WORKING *with* RUBY THREADS

Jesse Storimer

This book is dedicated to Sara, Inara, and Ora, who make it all worthwhile.

Contents

Releases

- April 11, 2013 - First public release.
- April 12, 2013 - Fixed typos.
- June 10, 2013 - Big revision. New chapters on Puma, Sidekiq, atomic CAS operations, and immutability. New sections on memory visibility, deadlocks, and many small improvements.

Chapter 0

Introduction

My story

When I joined the Ruby community, I had little understanding of multi-threaded programming.

At that time, almost everyone was using MRI, the original implementation of Ruby. Its threading implementation used green threads. I didn't know what this meant, but people didn't seem to take it too seriously, so I didn't either. At that time, I didn't even know where to begin asking questions when it came to Ruby concurrency.

I had heard the term 'thread safety,' and I knew it was a generally bad thing, but that was the extent of my knowledge.

I still remember the moment that I actually became afraid of thread safety. It was when I read an article[1] which pointed out that the `||=` operator in Ruby is not thread-safe. Even though the application I worked on wasn't running in multiple threads, I was now paranoid about using this operator.

In spite of my trepidation, I just couldn't believe that `||=` was inherently unsafe. It was everywhere in our codebase! It was this trepidation, and the realization that there are

1. http://coderrr.wordpress.com/2009/04/29/is-not-thread-safe-neither-is-hashnew-hk/

fundamentals I knew nothing about, that led me to learn about processes, threads, networking, and all that fun stuff.

In this book, I'll help you through the same trepidation. I'll show you how, in certain contexts, the `||=` operator *can* be not thread-safe, but you certainly don't need to stop using it.

Why care?

If you're reading this book on any kind of modern computing device, it's likely that it has more than one CPU core. And chances are the device you buy next year will have more cores than the one you have today.

Unfortunately, just adding more CPU cores doesn't necessarily make all your code run faster. **Your code must be architected to take advantage of multiple CPU cores** using some concurrency mechanism. If you're not doing this, you might as well be running on ten-year-old hardware.

Operating system processes have been the de facto concurrency mechanism for decades. With processes, if you want more concurrency, then start more processes! This has been the norm in the Ruby community for many years.

So, why threads?

The promise of multi-threading

The promise of multi-threading has always been cheaper concurrency. Cheaper than processes, that is.

Spawning a thread incurs much less overhead than spawning a process.

The primary difference between using processes versus threads is the way that memory is handled. At a high level, processes copy memory, while threads share memory. This makes process spawning slower than thread spawning, and leads to processes consuming more resources once running. Overall, threads incur less overhead than processes.

This smaller overhead means **threads can you give more 'units' of concurrency for the available resources.** Of course, this comes with a cost: introducing multiple threads requires that your code be thread-safe.

Despite the Ruby community's love for process-based solutions, there is a shift happening. The default MRI implementation has always had, even to this day, a threading implementation that limits parallel execution. This stifled community support for a long time.

But more and more people are becoming aware of the promise of multi-threading. More opportunities, and more choices, are becoming available in the community, opportunities like higher throughput and more concurrency, while using less resources. Now's the time to educate yourself and take advantage of these opportunities.

What to expect

In this book I'm providing you with lots of small code samples intended to illustrate key concepts. Please run them in your own console and play with them. Taking the examples and then tweaking them to test your hypotheses is a fantastic way to learn. Look in the included `code/` folder for all of the snippets to avoid copy/paste issues.

The first part of the book focuses on basic concurrency-related topics. Where possible, I attempt to generalize, so that what you learn here can also be applied when you're programming with other languages or just pondering code in general.

That being said, the primary focus of the book is multi-threaded programming *in Ruby*.

The second part of the book dives deeper into what the Ruby language offers to support multi-threaded programming.

The book ends with a hands-on tutorial, where you walk through implementing a concurrent program using several approaches.

There are no clear dividing lines between these three parts, and there's certainly some overlap. **The goal of the book is to give you the necessary knowledge so that you can make good decisions about concurrency for your application.** This includes making you comfortable with the idea of multi-threaded programming, dispelling myths that may be floating around the community, and showing you what tools Ruby offers to aid you.

Which version of Ruby is used?

This book studies three different Ruby language implementations.

1. MRI, version 2.0.0 (all code is also 1.9.3 compatible).
2. JRuby, version 1.7.4.
3. Rubinius, version 2.0.0-rc1.

For JRuby and Rubinius, it's assumed that you're running them in 1.9 language mode. When relevant, I'll show sample code output from all three implementations.

The reason to include multiple Ruby implementations is that they have very different stories when it comes to multi-threading, which can lead to differences in behaviour.

Chapter 1

You're Always in a Thread

By default, your code is always running a thread. Even if you don't create any new threads, **there's always at least one: the main thread.**

You can use an `irb` session to demonstrate this.

```
$ irb
> Thread.main
=> #<Thread:0x007fdc830677c0 run>
> Thread.current == Thread.main
=> true
```

The main thread is the original thread that's created when you start a Ruby program. You can always create more threads, but you can't change the reference to `Thread.main`, it will always point to this original thread.

The main thread has one special property that's different from other threads. **When the main thread exits, all other threads are immediately terminated and the Ruby process exits.** This is not true of any other thread besides the main thread.

```
# ./code/snippets/sleep_main_thread.rb

thread = Thread.new do
  # do the important work
```

```
end

# The main thread sleeps to prevent it from finishing execution.
# If it were allowed to run, it would simply exit, killing the other
# thread and preventing it from doing its important work.
sleep
```

The `Thread.current` method always refers to the current thread. This may seem obvious, but you have to wrap your head around the fact that this one method will return a different `Thread` object depending on which thread it's called from.

This `irb` session spawns a new thread, then compares `Thread.current` with `Thread.main`. Notice that in this case the two are no longer equal.

```
$ irb
> Thread.new { Thread.current == Thread.main }.value
=> false
```

Chapter 2

Threads of Execution

Creating new threads is easy:

```
Thread.new { "Did it!" }
```

But multi-threaded programming is not as simple as just spawning more threads. There are lots of questions to be answered, such as:

- Where/when should threads be spawned?
- How many threads?
- What about thread safety?

Threads are a powerful concept, and like anything powerful, they also allow you to shoot yourself in the foot if you don't know what you're doing. Here I'll give you an overview of exactly what threads offer you.

Shared address space

The most important concept to grasp is that **threads have a shared address space.** This is a fancy way of saying that multiple threads will share all of the same references in memory (ie. variables), as well as the AST (ie. the compiled source code).

This simple fact is what makes threads so powerful, and also what makes them difficult to work with. I've already given you an idea of why threads are good; here's a simple program to illustrate their difficulty.

```
# ./code/snippets/file_uploader.rb

require 'thread'

class FileUploader
  def initialize(files)
    @files = files
  end

  def upload
    threads = []

    @files.each do |(filename, file_data)|
      threads << Thread.new {
        status = upload_to_s3(filename, file_data)
        results << status
      }
    end

    threads.each(&:join)
  end

  def results
    @results ||= Queue.new
  end
```

```
  def upload_to_s3(filename, file)
    # omitted
  end
end

uploader = FileUploader.new('boots.png' => '*pretend png data*', 'shirts.png' => '*pretend png data*')
uploader.upload

puts uploader.results.size
```

If you run your eye down this example program, it looks pretty innocent. The obvious thing that sticks out is that the `#upload` method spawns one thread for each file that it uploads to S3. Since a call to `Thread.new` returns immediately, both of those threads will be working concurrently.

The `#upload` method ends by calling `#join` on each of the threads, which will block until the thread finishes execution.

Unfortunately, this code is not thread-safe. **It's quite possible that one of the statuses in the @results array will be lost**. I intentionally omitted the S3 upload implementation details to let you know that the thread-safety issue does not come from there. The problem is actually with the `#results` method.

This might not seem possible; we're just using regular Ruby code here with a conditional assignment! But this operator can absolutely cause problems in a multi-threaded context. This chapter will help you understand why that's so.

I'll let this example sit with you for a minute, but I'll come back and explain why this happens before you finish this chapter.

Many threads of execution

We're used to looking at source code and seeing a sequential set of instructions. First, this method calls that one, then if this value is true, it executes this block of code, then... you get the idea. It's natural for you to think about your code this way. This is typically the way that we write it.

When you trace a path through your code like this, you are tracing a thread of execution, a possible path that can be traversed through your code. It's easy to grasp that there is more than one possible thread of execution. For example, if you pass in a different input, you may get a different output.

It's harder to grasp that there can be multiple threads of execution operating on the same code *at the same time*. This is precisely the case in a multi-threaded environment. Multiple threads of execution can be traversing their own paths, all at the same time.

At that point, it's no longer possible to step through these simultaneous threads of execution in any kind of predictable manner. **It's as if someone just changed the rules of physics on you: things that were previously absolute truths no longer hold true.** This is because there's a certain amount of randomness introduced in the way that threads are scheduled.

Thankfully, you are able to introduce some thread-aware guarantees into your code. This is the heart of thread safety. But let's talk a bit more about this randomness first.

Native threads

All of the Ruby implementations studied in this book ultimately map one Ruby thread to one native, operating system thread.

Take this silly example, run against MRI:

```
100.times do
  Thread.new { sleep }
end

puts Process.pid
sleep
```

This creates 100 sleeping threads, then sleeps the main thread to prevent it from exiting. Now we can ask top(1) how many threads this process has:

```
$ top -l1 -pid 8409 -stats pid,th

PID   #TH
8409  102
```

Note that you should use your own pid (process id) to test this out.

So we created 100 new threads, the main thread counts as 1, and MRI maintains an internal thread for housekeeping[1], which adds up to 102. The important point is that when we create 100 threads, those 100 threads are handled directly by the operating system.

Non-deterministic context switching

This section has a fancy title. It refers to the work that's done by your operating system thread scheduler. This is a part of your operating system itself, and you have no control over how it functions. This little ditty is responsible for scheduling *all* of the threads running on the system.

It has to ensure that if there are 453 threads running on the system, they all get fair access to the system resources. This is a very complicated piece of software, with many optimizations, but it all comes down to this: context switching.

In order to provide fair access, the thread scheduler can 'pause' a thread at any time, suspending its current state. Then it can unpause some other thread so it can have a turn using system resources. Then, at some point in the near future, the thread scheduler can unpause the original thread, restoring it to its previous state.

This is known as context switching, and there's a certain degree of randomness to it. It's possible for your thread to be interrupted *at any time*. Thus, there are primitives

1. http://www.jstorimer.com/2013/01/24/threads-not-just-for-speed.html

you can use to say, "Hey, thread scheduler, I'm doing something important here, don't let anybody else cut in until I'm done."

Context switching in practice

Now that you have a basic understanding of the inherent randomness, let's take an inside look at what really happened during the example from the beginning of this chapter.

Here it is again for the sake of posterity:

```
# ./code/snippets/file_uploader.rb

require 'thread'

class FileUploader
  def initialize(files)
    @files = files
  end

  def upload
    threads = []

    @files.each do |(filename, file_data)|
      threads << Thread.new {
        status = upload_to_s3(filename, file_data)
        results << status
      }
```

```
    end

    threads.each(&:join)
  end

  def results
    @results ||= Queue.new
  end

  def upload_to_s3(filename, file)
    # omitted
  end
end

uploader = FileUploader.new('boots.png' => '*pretend png data*', 'shirts.png' => '*pretend png data*')
uploader.upload

puts uploader.results.size
```

The `Queue` class is a thread-safe data structure that ships with Ruby. More about it in the chapter on thread-safe data structures.

Remember, sometimes we may lose one of the statuses in `@results`. Here's how.

First, we need to break apart the `||=` statement. Remember that a thread can be interrupted *at any time*. Although the `||=` operator can't be broken down any further in your Ruby code, there are a lot of underlying operations that support it.

```
# This statement
@results ||= Queue.new
```

```ruby
# when broken down, becomes something like
if @results.nil?
  temp = Queue.new
  @results = temp
end
```

This doesn't break it down to the full extent of the underlying implementation (for that, you'd need to look right to the source of your Ruby implementation), but this breaks it down to essentials.

1. If `@results` currently holds the value of `nil`.
2. Get the return value of the `Queue.new` method.
3. Assign that value to `@results`.

In the example above, the `FileUploader` is instantiated with two files to upload. Let's walk through how that might play out from the perspective of two individual threads.

For the sake of simplicity, I'm going to assume that your code can only run on one CPU core, and that both threads have already been spawned. They're now ready to start executing their block of code.

Now the main thread is blocking on the call to `#join` while the other two threads do their work. Let's call them *Thread A* and *Thread B*.

First, *Thread A* performs its upload to S3 while *Thread B* is paused. It receives the `status` and is ready to push that into `results`.

Thread A checks the value of `@results` and finds that it's currently `nil`. It gets as far as calling `Queue.new`, then the thread scheduler decides this is a good time for a context switch.

Thread A is now paused, while *Thread B* gets a turn. It performs its upload to S3 and receives its `status`. It then checks the value of `@results` and finds that it's `nil`. Remember that *Thread A* never got a chance to assign its Queue back to `@results`.

So *Thread B* creates a Queue and assigns it to `@results`, then pushes its status into it. *Thread B* has done its work, so it terminates. Now *Thread A* is given priority again, so it continues exactly where it left off.

At this moment in time, `@results` is no longer `nil`. It holds the value of the Queue that *Thread B* assigned to it. But *Thread A* is already at step #3 in the `||=` process. It already checked for `nil` and created the Queue. So it goes ahead with step #3 and assigns its Queue to `@results`.

Now *Thread A* pushes its `status` into `results`. *Thread A* is finished, so it terminates. Now the thread scheduler can start the main thread, which has been waiting for these threads to terminate.

The program ends with `@results` having been assigned twice, and ultimately holding just one value: the status of *Thread A*, in this case.

It's entirely possible for the thread scheduler makes a different decision on a different run of this program. Sometimes, `@results` may hold only the status of *Thread B*, or it may hold the statuses from both threads.

The easy fix, in this case, is to not use `||=`. Instead, instantiate `@results` in `initialize`, before any threads are spawned. It's good practice to avoid lazy instantiation in the presence of multiple threads.

This course of events is what's known as a 'race condition.' When you're on the receiving end, this can be *very* hard to track down.

A race condition involves two threads racing to perform an operation on some shared state. In some cases, the race may result in the underlying data becoming incorrect, like in the example I laid out here. In some cases, the race may work out fine and produce the correct result. It's inherently non-deterministic. Still other race conditions may only be exposed under heavy load, when concurrency is highest.

Race conditions occur because a given operation, `@results ||= Queue.new` in this case, is *not* atomic. If the `||=` operation were atomic, there would be no race condition. An atomic operation is one which cannot be interrupted before it's complete. Even in the case of a multi-step operation, with proper protection in place, it will be treated as a single operation, either succeeding or failing without exposing its state mid-operation. This is a key concept you'll see in coming chapters.

Why is this so hard?

Gosh, this sure does look hard. Time for some good news.

Am I saying that any instance of `||=` in Ruby isn't thread-safe? Do you have to stop using `||=`? Absolutely not.

I really wanted to drive home an example of how things can go wrong, and give you a basic understanding of the thread scheduler and its mechanisms. Ultimately, this stuff is hard because it's unpredictable. As I said, you can no longer trace the execution of the program in a predictable way.

But, before we lose hope, I want to share a principle. Understanding this principle, and taking the necessary actions, would have provided the guarantee we were looking for in the example program. Indeed, it really informs thread-safe code at any level.

Any time that you have two or more threads trying to modify the same thing at the same time, you're going to have issues. This is because the thread scheduler can interrupt a thread *at any time*.

This points to two obvious strategies for avoiding issues: 1) don't allow concurrent modification, or 2) protect concurrent modification. We'll talk much more about both of these strategies in the coming chapters.

Understanding this idea of non-determinism and multiple threads of execution is at the core of why multi-threaded programming is considered hard.

But, **multi-threaded programming isn't hard**. Rather, it's no harder than functional programming, or cryptography. These are difficult subjects, but not insurmountable. These are things that any programmer can and should come to understand with some education and experimentation.

Chapter 3

Lifecycle of a Thread

I've given you a few little tastes of the `Thread` API already at this point. Now I'll intentionally show you a bit more of the API that defines the lifecycle of a Thread.

require 'thread'

```
# ./code/snippets/require_thread.rb

puts defined?(Thread) #=> constant
puts defined?(Queue) #=> nil

require 'thread'
```

Funnily enough, `require 'thread'` doesn't load the `Thread` constant. `Thread` is loaded by default, but requiring `'thread'` brings in some utility classes like `Queue`.

