yourself constantly writing synchronization code whenever you call certain methods of an object.

It would be nice if you could to do this synchronization implicitly, the way you can in Java: you just designate certain methods as "synchronized," and the interpreter won't start running those methods until it can obtain an exclusive lock on the corresponding object.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The simplest way to do this is to use aspect-oriented programming. The RAspect library described in Recipe 10.15 can be used for this.

The following code defines an `Aspect` that can wrap methods in synchronization code. It uses the `Object#mutex` method defined above, but it could easily be changed to define its own `Mutex` objects:

```
require 'aspectr'
require 'thread'

class Synchronized < AspectR::Aspect
  def lock(method_sym, object, return_value, *args)
    object.mutex.lock
  end

  def unlock(method_sym, object, return_value, *args)
    object.mutex.unlock
  end
end
```

Any AspectR aspect method needs to take three arguments: the symbol of the method being called, the object it's being called on, and (if the aspect method is being called after the original method) the return value of the method.

The rest of the arguments are the arguments to the original method. Since this aspect is very simple, the only argument we need is `object`, the object we're going to lock and unlock.

Let's use the `Synchronized` aspect to create an array where you can only call `push`, `pop`, or `each` once you get an exclusive lock.

```
array = %w{do re mi fa so la ti}
Synchronized.new.wrap(array, :lock, :unlock, :push, :pop, :each)
```

The call to `wrap` tells AspectR to modify our array's implementation of `push`, `pop`, and `each` with generated singleton methods. `Synchronized#lock` is called before the old implementation of those methods is run, and `Synchronized#unlock` is called afterward.

The following example creates two threads to work on our synchronized array. The first thread iterates over the array, and the second thread destroys its contents with repeated calls to `pop`. When the first thread calls `each`, the AspectR-generated code calls `lock`, and the first thread gets a lock on the array. The second thread starts and it wants to call `pop`, but `pop` has been modified to require an exclusive lock on the array. The second thread can't run until the first thread finishes its call to `each`, and the AspectR-generated code calls `unlock`.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
Thread.new { array.each { |x| puts x } }
Thread.new do
  puts 'Destroying the array.'
  array.pop until array.empty?
  puts 'Destroyed!'
end
# do
# re
# mi
# fa
# so
# la
# ti
# Destroying the array.
# Destroyed!
```

See Also

- See Recipe 10.15, "Doing Aspect-Oriented Programming," especially for information on problems with AspectR when wrapping operator methods in aspects
- Recipe 13.17, "Adding Hooks to Table Events," demonstrates the aspect oriented programming features of the Glue library, which are simpler than AspectR (but actually, in my experience, more difficult to use)
- Recipe 16.10, "Sharing a Hash Between Any Number of Computers," has an alternate solution: it defines a delegate class (`ThreadsafeHash`) whose `method_missing` implementation synchronizes on a mutex and then delegates the method call; this is an easy way to synchronize *all* of an object's methods
- Recipe 20.11, "Avoiding Deadlock"

Recipe 20.5. Terminating a Thread

Problem

You want to kill a thread before the end of the program.

Solution

A thread terminates if it reaches the end of its code block. The best way to terminate a thread early is to convince it to reach the end of its code block. This way, the thread can run cleanup code before dying.

This thread runs a loop while the instance variable `continue` is true. Set this variable to false, and the thread will die a natural death:

```
require 'thread'

class CounterThread < Thread
  def initialize
    @count = 0
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    @continue = true

    super do
      @count += 1 while @continue
      puts "I counted up to #{@count} before I was cruelly stopped."
    end
  end

  def stop
    @continue = false
  end
end

counter = CounterThread.new
sleep 2
counter.stop
# I counted up to 3413544 before I was cruelly stopped.
```

If you need to stop a thread that doesn't offer a `stop`-like function, or you need to stop an out-of-control thread *immediately*, you can always call `Thread#terminate`. This method stops a thread in its tracks:

```
t = Thread.new { loop { puts 'I am the unstoppable thread!' } }
# I am the unstoppable thread!
# I am the unstoppable thread!
# I am the unstoppable thread!
# I am the unstoppable thread!
t.terminate
```

Discussion

It's better to convince someone they should do something than to force them to do it. The same is `true` of threads. Calling `Thread.terminate` is a bit like throwing an exception: it interrupts the normal flow of execution in an unpredictable place. Worse, there's no equivalent of a `begin/ensure` construct for thread termination, so calling `Thread.terminate` may corrupt your data or leave your program in an inconsistent state. If you plan to stop a thread before the program is over, you should build that capability into the thread object itself.

A common type of thread implements a loop: threads that process requests from a queue, or that periodically poll for new data. In these, the end of an iteration forms a natural stopping point. These threads can benefit from some simple VCR-style controls: pause, unpause, and stop.

Here's a `Thread` subclass which implements a loop that can be paused or stopped in a predictable way. A code block passed into the `Thread` constructor would implement the entire loop, but the code block passed into the `LoopingThread` constructor should implement only one iteration of the loop. Setup and cleanup code should be handled in the methods `before_loop` and `after_loop`.

```
class LoopingThread < Thread
  def initialize
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    @stopped = false
    @paused = false
    super do
      before_loop
      until @stopped
        yield
        Thread.stop if @paused
      end
      after_loop
    end
  end

  def before_loop; end
  def after_loop; end

  def stop
    @stopped = true
  end

  def paused=(paused)
    @paused = paused
    run if !paused
  end
end
```

Here's the `CounterThread` class from the Solution, implemented as a `LoopingThread`. I've added a reader method for `count` so we can peek at its value when the thread is paused:

```
class PausableCounter < LoopingThread
  attr_reader :count

  def before_loop
    @count = 0
  end

  def initialize
    super { @count += 1 }
  end

  def after_loop
    puts "I counted up to #{@count} before I was cruelly stopped."
  end
end

counter = PausableCounter.new
sleep 2
counter.paused = true
counter.count                                          # => 819438
sleep 2
counter.count                                          # => 819438
counter.paused = false
sleep 2
counter.stop
# I counted up to 1644324 before I was cruelly stopped.
counter.count                                          # => 1644324
```

Recipe 20.6. Running a Code Block on Many Objects Simultaneously

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

Rather than iterating over the elements of a data structure one at a time, you want to run some function on all of them simultaneously.

Solution

Spawn a thread to handle each element of the data structure.

Here's a simple equivalent of `Enumerable#each` that runs a code block against every element of a data structure simultaneously.[1] It returns the `Thread` objects it spawned so that you can pause them, kill them, or `join` them and wait for them to finish:

[1] Well, more or less. The thread for the first element will start running before the thread for the last element does.

```
module Enumerable
  def each_simultaneously
    threads = []
    each { |e| threads >> Thread.new { yield e } }
    return threads
  end
end
```

Running the following high-latency code with `Enumerable#each` would take 15 seconds. With our new `Enumerable#each_simultaneously`, it takes only five seconds:

```
start_time = Time.now
[7,8,9].each_simultaneously do |e|
   sleep(5) # Simulate a long, high-latency operation
   print "Completed operation for #{e}!\n"
end
# Completed operation for 8!
# Completed operation for 7!
# Completed operation for 9!
Time.now - start_time                # => 5.009334
```

Discussion

You can save time by doing high-latency operations in parallel, since it often means you pay the latency price only once. If you're doing nameserver lookups, and the nameserver takes five seconds to respond to a request, you're going to be waiting at least five seconds. If you need to do 10 nameserver lookups, doing them in series will take 50 seconds, but doing them all at once might only take 5.

This technique can also be applied to the other methods of `Enumerable`. You could write a `collect_simultaneously`, a `find_all_simultaneously`, and so on. But that's a lot of methods to write. All the methods of `Enumerable` are based on `each`. What if we could just convince those methods to use `each_simultaneously` instead of each?

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

It would be too much work to replace all the existing methods of `Enumerable`, but we can swap out an individual `Enumerable` object's `each` implementation for another, by wrapping it in an `Enumerable::Enumerator`. Here's how it would work:

```
require 'enumerator'

array = [7, 8, 9]
simultaneous_array = array.enum_for(:each_simultaneously)
simultaneous_array.each do |e|
  sleep(5) # Simulate a long, high-latency operation
  print "Completed operation for #{e}!\n"
end
# Completed operation for 7!
# Completed operation for 9!
# Completed operation for 8!
```

That call to `enum_for` returns an `Enumerable::Enumerator` object. The `Enumerator` implements all of the methods of `Enumerable` as the original array would, but its `each` method uses `each_simultaneously` under the covers.

Do we now have simultaneous versions of all the `Enumerable` methods? Not quite. Look at this code:

```
simultaneous_array.collect { |x| sleep 5; x * -1 }             # => []
```

What happened? The collect method returns before the threads have a chance to complete their tasks. When we were using `each_simultaneously` on its own, this was a nice feature. Consider the following idealized code, which starts three infinite loops in separate threads and then goes on to other things:

```
[SSHServer, HTTPServer, IRCServer].each_simultaneously do |server|
  server.serve_forever
end

# More code goes here…
```

This is not such a good feature when we're calling an `Enumerable` method with a return value. We need an equivalent of `each_simultaneously` that doesn't return until all of the threads have run:

```
require 'enumerator'
module Enumerable
  def all_simultaneously
    if block_given?
      collect { |e| Thread.new { yield(e) } }.each { |t| t.join }
      self
    else
      enum_for :all_simultaneously
    end
  end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

You wouldn't use this method to spawn infinite loops (they'd all spawn, but you'd never regain control of your code). But you can use it to create multithreaded versions of `collect` and other `Enumerable` methods:

```
array.all_simultaneously.collect { |x| sleep 5; x * -1 }
# => [-7, -9, -8]
```

That's better, but the elements are in the wrong order: after all, there's no guarantee which thread will complete first. This doesn't usually matter for `Enumerable` methods like `find_all, grep`, or `reject`, but it matters a lot for `collect`. And `each_with_index` is simply broken:

```
array.all_simultaneously.each_with_index { |x, i| sleep 5; puts "#{i}=>#{x}" }
# 0=>8
# 0=>7
# 0=>9
```

Here are thread-agnostic implementations of `Enumerable#collect` and `Enumerable#each_with_index`, which will work on normal `Enumerable` objects, but will also work in conjunction with `all_simultaneously`:

```
module Enumerable
  def collect
    results = []
    each_with_index { |e, i| results[i] = yield(e) }
    results
  end

  def each_with_index
    i = -1
    each { |e| yield e, i += 1 }
  end
end
```

Now it all works:

```
array.all_simultaneously.collect { |x| sleep 5; x * -1 }
# => [-7, -8, -9]

array.all_simultaneously.each_with_index { |x, i| sleep 5; puts "#{i}=>#{x}" }
# 1=>8
# 0=>7
# 2=>9
```

See Also

- Recipe 7.9, "Looping Through Multiple Iterables in Parallel"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 20.7. Limiting Multithreading with a Thread Pool

Problem

You want to process multiple requests in parallel, but you don't necessarily want to run all the requests simultaneously. Using a technique like that in Recipe 20.6 can create a huge number of threads running at once, slowing down the average response time. You want to set a limit on the number of simultaneously running threads.

Solution

You want a thread pool. If you're writing an Internet server and you want to service requests in parallel, you should build your code on top of the `gserver` module, as seen in Recipe 14.14: it has a thread pool and many TCP/IP-specific features. Otherwise, here's a generic `ThreadPool` class, based on code from `gserver`.

The instance variable `@pool` contains the active threads. The `Mutex` and the `ConditionVariable` are used to control the addition of threads to the pool, so that the pool never contains more than `@max_size` threads:

```
require 'thread'

class ThreadPool
  def initialize(max_size)
    @pool = []
    @max_size = max_size
    @pool_mutex = Mutex.new
    @pool_cv = ConditionVariable.new
end
```

When a thread wants to enter the pool, but the pool is full, the thread puts itself to sleep by calling `ConditionVariable#wait`. When a thread in the pool finishes executing, it removes itself from the pool and calls `ConditionVariable#signal` to wake up the first sleeping thread:

```
def dispatch(*args)
  Thread.new do
    # Wait for space in the pool.
    @pool_mutex.synchronize do
      while @pool.size >= @max_size
        print "Pool is full; waiting to run #{args.join(',')}…\n" if $DEBUG
        # Sleep until some other thread calls @pool_cv.signal.
        @pool_cv.wait(@pool_mutex)
      end
    end
```