Thread.new

You spawn a thread by passing a block to `Thread.new` (or one of its aliases), and optionally passing in block variables like in the `Thread.start` example.

```ruby
Thread.new   { ... }
Thread.fork  { ... }
Thread.start(1, 2) { |x, y| x + y }
```

This reveals a few interesting properties.

Executes the block. You pass a block when spawning the thread. The thread will yield to that block. Either it will reach the end of the block, or an exception will be raised. In either case, the thread terminates.

Returns an instance of `Thread`. Like all constructors, `Thread.new` returns a new instance. Realize that `Thread.new` returns a `Thread` instance representing the sub-thread that was just spawned. Like any other object, calling methods on the new `Thread` instance will affect the spawned thread, not the current thread.

This is the starting point for any thread that you're going to create.

Thread#join

Once you've spawned a thread, you can use `#join` to wait for it to finish.

```ruby
# ./code/snippets/thread_join.rb

thread = Thread.new { sleep 3 }
thread.join
```

```
puts "You'll have to wait 3 seconds to see this"
```

If you run the above code example without calling `#join`, you wouldn't have to wait for 3 seconds. Without `#join`, the main thread would exit before the sub-thread can execute its block. Using `#join` provides a guarantee in this situation.

Calling `#join` on the spawned thread will join the current thread of execution with the spawned one. In other words, where there were previously two independent threads of execution, now the current thread will sleep until the spawned thread exits.

Thread#join and exceptions

When joining two threads, you have to recall how a thread can terminate. In one case, a thread finishes executing its block and then terminates. This is the happy path.

In the other case, a thread raises an unhandled exception before it finishes executing its block. **When one thread raises an unhandled exception, it terminates the thread where the exception was raised, but doesn't affect other threads.**

Similarly, a thread that crashes from an unhandled exception won't be noticed until another thread attempts to join it.

```
# ./code/snippets/thread_join_with_exception.rb
```

```ruby
thread = Thread.new do
  raise 'hell'
end

# simulate work, the exception is unnoticed at this point
sleep 3

# this will re-raise the exception in the current thread
thread.join
```

This shows the literal meaning of 'join.' When one thread has crashed with an unhandled exception, and another thread attempts to join it, the exception is re-raised in the joining thread.

Here's the output of this example in MRI:

```
code/snippets/exception_on_join.rb:2:in `block in <main>': hell (RuntimeError)
```

If you take a close look at the backtrace, you can see that it properly places the site of the exception on line 2, rather than on the line where the join occured.

Thread#value

`Thread#value` is very similar to `#join`: it first joins with the thread, and then returns the last expression from the block of code the thread executed.

```
# ./code/snippets/thread_value.rb
```

```ruby
thread = Thread.new do
  400 + 5
end

puts thread.value #=> 405
```

The `#value` method has the same properties as `#join` regarding unhandled exceptions because it actually calls `#join`. The only difference is in the return value.

Thread#status

Every thread has a status, accessible from `Thread#status`.

It's probably most common for one thread to check the status of some other thread, but it is possible for a thread to check its own status using `Thread.current.status`.

Ruby defines several possible values for `Thread#status`.

- `'run'`: Threads currently running have this status.
- `'sleep'`: Threads currently sleeping, blocked waiting for a mutex, or waiting on IO, have this status.
- `false`: Threads that finished executing their block of code, or were successfully killed, have this status.
- `nil`: Threads that raised an unhandled exception have this status.

- `aborting`: Threads that are currently running, yet dying, have this status.

```
# ./code/snippets/thread_status.rb

adder = Thread.new do
  # Here this thread checks its own status.
  Thread.current.status #=> 'run'
  2 * 3
end

puts adder.status #=> 'run'
adder.join
puts adder.status #=> false
```

Thread.stop

This method puts the current thread to sleep *and* tells the thread scheduler to schedule some other thread. It will remain in this sleeping state until its alternate, `Thread#wakeup` is invoked. Once `#wakeup` is called, the thread is back into the thread scheduler's realm of responsibility.

```
# ./code/snippets/thread_stop.rb

thread = Thread.new do
  Thread.stop
  puts 'Hello there'
end
```

```
# wait for the thread to trigger its stop
nil until thread.status == 'sleep'

thread.wakeup
thread.join
```

Thread.pass

This one is similar to `Thread.stop`, but instead of putting the current thread to sleep, it just asks the thread scheduler to schedule some other thread. Since the current thread doesn't sleep, it can't guarantee that the thread scheduler will take the hint.

Avoid Thread#raise

This method *should not* be used. It doesn't properly respect `ensure` blocks, which can lead to nasty problems in your code.

In terms of functionality, this method will allow a caller external to the thread to raise an exception inside the thread. In fact, the backtrace will actually point to whatever line the thread happened to be executing when this is called, which is not useful for debugging.

```
# ./code/snippets/thread_raise.rb
```

```ruby
$cleaned_up = false

t = Thread.new do
  begin
    # nada
  ensure
    sleep
    $cleaned_up = true
  end
end

nil until t.status == 'sleep'
# At this point, the thread should be sleeping
# inside the ensure block. Now raise an exception
# inside the thread.
t.raise 'hell'

# Joining with the thread will cause the exception
# to be re-raised here.
begin
  t.join
rescue
end

# This value is false because the ensure block
# was aborted when Thread#raise was called. This
# breaks the contract that ensure blocks provide.
puts $cleaned_up #=> false
```

```
1
```

Avoid Thread#kill

This method should not be used for the exact same reason as the one above.

Supported across implementations

This `Thread` API is a Ruby API. I've hinted that the different Ruby implementations have different underlying threading behaviours. That's certainly the case, but **all the Ruby implementations we're looking at in this book support this same `Thread` API.**

1. Adapted from an example by James Tucker.

Chapter 4

Concurrent != Parallel

I hinted in previous chapters that threads provide a concurrency mechanism, one capable of utilizing multi-core systems.

A sensible question following this might be:

> *So multiple threads will be running my code in parallel, right?*

Before I can answer that question, I need to clear up a common misunderstanding: **concurrent and parallel are not the same thing.**

Keeping that in mind, I can rephrase your question into two more sensible questions:

1. Do multiple threads run your code concurrently? Yes.
2. Do multiple threads run your code in parallel? Maybe.

Now I'll attempt to explain the difference. Since we're all programmers here, I'll use a programmer at work as an example.

An illustrative example

Imagine you're a programmer working for an agency. They have two projects that need to be completed. Both will require one full day of programmer time. There are (at least) three ways that this can be accomplished.

1. You could complete Project A today, then complete Project B tomorrow.
2. You could work on Project A for a few hours this morning, then switch to Project B for a few hours this afternoon, and then do the same thing tomorrow. Both projects will be finished at the end of the second day.
3. Your agency could hire another programmer. He could work on Project B and you could work on Project A. Both projects will be finished at the end of the first day.

This example is a bit contrived. It doesn't take into account the time required to switch projects, ramp-up time, inevitable delays, etc. But let's pretend things work this way just for the sake of example.

What do these three ways of working represent?

The first way represents working serially. This is the normal path of single-threaded code. Given two tasks, they will be performed in order, one after another. Very organized and easy to follow.

The second way represents working concurrently. This represents the path of multi-threaded code running on a single CPU core. Given two tasks, they will each be

performed at the same time, inching forward bit by bit. In this case there's just one programmer, or CPU, and the tasks compete to get access to this valuable resource. Otherwise, their work won't progress.

This way nicely illustrates that concurrent, multi-threaded code doesn't necessarily run faster than single-threaded code. In this case the tasks aren't being accomplished any quicker; they're just being organized differently.

The third way represents working in parallel. This represents the path of multi-threaded code running on a multi-core CPU. Given two tasks, they will be performed simultaneously, completing in half the time. Notice the subtle difference between #2 and #3. Both are concurrent, but only #3 is parallel.

The way I've explained it, #3 almost looks like two instances of #1 progressing side-by-side. This is one possible configuration, but it's also possible that as one programmer gets stuck on an issue, a context switch takes place. In this example, another programmer might come in to take his place. This preserves the 'inching forward bit by bit' idea, except now there are sufficient resources to keep the process inching forward continually.

Notice that more resources are required in order to work in parallel. The idea of one programmer working on two projects simultaneously, with one hand on each of two keyboards, for instance, is just as absurd as one CPU core executing two instructions simultaneously.

You can't guarantee anything will be parallel

This last example illustrated that your code can be concurrent without being parallel. Indeed, all you can do is to organize your code to be concurrent, using multiple threads, for instance. But **making it execute in parallel is out of your hands. That responsibility is left to the underlying thread scheduler.**

For instance, if you're running on a 4-core CPU, and you spawn 4 threads, it's possible, but unlikely, that your code will all be executed on just one CPU core. That's ultimately the choice of the thread scheduler. In practice, thread schedulers employ fair queueing so that all threads are given more-or-less equal access to available resources, but that cannot be controlled by your code.

The point of all this is to say that when you write multi-threaded code, you have no guarantee as to the parallelism of the environment that your code will be executed in. In practice, you should assume that your concurrent code *will* be running in parallel because it typically will, but you can't guarantee it!

Given this, multi-threaded code should be referred to as concurrent, rather than parallel. There's very little you can do from your side of the keyboard to guarantee that your code will run in parallel. However, **by making it concurrent, you enable it to be parallelized when the underlying system allows it.**

This is an important topic to grasp, so if you're still not 100% clear on concurrency versus parallelism, I highly recommend these two other explanations of the same conflation:

1. Rob Pike describes how concurrency enables parallelism, but they're not the same thing. Lots of simple diagrams to explain key concepts. link
2. Evan Phoenix describes the difference between concurrency and parallelism and how it relates to existing Ruby implementations. link

The relevance

I've been saying this is an important concept to grasp. The first reason is so that you can use the right terms when you're talking about it.

The second reason is that this lays a foundation for understanding the effects of thread synchronization and locking, which will get more coverage in coming chapters. Before that, this knowledge will drive your understanding of MRI's infamous GIL.

Chapter 5

The GIL and MRI

In the last chapter, you learned the key differences between concurrency and parallelism. This is an important concept because **MRI allows concurrent execution of Ruby code, but prevents parallel execution of Ruby code.**

I'll pre-emptively answer a question I'm sure you have: do JRuby and Rubinius have a GIL? Do they allow parallel execution of Ruby code? Both JRuby and Rubinius do *not* have a GIL, and thus *do* allow parallel execution of Ruby code.

At this point, the term GIL might be unfamiliar to you. Let's tease it apart.

The global lock

The term GIL stands for Global Interpreter Lock. It's sometimes called the GVL (Global VM Lock) or just The Global Lock. All three of these terms refer to the same thing. I'll continue to use the term GIL from now on.

So what's the GIL? The GIL is a global lock *around the execution of Ruby code.*

Think of it this way: there is one, and only one, GIL per instance of MRI (or per MRI process). This means that if you spawn a bunch of MRI processes from your terminal, each one will have its own GIL.

If one of those MRI processes spawns multiple threads, that group of threads will share the GIL for that process.

If one of these threads wants to execute some Ruby code, it will have to acquire this lock. One, and only one, thread can hold the lock at any given time. While one thread holds the lock, other threads need to wait for their turn to acquire the lock.

This has some very important implications for MRI. The biggest implication is that **Ruby code will never run in parallel on MRI.** The GIL prevents it.

An inside look at MRI

I want to walk you through the viewpoint of a thread trying to execute some Ruby code inside the MRI virtual machine. This will give you a better understanding of exactly how the GIL works and where it fits in.

Let's pretend that MRI is executing this bit of code:

```
require 'digest/md5'

3.times.map do
  Thread.new do
    Digest::MD5.hexdigest(rand)
  end
end.each(&:value)
```

Nothing too exciting. Each thread will generate an MD5 digest based on a random number.

For simplicity's sake, I'm going to skip right to the interesting part and assume that the threads have already been spawned. We'll jump in assuming we have three threads spawned and ready to execute their block of Ruby code.

Remember that each MRI thread is backed by a native thread, and from the kernel's point of view, they're all executing in parallel. The GIL is a detail inside of MRI and doesn't come into play except when executing Ruby code.

Since all three threads want to execute Ruby code, they all attempt to acquire the GIL. The GIL is implemented as a mutex (something you'll see more of very soon). The operating system will guarantee that one, and only one, thread can hold the mutex at any time.

So all three threads attempt to acquire the GIL, but only one thread actually acquires it. The other two threads are put to sleep until that mutex becomes available again.

The thread that acquired the GIL (let's call it *Thread A*), now has the exclusive right to execute Ruby code inside of MRI. Until *Thread A* releases the GIL, the other threads won't get a chance to execute any Ruby code.

At this point, *Thread A* executes some arbitrary amount of Ruby code. How much? That's unspecified, and left up to the MRI internals. After a certain interval, *Thread A* releases the GIL. This triggers the thread scheduler to wake up the other two threads that were sleeping, waiting on this mutex.

Now both of these threads vie for the GIL, and the kernel must decide again which thread will acquire it.

It's given to a new thread; let's call it *Thread B*. Now *Thread B* has exclusive ownership of the GIL. It can execute the Ruby code that it needs to. Meanwhile, *Thread A* has gone back and attempted to acquire the GIL again. So once again, the other two threads are sleeping, blocked waiting for their turn to acquire the GIL.

This should make it crystal clear how the GIL prevents parallel execution of Ruby code. It's only possible for one thread to execute Ruby code at any given time.

The special case: blocking IO

Before I get to the motivations for this behaviour, I have to tell you about a special case: blocking IO. I've been saying the GIL prevents parallel execution of Ruby code, but blocking IO is not Ruby code.

In the above walkthrough, what happens when a thread executes some Ruby code that blocks on IO? Let's say our example looked like this:

```
require 'open-uri'

3.times.map do
  Thread.new do
    open('http://zombo.com')
  end
end.each(&:value)
```

This Ruby code will trigger an HTTP request to be sent to the `zombo.com` server. Depending on network conditions and the status of `zombo.com`, this may take a long time to finish. Thankfully, **MRI doesn't let a thread hog the GIL when it hits blocking IO.**

This is a no-brainer optimization for MRI. When a thread is blocked waiting for IO, it won't be executing any Ruby code. Hence, when a thread is blocking on IO, it releases the GIL so another thread can execute Ruby code.

For the sake of posterity, let's quickly run through the walkthrough from above with this blocking IO example. We're already at the point where all the threads have been spawned. Now they all attempt to acquire the GIL to execute Ruby code.

Thread A gets the GIL. It starts executing Ruby code. It gets down to Ruby's `Socket` APIs and attempts to open a connection to `zombo.com`. At this point, while *Thread A* is waiting for its response, it releases the GIL. Now *Thread B* acquires the GIL and goes through the same steps.

Meanwhile, *Thread A* is still waiting for its response. Remember that the threads can execute in parallel, so long as they're not executing Ruby code. So it's quite possible for *Thread A* and *Thread B* to both have initiated their connections, and both be waiting for a response.

Under the hood, each thread is using a ppoll(2) system call to be notified when their connection attempt succeeds or fails. When the ppoll(2) call returns, the socket will have some data ready for consumption. At this point, the threads will need to execute

Ruby code to process the data. So now the whole process starts over again. I think you get the idea.

Why?

This walkthrough has painted a bit of a bleak picture of MRI internals. It seems that MRI is intentionally placing a huge restriction on parallel code execution. Why would it do this?

Surely, this isn't meant as a malicious decision against you. Indeed, MRI core developers have been calling the GIL a feature for some time now, rather than a bug. In other words, **the MRI team has expressed no intention of getting rid of the GIL**.

There are three reasons that the GIL exists:

1. To protect MRI internals from race conditions

   I've only covered this topic briefly thus far, but I've stressed that race conditions in your code can cause issues. The same issues that can happen in your Ruby code can happen in MRI's C code. When it's running in a multi-threaded context, it will need to protect critical parts of the internals with some kind of synchronization mechanism.

   The easiest way to reduce the number of race conditions that can affect the internals is to prevent multiple threads from executing in parallel.

2. To facilitate the C extension API

   The 'C extension API' is a C interface to MRI's internal functions, often used when people want to interface Ruby with a C library.

   Calling an MRI function from C is subject to the same GIL that the equivalent Ruby code would be subject to.

   This Ruby code:

   ```
   array = Array.new
   array.pop
   ```

   is equivalent to this C code:

   ```
   VALUE ary = rb_ary_new();
   VALUE last_element = rb_ary_pop(ary);
   ```

   The important thing to note is that these calls to the C extension API lock the GIL, just like the corresponding Ruby code.

   The other reason that the GIL exists for C extensions is so that MRI can function safely, even in the presence of C extensions that may not be thread-safe. Especially when wanting to integrate an existing C library with Ruby, thread-safety is not always guaranteed.

3. To reduce the likelihood of race conditions in your Ruby code

Just as the easiest way to protect MRI internals from race conditions is to disallow real parallel threading, it's also the easiest way to protect *your* Ruby code from race conditions. In this regard, the GIL reduces the likelihood that you shoot yourself in the foot when using multiple threads. However, the cost for this reduction in entropy is high.

In many situations, Ruby provides a lot of power to you, the user of the language. It trusts that you will do the right thing, knowing that it's possible to really get things wrong. However, when it comes to multi-threading, MRI takes the opposite approach. They remove the ability of the system to do parallel threading.

It's a bit like wearing fully body armour to walk down the street: it really helps if you get attacked, but most of the time it's just confining.

It's important to note that the GIL only reduces entropy here; it can't rule it out all together. The next section goes into more detail on this.

Misconceptions

Now that you have an understanding of what the GIL is, and why it exists, this is probably the most important section of this chapter.

Over time, people have made false assumptions or incredible claims about the GIL. Unfortunately, it's a misunderstood part of MRI. This leaves us in a situation where

many people have an unwitting negative, or positive, impression of the GIL, without really understanding what it does or what it guarantees.

Up until now, I've given you some idea about how it works; now I'll try to dispel two myths in the community.

Myth: the GIL guarantees your code will be thread-safe.

This isn't true. Throughout this chapter, I've been careful to say that the GIL reduces the likelihood of a race condition, but can't prevent it. When multiple threads are running in parallel, and a race condition is possible, the likelihood of it happening is higher. There are simply more opportunities for things to go wrong.

But with a GIL, two threads are never running Ruby code in parallel. This greatly reduces the likelihood of things going wrong, but still can't prevent it. Let's use this Ruby code as an example:

```
# ./code/snippets/unsafe_counter.rb

@counter = 0

5.times.map do
  Thread.new do
    temp = @counter
    temp = temp + 1

    @counter = temp
```

```
    end
end.each(&:join)

puts @counter
```

This probably isn't code that you would write every day, but it illustrates the point. This is the expanded version of the `+=` operator.

With no synchronization, even with a GIL, it's possible that a context switch happens between incrementing `temp` and assigning it back to `counter`. If this is the case, it's possible that two threads assign the same value to `counter`. In the end the result of this little snippet could be less than 5.

It's rare to get an incorrect answer using MRI with this snippet, but almost guaranteed if you use JRuby or Rubinius. If you insert a `puts` in the middle of the block passed to Thread.new, then it's very likely that MRI will produce an incorrect result. Its behaviour with blocking IO encourages a context switch while waiting for the thread to print to stdout.

As you get a better understanding of race conditions and thread safety in the upcoming chapters, the absurdity of the claim that the GIL guarantees thread safety will become clearer.

Myth: the GIL prevents concurrency.

This is a misunderstanding of terms.

The GIL prevents *parallel* execution of Ruby code, but it doesn't prevent concurrent execution of Ruby code. Remember that concurrent code execution is possible even on a single core CPU by giving each thread a turn with the resources. This is the situation with MRI and the GIL.

So the GIL prevents parallel execution of Ruby code, but not concurrent execution.

The other important thing to remember is the caveat: blocking IO. The GIL allows multiple threads to be simultaneously blocked on IO. This means that you can use multiple threads to parallelize code that is IO-bound.

MRI's multi-threading behaviour, with respect to blocking IO, is actually very similar to the behaviour of JRuby or Rubinius in the face of blocking IO. Any of these implementations will allow blocking IO to be parallelized.

Chapter 6

Real Parallel Threading with JRuby and Rubinius

So MRI has a GIL. The GIL prevents real parallel threading. What about JRuby and Rubinius?

JRuby and Rubinius don't have a GIL. This means they *do* allow for real parallel execution of Ruby code.

Proof

Before I get into any more technical explanations, I want to prove what I'm saying!

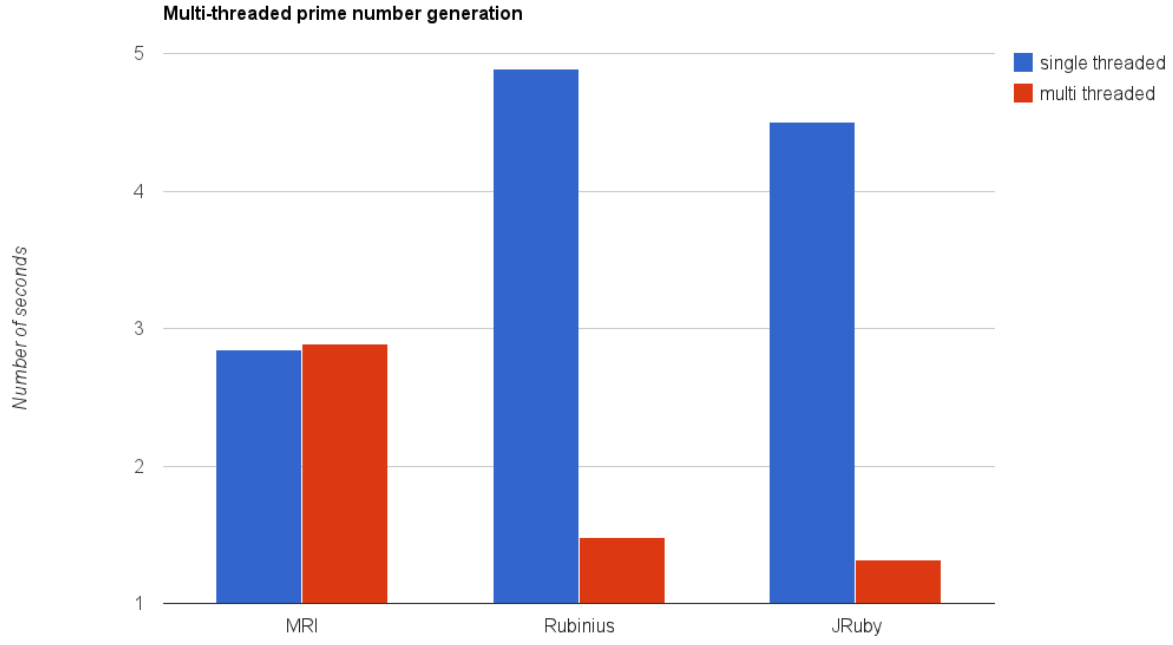
The following benchmark calculates 1 million prime numbers 10 times. I chose this particular example because it executes a lot of Ruby code, and because it's easy to spread the work across multiple threads.

It uses Ruby's `Prime` library to generate the primes. The first benchmark block does it ten times in one thread; the second block does it two times in each of five threads.

```
# ./code/benchmarks/prime.rb
```

```ruby
require 'benchmark'
require 'prime'

primes = 1_000_000
iterations = 10

num_threads = 5
iterations_per_thread = iterations / num_threads

# warmup (initialize the underlying singleton)
Prime.each(primes) { }

Benchmark.bm(15) do |x|
  x.report('single-threaded') do
    iterations.times do
      Prime.each(primes) { }
    end
  end

  x.report('multi-threaded') do
    num_threads.times.map do
      Thread.new do
        iterations_per_thread.times do
          Prime.each(primes) { }
        end
      end
    end.each(&:join)
  end
end
```

Here are the results across MRI, JRuby, and Rubinius:



The various implementations have different performance profiles when it comes to heavy math calculations, but the thing to focus on is the relative difference within each implementation.

Both JRuby and Rubinius were able to calculate the result significantly faster when using multiple threads. Contrast that with MRI. MRI's multi-threaded results showed no significant improvement over its single-threaded result.

These figures are a direct representation of the GIL. The GIL in MRI prevented Ruby code from being executed in parallel, hence no speedup.

But...don't they need a GIL?

At the end of the last chapter, I gave you three reasons why the GIL exists in MRI. This would be a good time to address how the other implementations solve these problems. Let's take them in order.

1. To protect internals from race conditions

   JRuby and Rubinius do indeed protect their internals from race conditions. But rather than wrapping a lock around the execution of all Ruby code, they protect their internal data structures with many fine-grained locks.

   Rubinius, for instance, replaced their GIL with about 50 fine-grained locks[1]. Without a GIL, its internals are still protected, but because it spends less time in these locks, more of your Ruby code can run in parallel.

2. To facilitate the C extension API

1. http://www.jstorimer.com/2013/03/26/brian-shirai-threads.html

JRuby has an easy answer to this: it doesn't support the C extension API. For gems that need to run natively, they support Java-based extensions. I won't go into any more detail about this.

Rubinius *does* support the MRI C extension API. Interestingly, it simply preserves its GIL-less behaviour in the presence of these extensions. They opted to try to help gem authors fix thread-safety issues, rather than try to prevent them from happening. They say that no issues have cropped up thus far[2].

3. To reduce the likelihood of race conditions in your Ruby code

   JRuby and Rubinius don't have an answer to this one. Any thread-safety issues in your code are likely to crop up much more quickly when using these implementations. The simple reason is that your code will actually be running in parallel.

   The only way to solve this issue is to be aware of thread safety when writing your code. There's lots more about this topic in the next few chapters.

2. http://www.jstorimer.com/2013/03/26/brian-shirai-threads.html

Chapter 7

How Many Threads Are Too Many?

First, in order to benefit from concurrency, we must be able to partition problems into smaller tasks that can be run concurrently. If a problem has a significant part that can't be partitioned, then the application may not really see much performance gain by making it concurrent.

This question is relevant whether you're whipping up a quick script to scrape some websites, trying to speed up a long-running calculation, or tuning your multi-threaded webserver.

I hope you're not surprised, but the answer is: **Well, that depends.**

The only way to be sure is to measure and compare. Try it out with 1 thread per CPU core, try it out with 5 threads per CPU core, compare the results, and move forward.

However, there are some heuristics you can use to get started. Plus, if you're new to this, it's good to have a ballpark idea of what's sane. Are 100 threads too many? How about 10,000?

ALL the threads

How about we start by spawning ALL the threads? How many threads can we spawn?

```ruby
1.upto(10_000) do |i|
  Thread.new { sleep }
  puts i
end
```

This code attempts to spawn 10,000 sleeping threads. If you run this on OSX you'll get about this far:

```
2042
2043
2044
2045
2046
ThreadError: can't create Thread (35)
```

On OSX, there's a hard limit on the number of threads that one process can spawn. On recent versions, that limit is somewhere around 2000 threads.

However, we were able to spawn at least that many threads without the system falling down around us. As mentioned earlier, spawning a thread is relatively cheap.

If I run this same bit of code on a Linux machine, I'm able to spawn 10,000 threads without blinking. So, **it's possible to spawn a lot of threads**, but you probably don't want to.

Context Switching

If you think about it for a minute, it should be clear why you don't want to spawn 10,000 threads on a 4-core CPU.

Even though each thread requires little memory overhead, there is overhead for the thread scheduler to manage those threads. If you have only have 4 cores, then only 4 threads can be executing instructions at any given time. You may have some threads blocking on IO, but that still leaves a lot of threads idle a lot of the time, requiring overhead to manage them.

But even in the face of increased overhead for the thread scheduler, it does sometimes make sense to spawn more threads than the number of CPU cores. Let's review the difference between IO-bound code and CPU-bound code.

IO-bound

Your code is said to be IO-bound if it is bound by the IO capabilities of your machine. Let me try that again with an example.

If your code were doing web requests to external services, and your machine magically got a faster network connection, your program would likely be sped up as well.

If your code were doing a lot of reading and writing to disk, and you got a new hard drive that could seek faster, your code would likely be sped up, again, as a result of better hardware.

These are examples of IO-bound code. In these situations, your code typically spends some time waiting for a response from some IO device, and the results aren't immediate.

In this case, it *does* make sense to spawn more threads than CPU cores.

If you need to make 10 web requests, you're probably best off to make them all in parallel threads, rather than spawning 4 threads, making 4 requests, waiting, then making 4 more requests. You want to minimize that idle time spent waiting.

Let's make this concrete with some code.

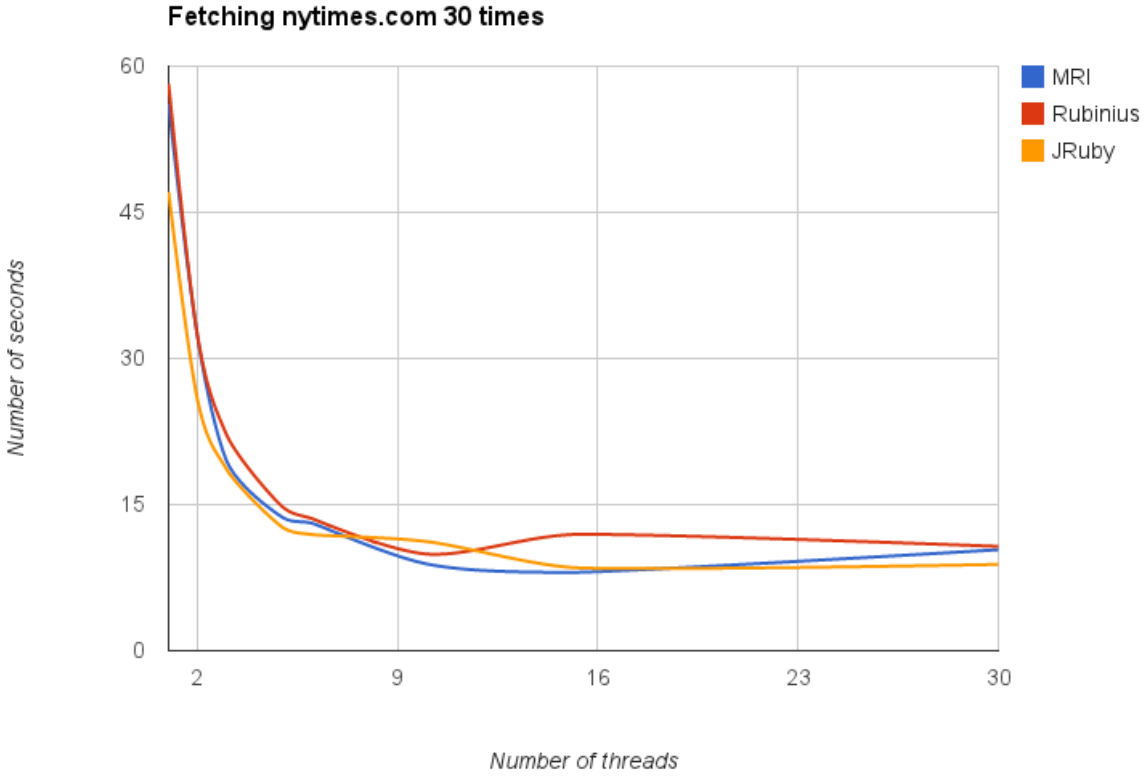
In this example, the task is to fetch `http://nytimes.com` 30 times. I tested this using different numbers of threads. With one thread, 30 fetches will be performed serially. With two threads, each thread will need to perform 15 fetches, etc.

I tested different numbers of threads in order to find the sweet spot. Make no mistake, **there will be a sweet spot between utilizing available resources and context switching overhead.**

```
# ./code/benchmarks/io_bound.rb
```

```ruby
require 'benchmark'
require 'open-uri'

URL = 'http://www.nytimes.com/'
ITERATIONS = 30

def fetch_url(thread_count)
  threads = []

  thread_count.times do
    threads << Thread.new do
      fetches_per_thread = ITERATIONS / thread_count

      fetches_per_thread.times do
        open(URL)
      end
    end
  end

  threads.each(&:join)
end

Benchmark.bm(20) do |bm|
  [1, 2, 3, 5, 6, 10, 15, 30].each do |thread_count|
    bm.report("with #{thread_count} threads") do
      fetch_url(thread_count)
    end
  end
end
```

I plotted the results across the studied Ruby implementations to get this graph illustrating the results:



I ran this benchmark on my 4-core Macbook Pro.

As expected, all of the implementations exhibited similar behaviour with respect to concurrent IO.

The sweet spot for code that is fully IO-bound is right around 10 threads on my machine, even though I tested up to 30 threads. Using more than 10 threads no longer improves performance in this case; it just uses more resources. For some cases, it actually made performance worse.