The newly-awakened thread adds itself to the pool, runs its code, and then calls `ConditionVariable#signal` to wake up the *next* sleeping thread:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
      @pool << Thread.current
      begin
        yield(*args)
      rescue => e
        exception(self, e, *args)
      ensure
        @pool_mutex.synchronize do
          # Remove the thread from the pool.
          @pool.delete(Thread.current)
          # Signal the next waiting thread that there's a space in the pool.
          @pool_cv.signal
        end
      end
    end
  end

  def shutdown
    @pool_mutex.synchronize { @pool_cv.wait(@pool_mutex) until @pool.empty? }
  end

  def exception(thread, exception, *original_args)
    # Subclass this method to handle an exception within a thread.
    puts "Exception in thread #{thread}: #{exception}"
  end
end
```

Here's a simulation of five incoming jobs that take different times to run. The pool ensures no more than three jobs run at a time. The job code doesn't need to know anything about threads or thread pools; that's all handled by `ThreadPool#dispatch`.

```
$DEBUG = true
pool = ThreadPool.new(3)

1.upto(5) do |i|
  pool.dispatch(i) do |i|
    print "Job #{i} started.\n"
    sleep(5-i)
    print "Job #{i} complete.\n"
  end
end
# Job 1 started.
# Job 3 started.
# Job 2 started.
# Pool is full; waiting to run 4…
# Pool is full; waiting to run 5…
# Job 3 complete.
# Job 4 started.
# Job 2 complete.
# Job 5 started.
# Job 5 complete.
# Job 4 complete.
# Job 1 complete.

pool.shutdown
```

Discussion

When should you use a thread pool, and when should you just send a swarm of threads after the problem? Consider why this pattern is so common in Internet servers that it's built into Ruby's `gserver` library. Internet server requests are usually I/O bound, because most servers operate on the filesystem or a database. If you run high latency requests in

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

parallel (like requests for filesystem files), you can complete multiple requests in about the same time it would take to complete a single request.

But Internet server requests can use a lot of memory, and any random user on the Internet can trigger a job on your server. If you create and start a thread for every incoming request, it's easy to run out of resources. You need to find a tradeoff between the performance benefit of multithreading and the performance hazard of thrashing due to insufficient resources. The simplest way to do this is to limit the number of requests that can be processed at a given time.

A thread pool isn't a connection pool, like you might see with a database. Database connections are often pooled because they're expensive to create. Threads are pretty cheap; we just don't want a lot of them actively running at once. The example in the Solution creates five threads at once, but only three of them can be active at any one time. The rest are asleep, waiting for a notification from the condition variable `pool_cv`.

Calling `ThreadPool#dispatch` with a code block creates a new thread that runs the code block, but not until it finds a free slot in the thread pool. Until then, it's waiting on the condition variable `@pool_cv`. When one of the threads in the pool completes its code block, it calls `signal` on the condition variable, waking up the first thread currently waiting on it.

The `shutdown` method makes sure all the jobs complete by repeatedly waiting on the condition variable until no other threads want access to the pool.

See Also

- Recipe 14.14, "Writing an Internet Server"

Recipe 20.8. Driving an External Process with popen

Problem

You want to execute an external command in a subprocess. You want to pass some data into its standard input stream, and read its standard output.

Solution

If you don't care about the standard input side of things, you can just use the %x{} construction. This runs a string as a command in an operating system subshell, and returns the standard output of the command as a string.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
%x{whoami}                                          # => "leonardr\n"
puts %x{ls -a empty_dir}
# .
# ..
```

If you want to pass data into the standard input of the subprocess, do it in a code block that you pass into the `IO.popen` method. Here's `IO.popen` used on a Unix system to invoke `tail`, a command that prints to standard output the last few lines of its standard input:

```
IO.popen('tail -3', 'r+') do |pipe|
  1.upto(100) { |i| pipe >> "This is line #{i}.\n" }
  pipe.close_write
  puts pipe.read
end
# This is line 98.
# This is line 99.
# This is line 100.
```

Discussion

`IO.popens` pawns a subprocess and creates a pipe: an `IO` stream connecting the Ruby interpreter to the subprocess. `IO.popen` makes the pipe available to a code block, just as `File.open` makes an open file available to a code block. Writing to the `IO` object sends data to the standard input of the subprocess; reading from it reads data from its standard output.