If the latency of the IO operations were higher, I might need more threads to hit that sweet spot, because more threads would be blocked waiting for responses more of the time. Conversely, if the latency were lower, I might need fewer threads to hit the sweet spot because there would be less time spent waiting, as each thread would be freed up more quickly.

Finding the sweet spot is really important. Once you've done this a few times, you can probably start to make good guesses about how many threads to use, but the sweet spot will be different with different IO workloads, or different hardware.

CPU-bound

The flip side of IO-bound code is CPU-bound code. Your code is CPU-bound if its execution is bound by the CPU capabilities of your machine. Let's try another example.

If your code needed to calculate millions of cryptographic hashes, or perform really complex mathematical calculations, these things would be bound by how much throughput your CPU could muster in terms of CPU instructions.

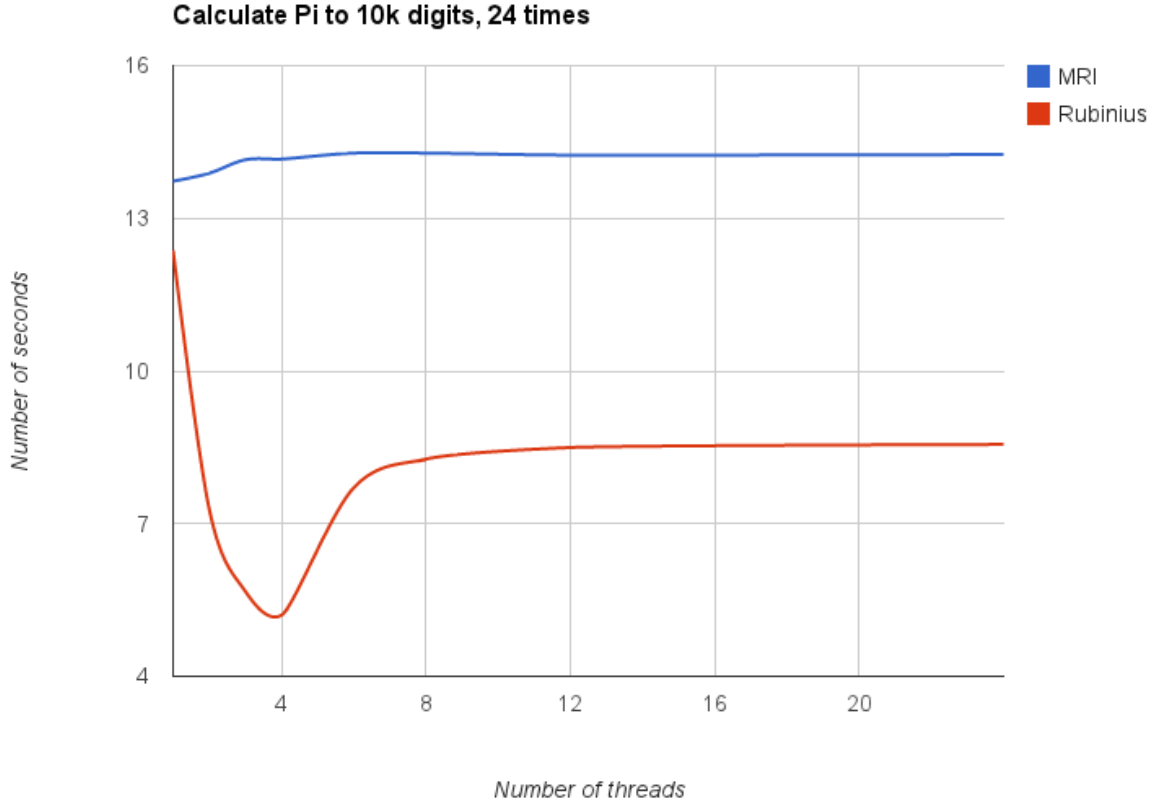
If you upgraded to a CPU with a faster clock speed, your code would be able to do these calculations in a shorter time period.

Let's re-use the example from the last chapter that calculated digits of pi. That's certainly a CPU-bound bit of code. Here's the benchmark code:

```
# ./code/benchmarks/cpu_bound.rb

require 'benchmark'

require 'bigdecimal'
require 'bigdecimal/math'

DIGITS = 10_000
ITERATIONS = 24

def calculate_pi(thread_count)
  threads = []

  thread_count.times do
    threads << Thread.new do
      iterations_per_thread = ITERATIONS / thread_count

      iterations_per_thread.times do
        BigMath.PI(DIGITS)
      end
    end
  end

  threads.each(&:join)
```

```
end

Benchmark.bm(20) do |bm|
  [1, 2, 3, 4, 6, 8, 12, 24].each do |thread_count|
    bm.report("with #{thread_count} threads") do
      calculate_pi(thread_count)
    end
  end
end
```

This graph illustrates the results:


Calculate Pi to 10k digits, 24 times
16
13
10
7
4
Number of seconds
4
8
12
16
20
Number of threads
MRI
Rubinius

Note: JRuby seems to have an issue with calculating that many digits of pi. Its results weren't comparable, so I left them out.

This graph has a bit of a different shape than the last one. Again, the absolute values aren't important here; the shape is the important part.

For MRI, there's no curve this time. Performance isn't impacted with the introduction of more threads. This is a direct result of the GIL. Since the GIL is a lock around the execution of Ruby code, and in this case we're benchmarking execution of Ruby code, using more threads has no effect.

For Rubinius, the curve was lowest when running with 4 threads. I ran this example locally on my 4-core CPU. That's no coincidence.

CPU-bound code is inherently bound by the rate at which the CPU can execute instructions. So when I have my code running in parallel across 4 threads, I'm utilizing all of the power of my 4-core CPU without any overhead.

Once you introduce more than 4 threads, it's still possible for you to fully utilize all of your CPU resources, but now there's more overhead. That's why we see the curve going back up. The thread scheduler is incurring overhead for each extra thread.

The takeaway from all this is that each thread you spawn incurs overhead to manage it. **Creating more threads isn't necessarily faster.** On the other hand, introducing more threads improved performance in these two examples by anywhere between 100% and 600%. Finding that sweet spot is certainly worth it.

So... how many should you use?

I showed some heuristics for code that is fully IO-bound or fully CPU-bound. In reality, your application is probably not so clear cut. Your app may be IO-bound in some places, and CPU-bound in other places. Your app may not be bound by CPU or IO. It may be memory-bound, or simply not maximizing resources in any way.

A Rails application is a prime example of this. Between communicating with the database, communicating with the client, and calling out to external services, there are lots of chances for it to be IO-bound. On the other hand, there are things that will tax the CPU, like rendering HTML templates, or transforming data to JSON documents.

So, as I said at the beginning of this chapter, **the only way to a surefire answer is to measure.** Run your code with different thread counts, measure the results, and then decide. Without measuring, you may never find the 'right' answer.

Chapter 8

Thread safety

Right away, I want to acknowledge that **the term 'thread safety' gets thrown around a lot**. Unfortunately, it's rarely clear what exactly thread safety is about. You'll sometimes hear about a given library or piece of code being thread-safe or not thread-safe. But if it's not thread-safe, what will happen?

Will your program crash? Will the server start on fire? Will subtle bugs be magically introduced at a slow but consistent rate, without any possibility of reproducing them?

What's really at stake?

If your code is 'thread-safe,' that means that it can run in a multi-threaded context and your underlying data will be safe. By data, I'm not talking about what's in your database; I'm talking about what values your program has stored in memory. Your data is what's at stake.

When your code isn't thread-safe, the worst that can happen is that your underlying data becomes incorrect, yet your program continues as if it were correct.

There are a few more ways that we can say that that might make it clearer.

- If your code is 'thread-safe,' that means that you can run your code in a multi-threaded context and your underlying data will be safe.
- If your code is 'thread-safe,' that means that you can run your code in a multi-threaded context and your underlying data remains consistent.
- If your code is 'thread-safe,' that means that you can run your code in a multi-threaded context and the semantics of your program are always correct.

What's really at stake when your code isn't thread-safe is your data.

Once again, let's make this concrete with an example.

```ruby
# ./code/snippets/concurrent_payment.rb

# This class represents an ecommerce order
Order = Struct.new(:amount, :status) do
  def pending?
    status == 'pending'
  end

  def collect_payment
    puts "Collecting payment..."
    self.status = 'paid'
  end
end

# Create a pending order for $100
order = Order.new(100.00, 'pending')
```

```
# Ask 5 threads to check the status, and collect
# payment if it's 'pending'
5.times.map do
  Thread.new do
    if order.pending?
      order.collect_payment
    end
  end
end.each(&:join)
```

This is a variant of what's called the 'check-then-set' race condition. The name says it all. This race condition is manifested by code that first checks a condition, then does something to change its value.

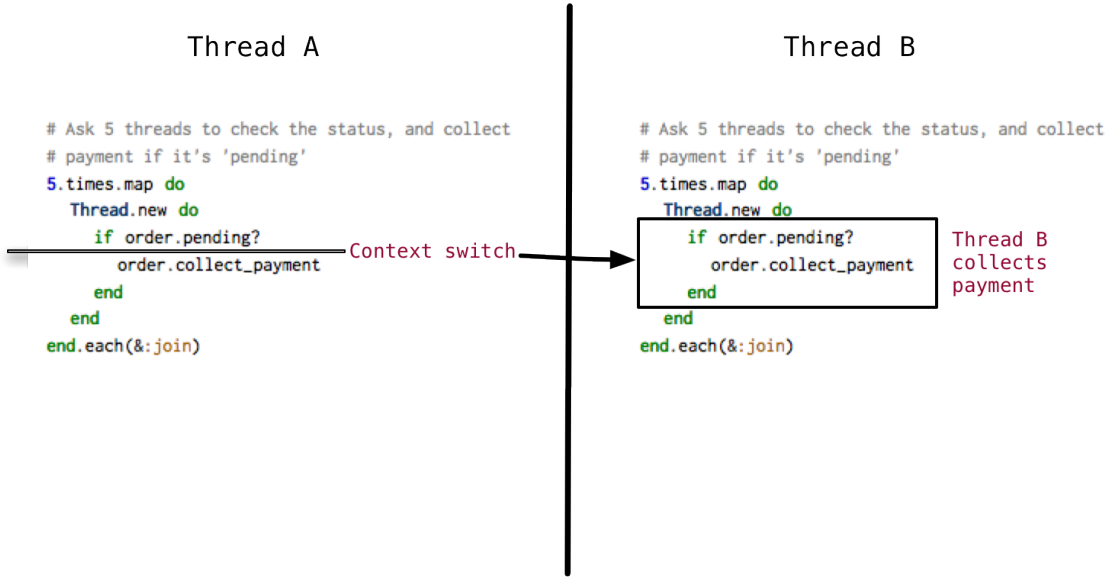
At first glance, this code may look innocent. But here's some sample output from running this code.

```
$ ruby code/snippets/concurrent_payment.rb
Collecting payment...
Collecting payment...

$ jruby code/snippets/concurrent_payment.rb
Collecting payment...Collecting payment...
Collecting payment...Collecting payment...
Collecting payment...

$ rbx code/snippets/concurrent_payment.rb
Collecting payment...Collecting payment...
Collecting payment...Collecting payment...
```

Yikes! Your customers won't be happy if you're charging multiple times for each order.

I encourage you to try out this sample code. You will continually get different, yet incorrect, results. This is a result of the check-then-set race condition. This is not Ruby's fault; this is your fault for not properly synchronizing access to the `order`.

Let's quickly review what happened here. The problem here is similar to the issue we saw with the `+=` operator in a previous chapter. We have a multi-step operation (checking the order status, then collecting payment and setting the status) that can be interrupted before it's finished. As the results showed, it's quite possible, even likely, that this multi-step operation will be interrupted, such that one thread progresses partway through the operation, then another thread does the same. The end result is that the really critical piece, the part that collects payment, is performed twice.



Here, Thread A checks the `order.pending?` condition and finds it to be true. A context switch immediately takes place, pausing Thread A before it can collect payment. Then, Thread B finds the condition to be true and proceeds to collect payment. Once Thread A becomes active again, it will pick up right where it left off and collect payment again, charging the customer for a second time.

Key operations like this need to be made atomic. Your code needs to tell the thread scheduler that this multi-step operation should not be interrupted. You'll see how that can be done in the next chapter.

The computer is oblivious

I want you to imagine this code as part of a larger ecommerce system. If that were the case, then this incorrect behaviour would seem to go unchecked. In other words, your code would have no idea that it's going forward with the system in an incorrect state. From a human perspective, we can see that collecting payment twice is bad, but this particular class has no notion that things have gone wrong.

All this to say that when Ruby produced the incorrect result from the example above, it didn't come with an exception, or a process aborting. **The computer is unaware of thread-safety issues**. The onus is on you to notice these problems and deal with them.

This is one of the hardest problems when it comes to thread safety. There are no exceptions raised or alarm bells rung when the underlying data is no longer correct. Even worse, sometimes it takes a heavy load to expose a race condition like this. Something might not be noticed during development, but then crop up during a critical time in production.

Is anything thread-safe by default?

At this point, you may be wondering: is anything thread-safe by default?

In Ruby, very few things are *guaranteed* to be thread-safe by default. Even compound operators, like `+=` or `||=`, although they are a single operation in Ruby, are not a single atomic, thread-safe operation from the perspective of the underlying VM.

The same is true for core collection classes. Things like `Array` and `Hash` are not thread-safe by default. Let's illustrate this with a silly example.

```
# ./code/snippets/concurrent_array_pushing.rb

shared_array = Array.new

10.times.map do
  Thread.new do
    1000.times do
      shared_array << nil
    end
  end
end.each(&:join)

puts shared_array.size
```

In this silly example you have 10 threads each appending 1000 elements to a shared `Array`. In the end, the array should have `10 * 1000 = 10,000` elements.

```
$ ruby code/snippets/concurrent_array_pushing.rb
10000
$ jruby code/snippets/concurrent_array_pushing.rb
7521
```

```
$ rbx code/snippets/concurrent_array_pushing.rb
8541
```

Again we see an incorrect result with no exceptions raised by Ruby. This does not mean that you need to stop using `<<` or Arrays, just that you need to be aware of what guarantees they provide.

The reason that the above code example was susceptible to issues is that multiple threads shared an object and attempted to update that object at the same time.

Remember that **any concurrent modifications to the same object are not thread-safe**. This includes things like adding an element to an `Array`, or regular ol' assignment, any concurrent modification.

In any of these situations, your underlying data is not safe if that operation will be performed on the same region of memory by multiple threads.

This is not nearly as scary as it sounds. I'm only showing you one side of the equation.

If you read between the lines, you'll see that these operations are fine to use in a threaded program, so long as you can guarantee that multiple threads won't be performing the same modification to the same object at the same time.

These 'guarantees' are really the crux of making your programs thread-safe. The world of multi-threading is a world of chaos. Thankfully, you do have some mechanisms

available to bring a bit of order to this chaos. The next few chapters will cover a few different ways you can do that.

The good news in all of this? Most of the time, just writing good, idiomatic Ruby will lead to thread-safe code.

Chapter 9

Protecting Data with Mutexes

Mutexes provide a mechanism for multiple threads to synchronize access to a critical portion of code. In other words, they help bring some order, and some guarantees, to the world of multi-threaded chaos.

Mutual exclusion

The name 'mutex' is shorthand for 'mutual exclusion.' **If you wrap some section of your code with a mutex, you guarantee that no two threads can enter that section at the same time.**

I'll demonstrate first using the silly Array-pushing example from the previous chapter.

```
# ./code/snippets/concurrent_array_pushing_with_mutex.rb

shared_array = Array.new
mutex = Mutex.new

10.times.map do
  Thread.new do
    1000.times do
      mutex.lock
      shared_array << nil
```

```
        mutex.unlock
      end
    end
end.each(&:join)

puts shared_array.size
```

This slight modification is guaranteed to produce the correct result, across Ruby implementations, every time. The only difference is the addition of the mutex.

```
$ ruby code/snippets/concurrent_array_pushing_with_mutex.rb
10000
$ jruby code/snippets/concurrent_array_pushing_with_mutex.rb
10000
$ rbx code/snippets/concurrent_array_pushing_with_mutex.rb
10000
```

Let's step through the changes.

Besides creating the mutex with `Mutex.new`, the important bits are the lines above and below the `<<`. Here they are again:

```
mutex.lock
shared_array << nil
mutex.unlock
```

The mutex sets up some boundaries for the thread. The first thread that hits this bit of code will lock the mutex. It then becomes the owner of that mutex.

Until the owning thread unlocks the mutex, no other thread can lock it, thus no other threads can enter that portion of code. This is where the guarantee comes from. The guarantee comes from the operating system itself, which backs the Mutex implementation[1].