`IO.popen` takes a file mode, just like `File.open`. To use both the standard input and output of a subprocess, you need to open it in read-write mode ("`r+`").

A command that accepts standard input won't really start running until its input stream is closed. If you use `popen` to run a command like `tail`, you must call `pipe.close_write` before you read from the pipe. If you try to read the subprocess' standard output while the subprocess is waiting for you to send it data on standard input, both processes will hang forever.

The %{} construct and the `popen` technique work on both Windows and Unix, but scripts that use them won't usually be portable, because it's very unlikely that the command you're running exists on all platforms.

On Unix systems, you can also use `popen` to spawn a Ruby subprocess. This is like calling `fork`, except that the parent gets a read-write filehandle that's hooked up to the standard input and output of the child. Unlike with `Kernel#fork` (but like C's implementation of `fork`), the same code block is called for the parent and the child. The presence or absence of the filehandle is the only way to know whether you're the parent or the child:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
IO.popen('-', 'r+') do |child_filehandle|
  if child_filehandle
    $stderr.puts "I am the parent: #{child_filehandle.inspect}"
    child_filehandle.puts '404'
    child_filehandle.close_write
    puts "My child says the square root of 404 is #{child_filehandle.read}"
  else
    $stderr.puts "I am the child: #{child_filehandle.inspect}"
    number = $stdin.readline.strip.to_i
    $stdout.puts Math.sqrt(number)
  end
end
# I am the child: nil
# I am the parent: #<IO:0xb7d25b9c>
# My child says the square root of 404 is 20.0997512422418
```

See Also

- Recipe 20.1, "Running a Daemon Process on Unix"
- Recipe 20.9, "Capturing the Output and Error Streams from a Unix Shell Command"
- Recipe 20.10, "Controlling a Process on Another Machine"

Recipe 20.9. Capturing the Output and Error Streams from a Unix Shell Command

Problem

You want to run an external program as in Recipe 20.8, but you also want to capture the standard error stream. Using `popen` only gives you access to the standard output.

Solution

Use the `open3` library in the Ruby standard library. Its `popen3` method takes a code block, to which it passes three `IO` streams: one each for standard input, output, and error.

Suppose you perform the Unix `ls` command to list a nonexistent directory. `ls` will rightly object to this and write an error message to its standard error stream. If you invoked `ls` with `IO.popen` or the %x{} construction, that error message is passed right along to the standard error stream of your Ruby process. You can't capture it or suppress it:

```
%x{ls no_such_directory}
# ls: no_such_directory: No such file or directory
```

But if you use `popen3`, you can grab that error message and do whatever you want with it:

```
require 'open3'
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
Open3.popen3('ls -l no_such_directory') { |stdin, stdout, stderr| stderr.read }
# => "ls: no_such_directory: No such file or directory\n"
```

Discussion

The same caveats in the previous recipe apply to the IO streams returned by `popen3`. If you're running a command that accepts data on standard input, and you read from `stdout` before closing `stdin`, your process will hang.

Unlike `IO.popen`, the `popen3` method is only implemented on Unix systems. However, the win32-open3 package (part of the Win32Utils project) provides a `popen3` implementation.

See Also

- Recipe 20.8, "Driving an External Process with popen"
- Like many other Windows libraries for Ruby, `win32-open3` is available from http://rubyforge.org/projects/win32utils

Recipe 20.10. Controlling a Process on Another Machine

Problem

You want to run a process on another machine, controlling its input stream remotely, and reading its output and error streams.

Solution

The `ruby-ssh` gem, first described in Recipe 14.10, provides a `popen3` method that works a lot like Ruby's built-in `popen3`, except that the process you spawn runs on another computer.

Here's a method that runs a Unix command on another computer and yields its standard I/O streams to a code block on your computer. All traffic going between the computers is encrypted with SSL. To authenticate yourself against the foreign host, you'll either need to provide a username and password, or set up an SSL key pair ahead of time.

```
require 'rubygems'
require 'net/ssh'

def run_remotely(command, host, args)
  Net::SSH.start(host, args) do |session|
    session.process.popen3(command) do |stdin, stdout, stderr|
      yield stdin, stdout, stderr
    end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  end
end
```

Here it is in action:

```
run_remotely('ls -l /home/leonardr/dir', 'example.com', :username=>'leonardr',
             :password => 'mypass') { |i, o, e| puts o.read }
# -rw-rw-r-- 1   leonardr leonardr      33 Dec 29 20:40 file1
# -rw-rw-r-- 1   leonardr leonardr     102 Dec 29 20:40 file2
```

Discussion

The `Net::SSH` library implements a low-level interface to the SSH protocol, but most of the time you don't need all that power. You just want to use SSH as a way to spawn and control processes on a remote computer. That's why `Net:SSH` also provides a `popen3` interface that looks a lot like the `popen3` you use to manipulate processes on your own computer.

Apart from the issue of authentication, there are a couple of differences between `Net::SSH.popen3` and `Open3.popen3`. With `Open3.popen3`, you must be careful to close the standard input stream before reading from the output or error streams. With the `Net::SSH` version of `popen3`, you can read from the output or error streams as soon as the process writes any data to it. This lets you interleave `stdin` writes and `stdout` reads:

```
run_remotely('cat', 'example.com', :username=>'leonardr',
             :password => 'mypass') do |stdin, stdout, stderr|
  stdin.puts 'Line one.'
  puts stdout.read
  stdin.puts 'Line two.'
  puts stdout.read
end
# "Line one."
# "Line two."
```

Another potential pitfall is that the initial working directory for an SSH session is the filesystem root (/). If you've used the `ssh` or `scp` commands, you may be accustomed to starting out in your home directory. To compensate for this, you can change to your home directory within your command: issue a command like `cd; ls` or `cd /home/[user name]/; ls` instead of just plain `ls`.

See Also

- The `Net::SSH` manual at: http://net-ssh.rubyforge.org/
- Recipe 14.2, "Making an HTTPS Web Request," has information on installing the OpenSSL extension that is a prerequisite of `ruby-ssh`
- Recipe 14.10, "Being an SSH Client covers the basic rules of SSH"

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

- Recipe 20.8, "Driving an External Process with popen," and Recipe 20.9, "Capturing the Output and Error Streams from a Unix Shell Command," cover the basic features of the `popen` family of methods

Recipe 20.11. Avoiding Deadlock

Problem

Your threads are competing for exclusive access to the same resources. With no coordination between threads, you'll end up with deadlock. Thread A will be blocking, waiting for a resource held by thread B, and thread B will be blocking, waiting for a resource held by thread A. Neither thread will ever be seen again.

Solution

There's no simple mix-in solution to this problem. You need to come up with some rules for how your threads acquire locks, and make sure your code always abides by them.

Basically, you need to guarantee that all your threads acquire locks in the same order. Impose an ordering (formally or informally) on all the locks in your program and make sure that your threads always acquire locks in ascending numerical order.

Here's how it would work. The standard illustration of deadlock is the Dining Philosophers problem. A table of philosophers are sharing a plate of rice and some chopsticks, but there aren't enough utensils to go around. When there are only two chopsticks, it's easy to see the problem. If philosopher A is holding one chopstick (that is, has a lock on it), and philosopher B is holding the other, then nobody can eat.

In this scenario, you'd designate the the lock on one chopstick as lock #1, and the lock on the other chopstick as lock #2. If you guarantee that no philosopher will pick up chopstick #2 unless they're already picked up the chopstick #1, deadlock is impossible. You can guarantee this by simply making all the philosophers implement the same behavior:

```
require 'thread'
$chopstick1 = Mutex.new
$chopstick2 = Mutex.new

class Philosopher < Thread
  def initialize(name)
    super do
      loop do
        $chopstick1.synchronize do
          puts "#{name} has picked up one chopstick."
          $chopstick2.synchronize do
            puts "#{name} has picked up two chopsticks and eaten a " +
                 "bite of tasty rice."
          end
        end
      end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    end
  end
end
Philosopher.new('Moore')
Philosopher.new('Anscombe')
# Moore has picked up one chopstick.
# Moore has picked up two chopsticks and eaten a bite of tasty rice.
# Anscombe has picked up one chopstick.
# Anscombe has picked up two chopsticks and eaten a bite of tasty rice.
# Moore has picked up one chopstick.
# Moore has picked up two chopsticks and eaten a bite of tasty rice.
# …
```

Discussion

It's hard to come up with an ordering of resources that isn't totally arbitrary. Why is chopstick #1 designated #1 and not #2? It just is. When you've got more than a few locks, it's hard to remember the order.

But if you keep a list of the locks in the proper order, you can have Ruby handle the locking order for you. The `lock_all` method defined below takes an unordered list of locks, and makes sure they get locked in the "right" order, as defined in the global hash `$lock_order`:

```
require 'thread'
pool_lock, lion_lock, penguin_lock, cabbage_lock = (1..4).collect { Mutex.new }
locks = [pool_lock, lion_lock, penguin_lock, cabbage_lock]
$lock_order = {}
locks.each_with_index { |lock, i| $lock_order[lock] = i }

def lock_all(*locks)
  ordered_locks = locks.sort_by { |x| $lock_order[x] }
  ordered_locks.each do |lock|
    puts "Locking #{$lock_order[lock]}." if $DEBUG
    lock.lock
  end
  begin
    yield
  ensure
    ordered_locks.reverse_each do |lock|
      puts "Unlocking #{$lock_order[lock]}." if $DEBUG
      lock.unlock
    end
  end
end
```

Now you can simply pass the locks you want to get into `lock_all`, without having to keep track of an arbitrary order:

```
$DEBUG = true
lock_all(penguin_lock, pool_lock) do
  puts "I'm putting the penguin in the pool."
end
# Locking 0.
# Locking 2.
# I'm putting the penguin in the pool.
# Unlocking 2.
# Unlocking 0.
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

When `lock_all` encounters a mutex that's already locked, the thread blocks until the mutex becomes available. A less greedy alternative is to drop all of the mutexes already obtained and try again from the start. This makes deadlock less likely even when not all of the code respects the order of the locks.

There are two locking-related problems that you can't solve by imposing a lock ordering. The first is resource starvation. In the context of the dining philosophers, this would mean that one philosopher continually puts down chopstick #1 and immediately takes it up again, preventing anyone else from eating.

The `thread` library prevents this problem by keeping a list of the threads that are waiting for a lock to be released. Once it's released, Ruby wakes up the first thread in line. So threads get the lock in the order they asked for it, rather than it being a free-for-all. You can see this if you create a bunch of `Philosopher` objects using the example from the Solution. Even if there are 20 philosophers and only one pair of chopsticks, the philosophers will take turns using the chopsticks in the order they were created, not randomly depending on the whims of the Ruby interpreter.

The second problem is harder to solve: a thread can "deadlock" with itself. The following code looks unobjectionable (why shouldn't you be able to lock what you already have?), but it creates a thread that sleeps forever:

```
require 'thread'
$lock = Mutex.new
Thread.new do
  $lock.synchronize { $lock.synchronize { puts 'I synchronized twice!' } }
end
```

The first time you call `lock.synchronize`, everything works fine: the `Mutex` isn't locked, and the thread gets a lock on it. The second time, the `Mutex` *is* locked, so the thread stops to wait until it gets unlocked.

The problem is, the thread B that's stopping to wait is the same thread as thread A, which has the lock. Thread A is supposed to wake up thread B once it's done, but it never does, because it *is* thread B, and it's asleep. A thread can't wake itself up.

That looks like a contrived example, but it's pretty easy to get there by accident. If you're synchronizing an object, as described in Recipe 20.4, there's a chance you'll go too far and synchronize two methods that call each other. Calling one method will synchronize and call the other, which will synchronize and put the thread to sleep forever. Short of hacking `Mutex` to keep track of which thread has the lock, the only way to avoid this problem is to be careful.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Recipe 6.13, "Locking a File," shows an alternate way of avoiding deadlock when the resource under contention is a file

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01