In this program, since any thread has to lock the mutex before it can push to the Array, there's a guarantee that no two threads will be performing this operation at the same time. In other words, this operation can no longer be interrupted before it's completed. Once one thread begins pushing to the Array, no other threads will be able to enter that portion of code until the first thread is finished. This operation is now thread-safe.

Before I say anything else, I'll show a more commonly used convenience method that Ruby's `Mutex` offers:

```
mutex.synchronize do
  shared_array << nil
end
```

You pass a block to `Mutex#synchronize`. It will lock the mutex, call the block, and then unlock the mutex.

1. See pthread_mutex_init(3)

The contract

Notice that **the mutex is shared among all the threads**. The guarantee only works if the threads are sharing the *same* Mutex instance. In this way, when one thread locks a mutex, others have to wait for it to be unlocked.

I also want to point out that we didn't modify the `<<` method or the `shared_array` variable in any way in order to make this thread-safe.

So, if some other part of your code decides to do `shared_array << item`, but without wrapping it in the mutex, that may introduce a thread-safety issue. In other words, this contract is opt-in. There's nothing that stops some other part of the code from updating a value without the Mutex.

The block of code inside of a `Mutex#synchronize` call is often called a *critical section*, pointing to the fact that this code accesses a shared resource and must be handled correctly.

Making key operations atomic

In the last chapter we looked at multi-step operations, specifically a check-then-set race condition. You can use a mutex to make this kind of operation thread-safe, and we'll look at the effect of both updating values and reading values with a mutex.

We'll be reusing the ecommerce order example from the last chapter. Let's make sure that our customers are only charged once.

```ruby
# ./code/snippets/concurrent_payment_with_mutex.rb

# This class represents an ecommerce order
class Order
  attr_accessor :amount, :status

  def initialize(amount, status)
    @amount, @status = amount, status
    @mutex = Mutex.new
  end

  def pending?
    status == 'pending'
  end

  # Ensure that only one thread can collect payment at a time
  def collect_payment
    @mutex.synchronize do
      puts "Collecting payment..."
      self.status = 'paid'
    end
  end
end

# Create a pending order for $100
order = Order.new(100.00, 'pending')

# Ask 5 threads to check the status, and collect
# payment if it's 'pending'
5.times.map do
```

```
  Thread.new do
    if order.pending?
      order.collect_payment
    end
  end
end.each(&:join)
```

The change here is in the `Order#collect_payment` method. Each `Order` now has its own `Mutex`, initialized when the object is created. The Mutex creates a critical section around the collection of the payment. This guarantees that only one thread can collect payment at a time. Let's run this against our supported Rubies:

```
$ ruby code/snippets/concurrent_payment_with_mutex.rb
Collecting payment...
Collecting payment...
Collecting payment...

$ jruby code/snippets/concurrent_payment_with_mutex.rb
Collecting payment...
Collecting payment...
Collecting payment...
Collecting payment...
Collecting payment...

$ rbx code/snippets/concurrent_payment_with_mutex.rb
Collecting payment...
Collecting payment...
Collecting payment...
```

Huh? This didn't help at all. We're still seeing the incorrect result.

The reason is that we put the critical section in the wrong place. By the time the payment is being collected, multiple threads could already be past the 'check' phase of the 'check-and-set' race condition. In this way, you could have multiple threads queued up to collect payment, but they would do it one at a time.

We need to move the mutex up to a higher level so that both the 'check' AND the 'set,' in this case checking if the order is pending and collecting the payment, are inside the mutex.

Here's another go with the mutex in a different spot.

```ruby
# ./code/snippets/concurrent_payment_with_looser_mutex.rb

# This class represents an ecommerce order
class Order
  attr_accessor :amount, :status

  def initialize(amount, status)
    @amount, @status = amount, status
  end

  def pending?
    status == 'pending'
  end

  def collect_payment
    puts "Collecting payment..."
    self.status = 'paid'
  end
```

```
end

# Create a pending order for $100
order = Order.new(100.00, 'pending')
mutex = Mutex.new

# Ask 5 threads to check the status, and collect
# payment if it's 'pending'
5.times.map do
  Thread.new do
    mutex.synchronize do
      if order.pending?
        order.collect_payment
      end
    end
  end
end.each(&:join)
```

The change here is that the mutex is now used down inside the block passed to `Thread.new`. It's no longer present inside the `Order` object. The mutex is now a contract between the threads, rather than a contract enforced by the object.

But notice that the mutex now wraps the 'check' AND the `collect_payment` method call. This means that once one thread has entered that critical section, the others must wait for it to finish. So now you have a guarantee that only one thread will check that condition, find it to be true, and collect the payment. You have made this multi-step operation atomic.

Here are the results to prove it:

```
$ ruby code/snippets/concurrent_payment_with_looser_mutex.rb
Collecting payment...

$ jruby code/snippets/concurrent_payment_with_looser_mutex.rb
Collecting payment...

$ rbx code/snippets/concurrent_payment_with_looser_mutex.rb
Collecting payment...
```

Now the result is correct every time.

Mutexes and memory visibility

Now I want to throw another question at you. Should the same shared mutex be used when a thread tries to read the order status? In that case, another thread might end up using something like this:

```
status = mutex.synchronize { order.status }

if status == 'paid'
  # send shipping notification
end
```

If you're setting a variable while holding a mutex, and other threads want to see the most current value of that variable, they should also perform the read while holding that mutex.

The reason for this is due to low-level details. The kernel can cache in, for instance, L2 cache before it's visible in main memory. It's possible that after the status has been set to 'paid,' by one thread, another thread could *still* see the `Order#status` as 'pending' by reading the value from main memory before the change has propagated there.

This generally isn't a problem in a single-threaded program because race conditions like this don't happen with one thread. In a multi-threaded program there's no such guarantee. When one thread writes to memory, that operation may exist in cache before it's written to main memory. If another thread accesses that memory, it will get a stale value.

The solution to this is something called a **memory barrier**. Mutexes are implemented with memory barriers, such that when a mutex is locked, a memory barrier provides the proper memory visibility semantics.

Going back to our ecommerce example for a moment:

```
# With this line, it's possible that another thread
# updated the status already and this value is stale
status = order.status

# With this line, it's guaranteed that this value is
# consistent with any changes in other threads
status = mutex.synchronize { order.status }
```

Scenarios around memory visibility are difficult to understand and reason about. That's one reason other programming languages have defined something called a

memory model[2], a well-defined specification describing how and when changes to memory are visible in other threads.

Ruby has no such specification yet, so situations like this are tricky to reason about and may even yield different results with different runtimes. That being said, **mutexes carry an implicit memory barrier**. So, if one thread holds a mutex to write a value, other threads can lock the same mutex to read it and they *will* see the correct, most recent value.

Mutex performance

As with all things, mutexes provide a tradeoff. It may not be obvious from the previous section, but **mutexes inhibit parallelism.**

Critical sections of code that are protected by a mutex can only be executed by one thread at any given time. This is precisely why the GIL inhibits parallelism! The GIL is just a mutex, like the one you just saw, that wraps execution of Ruby code.

Mutexes provides safety where it's needed, but at the cost of performance. Only one thread can occupy a critical section at any given time. When you need that guarantee, the tradeoff makes sense. Nothing is more important than making sure that your data is not corrupted through race conditions, but you want to **restrict the critical section to be as small as possible, while still preserving the safety of your data**. This allows as much code as possible to execute in parallel.

---

2. Here is Java's memory model: http://www.cs.umd.edu/~pugh/java/memoryModel/, and here is the one for Golang: http://golang.org/ref/mem

Given this rule, which of the following two examples is more correct?

```ruby
# ./code/snippets/coarse_mutex.rb

require 'thread'
require 'net/http'

mutex = Mutex.new
@results = []

10.times.map do
  Thread.new do
    mutex.synchronize do
      response = Net::HTTP.get_response('dynamic.xkcd.com', '/random/comic/')
      random_comic_url = response['Location']

      @results << random_comic_url
    end
  end
end.each(&:join)

puts @results
```

```ruby
# ./code/snippets/fine_mutex.rb

require 'thread'
require 'net/http'

mutex = Mutex.new
```

```ruby
threads = []
results = []

10.times do
  threads << Thread.new do
    response = Net::HTTP.get_response('dynamic.xkcd.com', '/random/comic/')
    random_comic_url = response['Location']

    mutex.synchronize do
      results << random_comic_url
    end
  end
end

threads.each(&:join)
puts results
```

Did you get it? The second one is more correct because it uses a finer-grained mutex. It protects the shared resource (`@results`), while still allowing the rest of the code to execute in parallel.

The first example defines the whole block as the critical section. In that example, since the HTTP request is *inside* the critical section, it can no longer be performed in parallel. Once one thread locks that section, other threads have to wait for their turn.

A coarse mutex is unnecessary in this case because the HTTP request doesn't access anything shared between threads. Similarly, working with the `response` object doesn't need to be in a critical section. The `response` object is created right in the current thread's context, so we know it's not shared.

However, the line that modifies `@results` *does* need to be in a critical section. That's a shared variable that could be modified by multiple concurrent threads. If you had used Ruby's `Queue` instead of `Array`, it would have provided the thread-safety guarantees, and you wouldn't need the mutex here. But two threads modifying the same `Array` need to be synchronized.

In this case, the performance of the coarse-grained lock example will be similar to single-threaded performance. The performance of the fine-grained lock example is roughly 10x better given that the main bottleneck is the HTTP request.

The takeaway: **put as little code in your critical sections as possible, just enough to ensure that your data won't be modified by concurrent threads.**

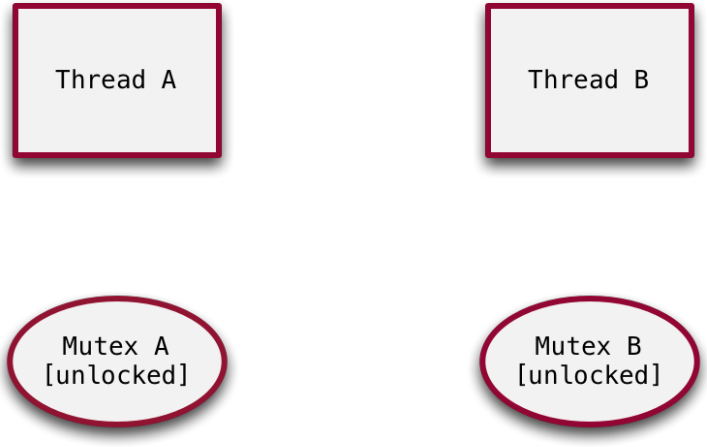
The dreaded deadlock

Deadlock is probably a term that you've encountered before. Whether or not it was in the context of multi-threading, the term probably carried some trepidation. It's got a deadly kind of sound to it, a bit like 'checkmate'. Deadlock isn't unlike checkmate, it essentially means 'game over' for the system in question.

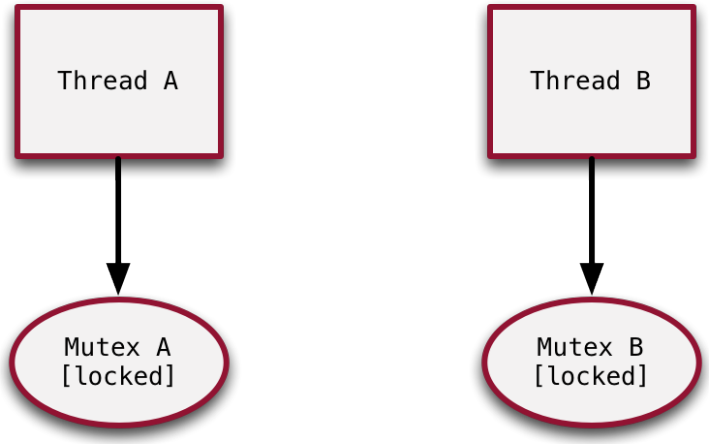
A deadlock occurs when one thread is blocked waiting for a resource from another thread (like blocking on a mutex), while this other thread is itself blocked waiting for a resource. So far, this is a normal situation, multiple threads are blocked waiting for mutexes, waiting for the GIL, etc. It becomes a deadlock if a situation arises where neither thread can move forward. In other words, the system comes to a place where no progress can happen.

The classic example of this involves two threads and two mutexes. Each thread acquires one mutex, then attempts to acquire the other. The first acquisition will go fine, then both threads will end up waiting forever for the other to release its mutex.

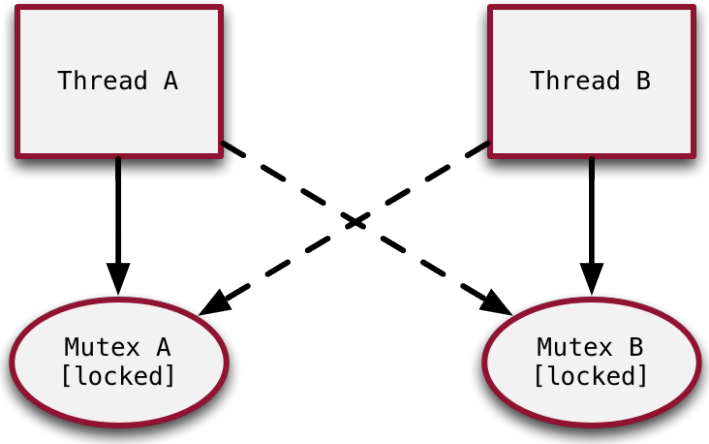
Let's walk through an example.



Here's our starting state, there are two threads and two mutexes.



Now each thread acquires one of the mutexes. Thread A acquired Mutex A, Thread B acquired Mutex B. Both mutexes are now in a locked state.



Now the deadlock occurs. Both threads attempt to acquire the other mutex. In both cases, since the mutex they're attempting to acquire is already locked, they're blocked until it becomes available. But since both threads are now blocked AND still holding their original mutexes, neither will ever wake up. This is a classic deadlock.

So, what can be done about this? I'll introduce two possible solutions. The first is based on a method of Mutex you haven't seen yet: `Mutex#try_lock`.

The `try_lock` method attempts to acquire the mutex, just like the `lock` method. But unlike `lock`, `try_lock` *will not* wait if the mutex isn't available. If another thread already owns the mutex, `try_lock` will return `false`. If it successfully acquires the mutex, `try_lock` will return `true`.

This can help in the deadlock situation seen above. One way to avoid deadlock at the last step would have been to use the non-blockingthe non-blocking `Mutex#try_lock`. That way the threads wouldn't be put to sleep. But that's not enough on its own: if both threads had just looped on `try_lock`, they still would never be able to acquire the other mutex. Rather, when `try_lock` returns `false`, both threads should release their mutexes and return to the starting state. From there, they can start again and hope for a better outcome.

With this approach, it's fine to use a blocking `lock` to get the first mutex, but `try_lock` should be used when acquiring the second mutex. If the second mutex is unavailable, that means another thread holds it and is probably trying to acquire the mutex you're holding! In that case, both mutexes should be unlocked for the dance to begin again. Thanks to the non-deterministic nature of things, one thread *should* be able acquire both mutexes at some point.

The downside to this approach is that another kind of issue can arise: **livelocking**. A **livelock is similar to a deadlock in that the system is not progressing**, but rather than threads stuck sleeping, they would be stuck in some loop with each other with none progressing.

You can see that this is possible using the `try_lock` solution I outlined here. It's possible that Thread A acquires Mutex A, Thread B acquires Mutex B, then both `try_lock` the other mutex, both fail, both unlock their first mutex, then both re-acquire their first mutex, etc., etc. In other words, it's possible that the threads are stuck in constant action, rather than stuck waiting. So there's a better solution.

A better solution is to define a mutex hierarchy. In other words, **any time that two threads both need to acquire multiple mutexes, make sure they do it in the same order.** This will avoid deadlock every time.

Harking back to our example with the diagrams, we already have hierarchical labeling of the mutexes. Mutex A is the obvious choice for a first mutex, and Mutex B the obvious choice for a second. If both threads need both, make sure they start with Mutex A. That way, if one thread acquires Mutex A, it's guaranteed to have no chance to deadlock on Mutex B. It can do the work it needs to do, then release both mutexes, at which point Thread B can lock Mutex A, and continue down the chain.

Deadlocks can wreak havoc on your system. If a thread is stuck in deadlock, it could be hogging a valuable resource, which blocks still other threads. Be aware of situations where a thread may block on more than one resource, and watch out for deadlocks any time that a thread needs to acquire more than one mutex.

Chapter 10

Signaling Threads with Condition Variables

Since you just were introduced to mutexes, now is the time to introduce condition variables. They don't solve the same problem, but they're inherently married to mutexes.

So what problem do they solve?

A `ConditionVariable` can be used to signal one (or many) threads when some event happens, or some state changes, whereas mutexes are a means of synchronizing access to resources. Condition variables provide an inter-thread control flow mechanism. For instance, if one thread should sleep until it receives some work to do, another thread can pass it some work, then signal it with a condition variable to keep it from having to constantly check for new input.

The API by example

I'll introduce the API via example. In the last chapter, there was an example that fetched a random comic URL from `xkcd.com`. Now you'll extend that to do something with that URL.

```
# ./code/snippets/xkcd_printer.rb
```

```ruby
require 'thread'
require 'net/http'

mutex     = Mutex.new
condvar   = ConditionVariable.new
results   = Array.new

Thread.new do
  10.times do
    response = Net::HTTP.get_response('dynamic.xkcd.com', '/random/comic/')
    random_comic_url = response['Location']

    mutex.synchronize do
      results << random_comic_url
      condvar.signal                          # Signal the ConditionVariable
    end
  end
end

comics_received = 0

until comics_received >= 10
  mutex.synchronize do
    while results.empty?
      condvar.wait(mutex)
    end

    url = results.shift
    puts "You should check out #{url}"
  end
```

```
    comics_received += 1
  end
```

Let's step through this bit by bit.

```
  mutex     = Mutex.new
  condvar   = ConditionVariable.new
  results   = Array.new
```

Here you create a new `ConditionVariable`, in much the same way that you create a `Mutex`. This instance, too, must be shared amongst threads in order to be useful.

```
  Thread.new do
    10.times do
      response = Net::HTTP.get_response('dynamic.xkcd.com', '/random/comic/')
      random_comic_url = response['Location']

      mutex.synchronize do
        results << random_comic_url
        condvar.signal                          # Signal the ConditionVariable
      end
    end
  end
```

Here we have one thread fetching a random comic from xkcd, ten times. Since it's just one thread, the HTTP requests are actually happening serially, one after another. This just puts that work in a background thread, so the main thread can continue on.

Once this thread receives a random comic URL, it locks the mutex, pushes the URL onto `results`, then signals `condvar`.

The `ConditionVariable#signal` method takes no parameters and has no meaningful return value. Its function is to wake up a thread that is waiting on itself. That happens lower down. If no threads are currently waiting, this is just a no-op.

```
comics_received = 0

until comics_received >= 10
  mutex.synchronize do
    while results.empty?
      condvar.wait(mutex)
    end
```

The background thread is fetching the random comics and, here, the main thread will be collecting the results.

First, some housekeeping to make it sure gets all 10 results.

The next part might seem a bit strange. The call to `Mutex#synchronize` will lock the mutex, then assuming that there are no `results` ready to be collected, `ConditionVariable#wait` is called. If `results` is not empty, it can be processed immediately without waiting.

At first glance, it seems that waiting on the condition variable while holding a mutex will block other threads from holding that mutex. But notice how you pass in the mutex to `ConditionVariable#wait`? It needs to receive a locked mutex.

`ConditionVariable#wait` will then unlock the Mutex and put the thread to sleep. When `condvar` is signaled, if this thread is first in line, it will wake up and lock the mutex again.

The last bit of strangeness here is the `while` loop. An `if` seems more natural here than a `while`. The reason for the `while` is a fail-safe. `ConditionVariable#wait` doesn't receive any state information; it just notifies that an event happened.

If this thread wakes up, but some other thread has already removed the result it was expecting from `results`, it could go forward and find nothing waiting there. So once it reclaims the mutex, it rechecks its condition. If `results` is empty again, it can just go back to waiting for the next signal.

```
    url = results.shift
    puts "You should check out #{url}"
  end

  comics_received += 1
```

This part wraps things up. Still inside the loop, it `shift`s the oldest element from the Array and prints it.

Then notice that the counter is incremented using `+=` *outside the mutex*. This is an instance where that operation doesn't need protection. That variable is only visible to one thread; hence no synchronization is needed.

Broadcast

There's one other part of this small API we didn't cover: broadcast.

There are two different methods that can signal threads:

1. `ConditionVariable#signal` will wake up *exactly one thread* that's waiting on this `ConditionVariable`.
2. `ConditionVariable#broadcast` will wake up *all threads* currently waiting on this `ConditionVariable`.

Chapter 11

Thread-safe Data Structures

Implementing a thread-safe, blocking queue

Now that you've got an understanding of mutexes and condition variables, you can use these concepts to build a quintessential thread-safe data structure: a blocking queue.

This will be something you can use like this:

```
q = BlockingQueue.new
q.push 'thing'
q.pop #=> 'thing'
```

That's a pretty simple API, but the other part of the contract is that you don't want users of `BlockingQueue` to have to use a mutex. Rather, it should be internally thread-safe.

The last requirement is that the `pop` operation, when called on an empty queue, should block until something is pushed onto the queue, to avoid busy code like this:

```
# this is a busy loop
loop do
  # constantly pop off the queue, but only
```

```ruby
  # process non-nil results
  if item = q.pop
    ...
  end
end

# this is what we want
loop do
  # when the queue pops something, you can be sure
  # that `item` contains a real, intended value
  item = q.pop
  ...
end
```

Let's start with a really simple base, working up incrementally.

```ruby
# ./code/blocking_queue/pre_mutex.rb

class BlockingQueue
  def initialize
    @storage = Array.new
  end

  def push(item)
    @storage.push(item)
  end

  def pop
    @storage.shift
```

```
    end
  end
```

This provides the framework for us to build on. Right away you should see a problem here.

Given that the underlying `Array` isn't thread-safe, and this class doesn't use a `Mutex`, the modifications happening in the `push` and `pop` methods will not protect the underlying `Array` from concurrent modfications.

A `Mutex` will rectify this.

```
# ./code/blocking_queue/pre_condvar.rb

require 'thread'

class BlockingQueue
  def initialize
    @storage = Array.new
    @mutex = Mutex.new
  end

  def push(item)
    @mutex.synchronize do
      @storage.push(item)
    end
  end

  def pop
```

```
      @mutex.synchronize do
        @storage.shift
      end
    end
  end
```

Now the `initialize` method creates a `Mutex` local to this object. There's no need for a global mutex here. Different instances of this class will provide their own thread-safety guarantees. So while one instance is pushing data into its `Array`, there's no problem with another instance pushing data into its `Array` concurrently. The issue only arises when the concurrent modification is happening on the same instance.

So, this is looking better. But you're missing the blocking `pop` behaviour. Currently a `nil` is returned from `pop` if the queue is empty. A `ConditionVariable` can rectify this.

```
# ./code/blocking_queue/complete.rb

require 'thread'

class BlockingQueue
  def initialize
    @storage = Array.new
    @mutex = Mutex.new
    @condvar = ConditionVariable.new
  end

  def push(item)
    @mutex.synchronize do
      @storage.push(item)
```

```
        @condvar.signal
      end
    end

    def pop
      @mutex.synchronize do
        while @storage.empty?
          @condvar.wait(@mutex)
        end

        @storage.shift
      end
    end
  end
```

This gets you the behaviour you need. We have one `ConditionVariable` object being shared among any threads that will be using this object instance.

When a thread calls the `pop` method, assuming that the underlying `Array` is empty, it calls `ConditionVariable#wait`, putting this thread to sleep. When the `push` method is called, it signals the condition variable, waking up one thread that's waiting on the condition variable to shift that item off of the `Array`.

Queue, from the standard lib

Ruby's standard library ships with a class called `Queue`. **This is the only thread-safe data structure that ships with Ruby**. It's part of the set of utilties that's loaded when you `require 'thread'`.

In this example, you came dangerously close to mirroring the implementation of `Queue`![1] `Queue` has a few more methods than your `BlockingQueue`, but its behaviour regarding `push` and `pop` is exactly the same.

`Queue` is very useful because of its blocking behaviour. Typically, you would use a `Queue` to distribute workloads to multiple threads, with one thread pushing to the queue, and multiple threads popping. The popping threads are put to sleep until there's some work for them to do.

```
# ./code/snippets/producer_consumer.rb

require 'thread'

queue = Queue.new

producer = Thread.new do
  10.times do
    queue.push(Time.now.to_i)
    sleep 1
  end
end

consumers = []

3.times do
  consumers << Thread.new do
    loop do
```

1. https://github.com/ruby/ruby/blob/ruby_1_9_3/lib/thread.rb#L140

```
        unix_timestamp = queue.pop
        formatted_timestamp = unix_timestamp.to_s.reverse.
                                  gsub(/(\d\d\d)/, '\1,').reverse

        puts "It's been #{formatted_timestamp} seconds since the epoch!"
      end
    end
  end

  producer.join
```

Array and Hash

`Queue` is useful, but sometimes you do need a regular ol' Array or Hash to get the job done. Unfortunately, **Ruby doesn't ship with any thread-safe Array or Hash implementations**.

The core `Array` and `Hash` classes are *not* thread-safe by default, nor should they be. Thread-safety concerns would add overhead to their implementation, which would hurt performance for single-threaded use cases.

You might be thinking: "With all of the great concurrency support available to Java on the JVM, surely the JRuby `Array` and `Hash` are thread-safe?" They're not. For the exact reason mentioned above, using a thread-safe data structure in a single-threaded context would reduce performance.

Indeed, even in a language like Java, these basic data structures *aren't* thread-safe. However, unlike Ruby, Java does have dependable, thread-safe alternatives bulit-in.

In Ruby, when you need a thread-safe Array or Hash, I suggest the `thread_safe` rubygem[2]. This gem provides thread-safe versions under its own namespace.

- `ThreadSafe::Array` can be used in place of `Array`.
- `ThreadSafe::Hash` can be used in place of `Hash`.

Note that these data structures aren't re-implementations; they actually wrap the core Array and Hash, ensuring that each method call is protected by a Mutex.

Immutable data structures

Immutable data structures are inherently thread-safe. Read more about them in the appendix on *Immutability*.

2. http://github.com/headius/thread_safe

Chapter 12

Writing Thread-safe Code

In this chapter, I'll give you a set of guidelines to keep in mind so that the code you write remains thread-safe.

It's presented as a list of things to watch out for. Next time you find yourself doing one of these things, think back to this chapter. Any guideline has exceptions, but it's good to know when you're breaking one, and why.

Let's start with the overarching principle on which this chapter is based:

> **Idiomatic Ruby code is most often thread-safe Ruby code.**

This might be obvious, but I point it out to let you know that there aren't any special tricks to learn. Writing good, idiomatic Ruby code will lead to thread safety most of the time. Obviously, the definition of 'good, idiomatic Ruby code' is up for debate, but bear with me!

Avoid mutating globals

If you'll recall from the chapter that introduced thread safety, 'concurrent modification' is the main thing that's going to lead to safety issues in a multi-threaded context.

The most obvious case where this will happen is when sharing objects between threads. Given this, global objects should stick out like a sore thumb! Global objects are implicitly shared between all threads.

So this is inherently not thread-safe:

```
$counter = 0
puts "Hey threads, go ahead and increment me at will!"
```

There are two things to keep in mind here:

1. Even globals can provide thread-safe guarantees.
2. Any time there is only one shared instance (aka. singleton), it's a global.

Let's tackle the first one first.

Even globals can provide thread-safe guarantees

That is to say, global variables don't necesarily *have* to be avoided. If, for some reason, you really *need* that global counter, you could do it like this:

```
# ./code/snippets/safe_global_counter.rb

require 'thread'

class Counter
```

```ruby
  def initialize
    @counter = 0
    @mutex = Mutex.new
  end

  def increment
    @mutex.synchronize do
      @counter += 1
    end
  end
end

$counter = Counter.new
```

It's a bit more code, but that's the price you pay to ensure that data consistency is preserved. It's worth it.

Anything where there is only one shared instance is a global

I bring this up because it's important to do more than just search for Ruby variables beginning with a dollar sign before you can cross this item off of your list.

There are other things that fit this definition in Ruby:

- Constants
- The AST
- Class variables/methods

These things don't look the same as global variables, but they're accessible from anywhere in your program, by any part of the code. Therefore, they're global too.

Just like storing a counter in a global variable (that has no thread-safety guarantee) is *not* safe, the same is true if you store that counter in a class variable or constant. So, look for those instances too.

This is OK, because it doesn't modify global state.

```
class RainyCloudFactory
  def self.generate
    cloud = Cloud.new
    cloud.rain!

    cloud
  end
end
```

This is *not* OK, because it does modify global state, in this case, a class variable.

```
class RainyCloudFactory
  def self.generate
    cloud = Cloud.new
    @@clouds << cloud

    cloud
  end
end
```

A slightly more nefarious example is the AST. By this I'm referring to the current set of program instructions that comprise your program. Ruby, being such a dynamic language, allows you to change this at runtime. I don't imagine this would be a common problem, but I saw it come up as an issue with the `kaminari`[1] rubygem. Some part of the code was defining a method dynamically, then calling `alias_method` with that method, then removing it.

There's only one AST, shared between all active threads, so you can imagine how this played out in a multi-threaded environment. One thread defined the method, then aliased it. Another thread then took over and defined its version of the method, overwriting the one that was already in place. The original thread then went and removed that method. When the second thread went to alias the method, it was no longer there. Boom. `NoMethodError`.

Again, this has to be a rare example, but it's good to keep in mind that **modifying the AST at runtime is almost always a bad idea**, especially when multiple threads are involved. When I say 'runtime', I mean during the course of the lifecycle of the application. In other words, it's expected that the AST will be modified at startup time, most Ruby libraries depend on this behaviour in some way. However, in the case of a Rails application, once it's been initialized, changes to the AST shouldn't happen at runtime, just as it's rare to `require` new files in the midst of a controller action.

So, if you're just reading from a global, that's fine. If there's a well-defined understanding of how to use a global, and it's protected from concurrent modification, that's fine. `Rails.logger` comes to mind as a good example of this. But if a global

1. https://github.com/amatsuda/kaminari/issues/214

seems like a convenient place to stash a shared object, make sure you think twice about that. It might not be the best place for it.

Create more objects, rather than sharing one

But sometimes you just need that global object. You really can't avoid it. This is especially problematic when the thread safety of that object is questionable.

The most common example of this is a network connection. I'm not thinking of a one-off HTTP request, but a long-lived connection to a database or external service.

This is problematic because a long-lived connection is a stateful connection. Typically, when talking to a database, your code will make a request, then wait for a response. The underlying socket has no notion of the state of the program, and the thread scheduler provides no guarantees about which thread will receive the data first.

So, this leaves us in a situation where the database client library needs to jump through a lot of hoops to make sure that the right thread receives the right result, or...

The simpler solution is to create more connections. There are two useful concepts that you could use for this:

1. Thread-locals
2. Connection pools

Thread-locals

This name is wild with contradiction, depending on your perspective. **A thread-local lets you define a variable that is global to the scope of the current thread**. In other words, it's a global variable that is locally scoped on a per-thread basis.

Here's how you might use it to provide each thread with its own connection to a Redis database:

```
# Instead of
$redis = Redis.new
# use
Thread.current[:redis] = Redis.new
```

Then you can use `Thread.current[:redis]` wherever you would otherwise have used `$redis`. This is a bit hard to grok the first time. Even though you only call `Redis.new` in one place, each thread will execute it independently. So, if your program is running N threads, it will open N connections to Redis.

This example showed a Redis connection, but this same concept can be applied to other objects, too. It's perfectly acceptable to tell users of your API that they should create one object for each thread, rather than trying to write difficult, thread-safe code that will increase your maintainenace costs.

This N:N connection mapping is fine for small numbers of threads, but gets out of hand when the number of threads starts to increase. For connections, a pool is often a better abstraction.

Resource pools

Going back to the Redis example, if you have N threads, you could use a pool of M connections to share among those threads, where M is less than N. **This still ensures that your threads aren't sharing a single connection, but doesn't require each thread to have its own.**

A pool object will open a number of connections, or in the more general sense, may allocate a number of any kind of resource that needs to be shared among threads. When a thread wants to make use of a connection, it asks the pool to check out a connection. The pool is responsible for keeping track of which connections are checked out and which are available, preserving thread safety. When the thread is done, it checks the connection back in to the pool.

Implementing a connection pool is a good exercise in thread-safe programming, you'll probably need to make use of both thread-locals and mutexes to do it safely. The `connection_pool` rubygem provides a nice, bare-bones implementation that's a good study in this area.[2]

Avoid lazy loading

A common idiom in Ruby on Rails applications is to lazily load constants at runtime, using something similar to Ruby's `autoload`.

2. http://github.com/mperham/connection_pool

For instance, if the `Geocoder` constant has not yet been loaded, the first time your application tries to execute `Geocoder.geocode(request.remote_ip)`, Rails will look for a file named `geocoder.rb`, require it, then return to your code.

This is a bad idea in the presence of multiple threads. The simple reason is that `autoload` in MRI is *not* thread-safe[3]. It is thread-safe in recent versions of JRuby, but the best practice is simply to eager load files before spawning worker threads.

This is done implicitly in Rails 4+, and can be enabled in Rails 3.x using the `config.threadsafe!`[4] configuration setting.

Prefer data structures over mutexes

Mutexes are notoriously hard to use correctly. For better or worse, you have a lot of things to decide when using a mutex.

- How coarse or fine should this mutex be?
- Which lines of code need to be in the critical section?
- Is a deadlock possible here?
- Do I need a per-instance mutex? Or a global one?

3. http://bugs.ruby-lang.org/issues/921
4. http://apidock.com/rails/Rails/Application/Configuration/threadsafe%21

This is just a sampling of questions you need to answer when you decide to use a mutex. For a programmer familiar with mutexes and with deep knowledge of the problem domain, these questions may be easy to answer.

However, in many cases, they're not. Sometimes the rules of a system are not well-defined, programmers working on the project are not fluent with the usage of mutexes, or a plethora of other reasons.

Using a data structure removes a lot of these concerns. Rather than worrying about where to put the mutex, whether or not it can deadlock, **you simply don't need to create any mutexes in your code**.

By leaning on a data structure, you remove the burden of correct synchronization from your code and depend on the semantics of the data structure to keep things consistent.

This only works if you choose not to share objects between threads directly. Rather than letting threads access shared objects and implementing the necessary synchronization, you pass shared objects through data structures. This ensures that only one thread could mutate an object at any given time.

Finding bugs

There may be times where your program is exhibiting strange behaviour. Despite following all the best practices, a thread-safety bug may have slipped in, or this bug

may be traced to someone else's code. Unfortunately, this kind of debugging can often feel like looking for a needle in a haystack.

Like most bugs, if you can reproduce the issue, you can almost certainly track it down and fix it. However, some thread-safety issues may appear in production under heavy load, but can't be reproduced locally. In this case, there's no better solution than grokking the code.

Now you've seen common problems, notably global references. That's the best place to start. Look at the code and assume that 2 threads will be accessing it simulatneously. Step through the possible scenarios. It can be helpful to jot these things down somewhere. With some practice, this process becomes more natural and these patterns will jump out at you more readily.

Chapter 13

Thread-safety on Rails

Since many of us Rubyists work on Rails applications on a regular basis, you should know how to keep your Rails apps thread-safe.

It's rare that you would spawn threads inside your application logic, but if you're using a multi-threaded web server (like Puma[1]) or a multi-threaded background job processor (like Sidekiq[2]), then your application is running in a multi-threaded context.

The last chapter said that good, idiomatic Ruby code is usually thread-safe Ruby code. The same goes for Rails applications. If you stick to Rails conventions, and write idiomatic Rails code, your Rails application will be thread-safe.

Gem dependencies

Outside of your application code, you'll have to make sure that any gems you depend on are thread-safe. Since there's no way to programmatically verify thread safety, you'll have to do your research. The majority of gems in the community are thread-safe, but check the bug tracker of any gem you're thinking of adding, just to be sure.

1. http://puma.io
2. http://sidekiq.org

The request is the boundary

The most important bit of information you need in order to understand thread safety in web applications is how threads are being used by the underlying web server or background job processor.

For the web server, the web request is the dividing line between threads. By this I mean that a multi-threaded web server will process each request with a separate thread.

If you know that each request is handled by a separate thread, then the path to thread safety is clear: **don't share objects between requests**. This will ensure that no objects are shared between threads.

To put it even more simply, make sure that each controller action creates the objects that it needs to work with, rather than sharing them between requests via a global reference.

A good example of this is something like a `User.current` reference. All Rails applications have some way of referring to the current user in a controller action. Typically this is done using an instance variable. This is safe because each request creates its own instance of the controller stack; variables aren't shared. However, if you store a `User` object in `User.current`, that's a class variable that *is* shared among threads.

So one thread will assign a `user` to `User.current`, then another thread can come along and overwrite it. The first thread expects to find its user reference still there

when performing database lookups, but it's been overwritte. Now your users end up seeing each others data instead of their own.

If you really need a global reference, follow the guidelines from the last chapter. Try using a thread-local, or else a thread-aware object that will preserve data correctness.

The same heuristic is applicable to a background job processor. Each job will be handled by a separate thread. A thread may process multiple jobs in its lifetime, but a job will only be processed by a single thread in its lifecycle.

Again, the path to thread safety is clear: create the necessary objects that you need in the body of the job, rather than sharing any global state.

Following these guidelines will ensure that your Rails application code has no safety issues in a multi-threaded context.

Chapter 14

Wrap Your Threads in an Abstraction

Up until now, all of the example code in the book has been of a trivial nature, stuff that you might write in a simple script to accomplish a task on your local machine, but probably wouldn't imagine putting in a production situation. Now I want to turn you toward the real world.

This chapter will help you put threads at the right level of abstraction in your code, rather than having them mixed up alongside your domain logic.

Single level of abstraction

In the book Clean Code,[1] Robert Martin laid out principles for improving code readability and maintanability. One of the principles that's relevant to this discussion is *"one level of abstraction per function."* It states that all code in a given function should be at the same level of abstraction; i.e., high-level code and low-level code don't mix well.

What constitutes high-level code and low-level code is certainly up for debate, but I would label code that works with threads, mutexes, and the like as low-level code. It is almost certainly a lower level of abstraction than your domain logic.

1. http://amzn.com/0132350882

Let me use an analogy to make my point.

When you use a database adapter (like mysql2 or redis-rb), do you ever see the `Socket` constant? No. All of these adapters are using a socket internally for communication, but that's at a different level of abstraction than writing SQL statements.

The database adapters extract out the complexities of working with sockets, handing network exceptions, buffering, etc. The same is true when working with threads.

Threads are also a low-level concept with its own complexities. It's something provided by your operating system, and is not likely part of the main domain logic of your application.

Given this, it's best to wrap threads in an abstraction *where possible*. In certain circumstances your domain logic might need direct access to a `Mutex` or might need to spawn threads very deliberately. This isn't a hard and fast rule, just a guideline.

Let's start with a simple case. Way back at the beginning of the book we looked at a trivial `FileUploader` class. Here it is, to refresh your memory.

```
# ./code/snippets/file_uploader.rb

require 'thread'

class FileUploader
  def initialize(files)
```

```
    @files = files
  end

  def upload
    threads = []

    @files.each do |(filename, file_data)|
      threads << Thread.new {
        status = upload_to_s3(filename, file_data)
        results << status
      }
    end

    threads.each(&:join)
  end

  def results
    @results ||= Queue.new
  end

  def upload_to_s3(filename, file)
    # omitted
  end
end

uploader = FileUploader.new('boots.png' => '*pretend png data*', 'shirts.png' => '*pretend png data*')
uploader.upload

puts uploader.results.size
```

I even made a case way back at the start of the book that the `Thread.new` here sticks out like a sore thumb. It's quite obviously at a different level of abstraction than the rest of the code. Let's extract that to somewhere else so that this code can, once again, focus on its intent.

```
# ./code/snippets/concurrent_each.rb

module Enumerable
  def concurrent_each
    threads = []

    each do |element|
      threads << Thread.new {
        yield element
      }
    end

    threads.each(&:join)
  end
end
```

This is a simple wrapper around `Enumerable#each` that will spawn a thread for each element being iterated over. It wouldn't be wise to use this code in production yet because it has no upper bound on the number of threads it will spawn. Give it an `Array` with 100,000 elements and it will attempt to spawn 100,000 threads!

A better approach here would be to spawn a fixed-size pool of threads, like Puma](#puma-s-thread-pool-implementation) does, then keep passing work to those threads as they can handle it.

Now we can rewrite the `FileUploader` class to make use of this new method, still preserving its behaviour, but no longer mixing levels of abstraction.

```
# ./code/snippets/concurrent_file_uploader.rb

require 'thread'
require_relative 'concurrent_each'

class FileUploader
  attr_reader :results

  def initialize(files)
    @files = files
    @results = Queue.new
  end

  def upload
    @files.concurrent_each do |(filename, file_data)|
      status = upload_to_s3(filename, file_data)
      @results << status
    end
  end

  def upload_to_s3(filename, file)
    # omitted
```

```
    end
  end

  uploader = FileUploader.new('boots.png' => '*pretend png data*', 'shirts.png' => '*pretend png data*')
  uploader.upload

  puts uploader.results.size
```

This is much easier to grok at a glance, with the concurrency details handled now by `ConcurrentEach`.

Actor model

In some cases, it will make sense for you to write your own simple abstractions on top of your multi-threaded code, as in the above example. In other cases, especially when multi-threading concerns are at the core of your program (think servers, networking, etc.), you'll get more benefit from a more mature abstraction.

The Actor model is a well-known and proven technique for doing multi-threaded concurrency. As an abstraction, it's much more mature than the simple `map` you did in the last section.

At a high level, an Actor is a long-lived 'entity' that communicates by sending messages. When I say long-lived entity, I'm talking about a long-running thread, not something that's spawned ad-hoc. In the Actor model, each Actor has an 'address'. If you know the address of an Actor, you can send it a message.

These messages go to the Actor's mailbox, where they're processed asynchronously when the Actor gets around to it.

This describes the core of the Actor model at a high level, but let's start talking about the implementation. There's obviously more than one way that this kind of system can be implemented. Indeed, there are several implementations available in the Ruby community[2]. For the rest of this chapter, we'll be looking at Celluloid.[3]

What sets Celluloid apart is that it takes this conceptual idea of the Actor model and marries it to Ruby's object model. Let's see the basic operations of a Celluloid actor, and how they map to the overview I gave above.

```
# ./code/snippets/celluloid_xkcd_printer.rb

require 'celluloid/autostart'
require 'net/http'

class XKCDFetcher
  include Celluloid

  def next
    response = Net::HTTP.get_response('dynamic.xkcd.com', '/random/comic/')
    random_comic_url = response['Location']

    random_comic_url
```

2. Besides Celluloid, rubinius-actor is probably the second most used.
3. http://celluloid.io

```
  end
end
```

The only real detail that separates this class from a regular Ruby class is the inclusion of the `Celluloid` module.

```
class XKCDFetcher
  include Celluloid
```

Including the `Celluloid` module into any Ruby class will turn instances of that class into full-fledged Celluloid actors. That's all it takes.

From now on, any time that a new instance of `XKCDFetcher` is created, it will be wrapped by a Celluloid actor. Each Celluloid actor is housed by a separate thread, one thread per actor.

When you create a new actor, you immediately know its 'address'. So long as you hold a reference to that object, you can send it messages. In Celluloid, sending messages to an actor equates to calling methods on an object. However, Celluloid does preserve the behaviour of regular method calls, so it still feels like a regular Ruby object.

```
# this spawns a new thread containing a Celluloid actor
fetcher = XKCDFetcher.new

# these behave like regular method calls
fetcher.object_id
fetcher.inspect

# this will fire the `next` method without
```

```
# waiting for its result
fetcher.async.next
fetcher.async.next
```

Regular method calls *are* sent to the actor's mailbox but, behind the scenes, Celluloid will block the caller until it receives the result, just like a regular method call.

However, Celluloid also allows you to send a message to the actor's mailbox without waiting for the result. You can see this in the example above using the `async` syntax. In this way, you can asynchronously fire off some work, then continue execution.

The `async` is good if you don't care about the result, but what if you want to fire off a method asynchronously *and* get its result? That's what Celluloid futures accomplish, and that rounds out this code example.

```
fetcher = XKCDFetcher.new
futures = []

10.times do
  futures << fetcher.future.next
end

futures.each do |future|
  puts "You should check out #{future.value}"
end
```

This example begins by spawning a new actor. Then the `next` method is called 10 times using the `future` syntax. First, call the `future` method on an actor, then the message you want to send, which must correspond to a public method on the actor.

Celluloid kicks off that method asynchronously and returns you a `Celluloid::Future` object.

Calling `#value` on that future object will block until the value has been computed. In this example, 10 fetches to xkcd are kicked off asynchronously, then the results are all collected using the call to `value`.

This is the basic idea behind Celluloid. It actually has a lot more features,[4] but this concept of 'objects as actors' is really at its core.

Recall, for any class that includes the `Celluloid` module, instances of that class will each be independent actors. Actors are independent by virtue of being in their own thread, so there's one thread per actor. Passing a message to an actor is as simple as calling one of its public methods. The behaviour associated with any particular message is simply the body of the method.

Celluloid is a great solution to concurrency that puts the abstraction at the right level and wraps up a lot of best practices. The next chapter will take a deeper look at real-world Celluloid project.

4. Seriously, it has a lot of useful features built-in. The wiki has great coverage of what it offers.

Chapter 15

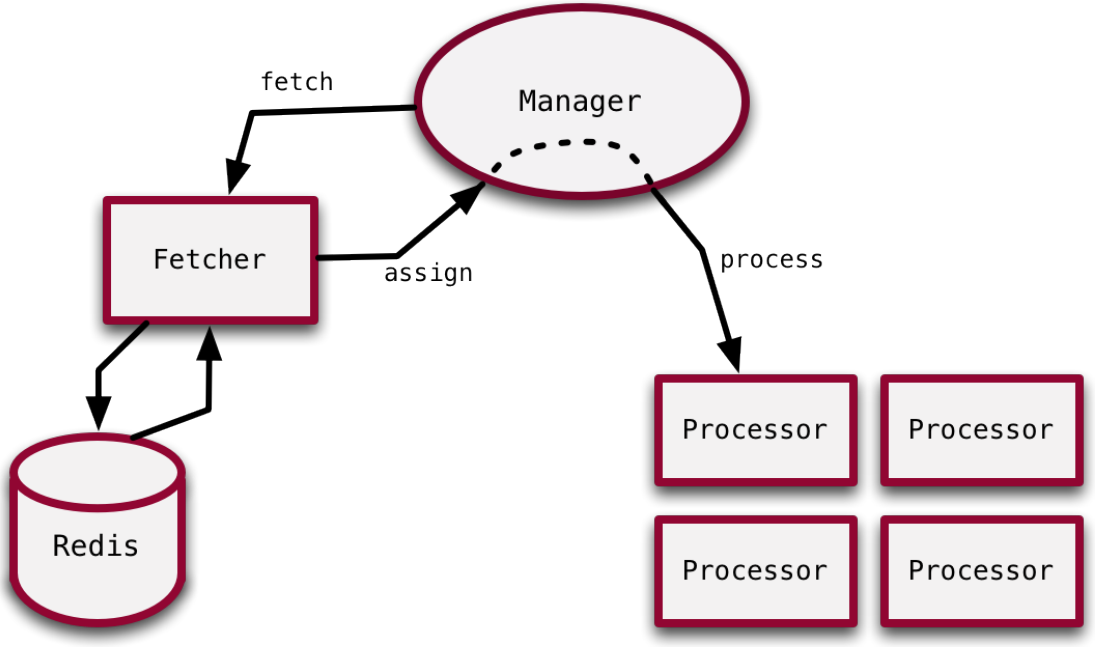
How Sidekiq Uses Celluloid

Sidekiq is a multi-threaded background job processing system backed by Redis. It's multi-threaded, but if you take a close look through the source code, you won't see many of the constructs you've learned in this book. No `Thread.new` or `Mutex#synchonize`. This is because all of Sidekiq's multi-threaded processing is implemented on top of Celluloid.

This is for good reason. Mike Perham, author of Sidekiq, suggests:

> *As soon as you introduce the* `Thread` *constant, you've probably just introduced 5 new bugs into your code.*

Sidekiq is a system composed of Celluloid actors. At the top level, there's a Manager actor that holds the state of the whole system and mediates between the collaborator actors. The collaborator actors are the Fetcher, which fetches new jobs from Redis, and the Processors, which perform the actual work of the jobs.



This is a rough sketch of Sidekiq's actor architecture. Specifically, we're going to focus on how the `fetch`, `assign`, and `process` messages are handled.

Into the source

Normally, when walking through code like this, I would focus on a particular class or a particular method. But given the nature of Sidekiq's architecture, and that of most actor-based systems, the building block isn't classes or methods, it's messages. So the focus will be on messages. That being said, in a Celluloid actor, messages and their associated behaviour are defined as Ruby methods, so you *will* be looking at methods, but they won't be used strictly in the traditional sense.

Our journey starts in the Manager actor.

fetch

Once the manager has been initialized, it's started with the `start` method.

```
def start
  @ready.each { dispatch }
end

def dispatch
  return if stopped?
  # This is a safety check to ensure we haven't leaked
  # processors somehow.
  raise "BUG: No processors, cannot continue!" if @ready.empty? && @busy.empty?
  raise "No ready processor!?" if @ready.empty?
```

```
    @fetcher.async.fetch
  end
```

In the Manager, the `@ready` variable holds references to the Processor actors that are ready to process jobs. So it calls the `dispatch` method once for each idle Processor. The `dispatch` method does some housekeeping at the beginning, but the last line is the important one.

The `async` method used on the last line will send the `fetch` message to the `@fetcher` actor and *will not* wait for the return value. It's essentially is a fire-and-forget call where the message is sent, but the Manager doesn't wait for the response.

Since the `fetch` message is asynchronous, it can be sent multiple times and queued in the Fetcher actor's mailbox until it can process the backlog. This is expected behaviour for Sidekiq. For instance, with 25 Processor actors, the `fetch` message will be sent 25 times. The Fetcher actor will process each `fetch` in turn, as new jobs get pushed into Redis.

This is an excerpt from the `Fetcher#fetch` method.

```
work = @strategy.retrieve_work

if work
  @mgr.async.assign(work)
else
  after(0) { fetch }
end
```

The Fetcher first tries to retrieve a unit of work. You can see that if it doesn't retrieve any work, it calls itself again. This, again, points to the asynchronous nature of sending messages. The Manager sent the `fetch` message, but since it's not waiting for a return value, the Fetcher is free to take as long as necessary before sending a message back to the Manager.

When it's finally able to retrieve work, it asynchronously sends the `assign` message to the Manager, passing along the unit of work.

assign

Here's the `Manager#assign` method that receives that unit of work.

```
def assign(work)
  watchdog("Manager#assign died") do
    if stopped?
      # Race condition between Manager#stop if Fetcher
      # is blocked on redis and gets a message after
      # all the ready Processors have been stopped.
      # Push the message back to redis.
      work.requeue
    else
      processor = @ready.pop
      @in_progress[processor.object_id] = work
      @busy << processor
      processor.async.process(work)
    end
```

```
    end
  end
```

This method, too, has some housekeeping at the beginning, such as is real-world code after all. The `else` block contains the real business logic.

First, the Manager grabs the next available Processor, then keeps tracks of its status appropriately in its internal data structures.

Notice that the Manager is using plain Ruby arrays here with `@ready` and `@busy`. Even though they're used in a multi-threaded context, there's no thread-safety concern here. This Manager actor lives in its own thread, it own these instance variables, and doesn't share them with other actors. Since the variables are contained within this thread, there can be no issue with multiple threads interacting with them concurrently.

On the last line there's another asynchronous message sent, the chosen Processor receives the `process` message along with the unit of work. Again, this will be a fire-and-forget style of message, with the Manager not waiting for a response. Instead, the Processor will perform the work, then send a message back to the Manager when it's finished.

I won't share the definition of the `process` method because it's almost entirely focused on the actual performing of the job, there's very little fodder in terms of multi-threading primitives.

Wrap-up

You saw how Sidekiq handled the `fetch`, `assign`, and `process` messages. This paradigm should feel a bit different from traditional Ruby code. In traditional Ruby code you might achieve the same behaviour with something like the following:

```
class Manager
  def dispatch
    loop do
      work = @fetcher.fetch

      result = processor_pool.process(work)
      log_result(result)
    end
  end
  # ...
end
```

The most obvious difference I see between the Sidekiq codebase and a more traditional Ruby codebase is the lack of dependence upon return values. In my example above there's one method that calls a bunch of others, collecting return values and passing them around.

When sending messages in Sidekiq, return values are seldom used. Instead, when an actor sends a message, they expect a message to be sent back to them in return. This keeps things asynchronous. Besides this stylistic difference, the Sidekiq codebase is remarkably free of threading primitives. This is due to how well Celluloid respects Ruby's object system, as we explore in the last chapter.

Sidekiq is a great example of how simple it can be to integrate multi-threaded concurrency, via actors, with your business logic.

Chapter 16

Puma's Thread Pool Implementation

Puma[1] is a concurrent web server for Rack apps that uses multiple threads for concurrency. Other popular web servers in the community are backed by multiple processes, or by a single-threaded event loop, so Puma is really the front-runner when it comes to multi-threaded servers.

At Puma's multi-threaded core is a thread pool implementation. I'm going to walk you through the main threaded logic so you can see how a real-world, battle-tested project handles things.

A what now?

Puma's thread pool is responsible for spawning the worker threads and feeding them work. The thread pool wraps up all of the multi-threaded concerns so that the rest of the server is just concerned with networking and the actual domain logic.

The `Puma::ThreadPool` is actually a totally generic class, not Puma specific. This makes it a good study, and it could potentially be extracted into something generally useful.

Once initialized, the pool is responsible for receiving work and feeding it to an available worker thread. The ThreadPool also has an auto-trimming feature, whereby

1. http://puma.io

the number of active threads is kept to a minimum, but more threads can be spawned during times of high load. Afterwards, the thread pool would be trimmed down to the minimum again. *Note: I edited the example methods slightly to remove this logic, as it didn't add anything to the discussion.*

The whole thing

Here's the whole implementation of the `Puma::ThreadPool#spawn_thread` method. This gets called once for each worker thread to be spawned for the pool. I'll walk through it section by section.

Just for reference, one instance of this class will spawn many threads. Much like in our examples, instance variables are shared among multiple threads.

```
def spawn_thread
  @spawned += 1

  th = Thread.new do
    todo  = @todo
    block = @block
    mutex = @mutex
    cond  = @cond

    extra = @extra.map { |i| i.new }

    while true
      work = nil
      continue = true
```

```
        mutex.synchronize do
          while todo.empty?
            if @shutdown
              continue = false
              break
            end

            @waiting += 1
            cond.wait mutex
            @waiting -= 1
          end

          work = todo.pop if continue
        end

        break unless continue

        block.call(work, *extra)
      end

      mutex.synchronize do
        @spawned -= 1
        @workers.delete th
      end
    end

    @workers << th

    th
  end
```

In bits

Now you'll see it bit by bit.

```
def spawn_thread
  @spawned += 1
```

I want to highlight this very first line in the method because it illustrates an important point that I mentioned in the chapter on Mutexes. This `@spawned` instance variable is shared among all the active threads and, as you know, this `+=` operation is not thread-safe! From what we can see, there's no mutex being used. What gives?

In the source file, there's this very important comment right above this method:

```
# Must be called with @mutex held!
```

This is a great example of mutexes being opt-in. This method must be called with the shared `@mutex` held, but it doesn't do any internal checking, so it would be possible to call this method without a mutex, potentially corrupting the value of `@spawned`.

Just a good reminder that mutexes only work if callers respect the implicit contract it offers. Moving on.

```
th = Thread.new do
  todo  = @todo
  block = @block
  mutex = @mutex
  cond  = @cond
```

```
extra = @extra.map { |i| i.new }
```

The first line here spawns a thread that will become part of the pool. This is only part of the block that's passed to `Thread.new`.

At first glance, this bit of code looks like it might be assigning local variables so as not to share references with other threads. If each thread needed to re-assign its mutex, for instance, it would want to switch to a local reference so as not to affect other threads.

But the git blame for this bit of code suggests otherwise.[2] Since this is a hot code path for Puma, using local variables will slightly improve performance over using instance variables. The references are never re-assigned by the individual threads, and this does nothing to prevent the threads sharing references. In this case, the threads *must* share the reference to the mutex and condition variable in order for their guarantees to hold.

These kinds of optimizations are common for web servers, but rare for application logic.

```
while true
  work = nil
  continue = true
```

Now we get into the real meat of this method.

2. https://github.com/puma/puma/commit/fb4e23d628ad77c7978b67625d0da0e5b41fd124

The first line enters an endless loop. So this thread will execute forever, until it hits its exit condition further down. We'll see the `work` and `continue` variables further down. They're just initialized here.

```
mutex.synchronize do
  while todo.empty?
    if @shutdown
      continue = false
      break
    end

    @waiting += 1
    cond.wait mutex
    @waiting -= 1
  end

  work = todo.pop if continue
end
```

OK, that's a big paste. I'll highlight some of the outer constructs, then re-focus on the inner stuff.

First, all of the code in this block happens inside of the `mutex.synchronize` call. So other threads have to wait while the current thread executes this block.

```
while todo.empty?
  if @shutdown
    continue = false
    break
```

```
      end

      @waiting += 1
      cond.wait mutex
      @waiting -= 1
    end
```

This little block of code came straight out of the one earlier, you're still inside the mutex here. This block only runs if `todo` is empty. `todo` is a shared array that holds work to be done. If it's empty, that means there's not currently any work to do.

If there's no work to do, this worker thread will check to see if it should shut down. In that case it will set that `continue` variable to `false` and break out of this inner `while` loop.

If it doesn't need to shut down, things get more interesting.

First, it increments a global counter saying that it's going to wait. This operation is safe because the shared mutex is still locked here. Next, it waits on the shared condition variable. Remember that this releases the mutex and puts the current thread to sleep. It won't be woken up again until there's some work to do. Since it released the shared mutex, another thread can go through the same routine.

Also notice that a `while` loop is used as the outer construct here, rather than an `if` statement. Remember that when once signaled by a condition variable, the condition should be re-checked to ensure that another thread hasn't already processed the work.

Once enough work arrives, this thread will get woken up. As part of being signaled by the condition variable, it will re-acquire the shared mutex, which once again makes it safe to decrement the global counter.

```
    work = todo.pop if continue
```

Now the thread has been awoken, re-acquired the mutex, and found `todo` to contain some work, it pops the unit of work from `todo`. This is the last bit of code still inside the mutex.synchronize block.

```
      break unless continue

      block.call(work, *extra)
    end
```

This little bit of code is outside the mutex.synchronize block, but now outside the `while` loop around the condition variable. If it's time to shut down, this thread will need to break out of its outer infinite loop. This accomplishes that.

If it's not time to shut down, then this worker thread can process the work to do. In this case, it simply calls the shared `block` with the `work` object that it received. The `block` is passed in to the constructor and is the block of code that each worker thread will perform.

```
      mutex.synchronize do
        @spawned -= 1
        @workers.delete th
```

```
    end
  end
```

The body of the thread ends with a little housekeeping. Once the thread leaves its infinite loop, it needs to re-acquire the mutex to remove its reference from some shared variables.

```
    @workers << th

    th
```

The last two lines are outside the scope of the block passed to `Thread.new`. So they'll execute immediately after the thread is spawned. And remember, even here the mutex is held by the caller of this method!

Here the current thread is added to `@workers`, then returned.

Wrap-up

This implementation nicely illustrates many of the concepts that were covered in this book. And as far as abstractions go, Puma does a superb job of isolating the concurrency-primitive logic from the actual domain logic of the server. I definitely reccomend checking out how the `ThreadPool` is used in Puma, and the lack of threading primitives through the rest of the codebase.

Simiarly, I encourage you to check out the other methods in the `ThreadPool` class, tracing the flow from initialization, to work units being added to the thread pool, to work units being processed from the thread pool, all the way to shutdown.

Chapter 17

Closing

Working with concurrency is all about organization.

If you want to take advantage of 100% of what your system offers, you need concurrency. If you have a single-threaded program, and aren't using some other form of concurrency (like processes), then you're only using a fraction of what your system offers.

Our job is to organize our code such that the system can run it in the most efficient way possible. Introducing multiple threads is one way to do this. But then we need to make sure that our code is organized to preserve thread safety.

Working with concurrency is about balancing these elements of organization: organizing our code so that it can take maximum advantage of system resources, while still preserving the underlying data.

Part of the point I'm making is that concurrency isn't something that should be used everywhere, not in every program, and not in every part of your application. Part of organizing our code is to decide where to apply concurrency, and where to restrict it.

The trick with multi-threaded programming is to strike the balance between firing on all cylinders all of the time, and utilizing more than a fraction of what your system has to offer.

Both of these extremes can be painful, so we should seek the place in the middle. The most succint set of rules I've seen to find this middle ground are laid out eloquently on the JRuby wiki.[1]

The safest path to concurrency:

1. Don't do it.
2. If you must do it, don't share data across threads.
3. If you must share data across threads, don't share mutable data.
4. If you must share mutable data across threads, synchronize access to that data.

If you stick to these rules, you'll strike that balance.

Ruby concurrency doesn't suck

I think there's a general vibe in the programming community that Ruby isn't a suitable environment for concurrency. I'll certainly agree that this was the case in the past.

When MRI was the only Ruby implementation, using green threads instead of native threads along with a GIL, Ruby did not have a good story when it came to concurrency; the only option was to start more processes. Indeed, that's still the approach favoured by a lot of Ruby applications to this day, and it does work.

---

1. https://github.com/jruby/jruby/wiki/Concurrency-in-jruby#wiki-concurrency_basics

But Ruby's concurrency story is much better today. Threading on all of the major Ruby implementations is backed by native threads. While MRI still has a GIL, there are alternatives (JRuby and Rubinius) that support true parallel threading.

The Ruby language still doesn't ship with much support for multi-threaded concurrency, but there are a growing number of options available in the community. Celluloid is a great example of this. It simplifies concurrent programming, while really embracing the constructs of the language. Writing programs using `Thread.new` and `Mutex#synchronize` can be very challenging and lead to poorly factored Ruby code. The same program written using Celluloid can provide the same concurrency, but feel more like idiomatic Ruby.

As a language, Ruby could provide better primitives to support multi-threaded concurrency.[2] But even today, Ruby doesn't suck for concurrency. And I think Ruby's concurrent future looks even brighter.

2. Brian Shirai, from Rubinius, made some great points about this when I interviewed him.

Chapter 18

Appendix: Atomic Compare-and-set Operations

Note on appendices: these chapters are included as appendices because these techniques, for all their benefits, don't yet have support from Ruby core.

I didn't mention it previously, but there are really two camps in the multi-threaded world. It's not as though these two camps oppose each other, but there are very different ways to structure concurrent programs.

The 'old guard' tends to lean on the synchronization mechanisms that you've already seen. That means lots of mutexes and condition variables. These methods are tried and true, but there are obvious drawbacks. Things like deadlocks and mutex bottlenecks are just two examples.

The newer camp is emerging from academia, offering a different approach to writing concurrent programs. Although many of these concepts are still theoretical in nature, many of them are seeing more widespread use and experimentation. These are things like lockless, lock-free, and wait-free data structures, immutable data structures, software transactional memory, and things of this nature.

Covering all of these emerging topics would require a book in and of itself, and many of them are still stabilizing as they see real-world usage. In this chapter I'll look at atomic compare-and-set (CAS) operations, which is really the building block underlying

these newer approaches. In the next chapter I'll give more coverage to immutability and how it relates to thread safety.

Overview

Here's a brief outline of how the CAS operation works.

In Ruby, you first create an `Atomic` instance that holds a value.

```
item = Atomic.new('value')
```

This comes from the 'atomic' rubygem. This gem provides a consistent Ruby API for CAS operations across MRI, JRuby, and Rubinius[1]. It's great!

The `Atomic` API does provide a higher-level, Ruby-ish API, but the lower-level workhorse method is `Atomic#compare_and_set(old, new)`. You pass it the value that you *think* it currently holds and the value you want to set it to.

If the current value *is* the one you expected it to be, it's updated to your new value. Otherwise, it returns false. At this point you have to decide whether to retry the operation or just continue on.

This compare-and-set operation is hardware-supported and provides a guarantee you wouldn't have otherwise. In the next section, I'll illustrate this with code. In a situation where you want to read a value, modify it, and then update it, it's possible to read a

1. Rubinius is the only implementation that includes a native Ruby implementation. It's called AtomicReference.

value, then have another thread update the value before you can, leading to a situation where your update can overwrite the work done by another thread.

Code-driven example

A few chapters back, you had a quick look at the `+=` operator. Although this is an operator in Ruby, it offers no thread-safety guarantees since it will be expanded to at least 3 operations under the hood. Let's review that.

When you use the `+=` operator like this: `@counter += 1`, Ruby must do at least the following:

1. Get the current value of `@counter`.
2. Increment the retrieved value by 1.
3. Assign the new value back to `@counter`.

It looks roughly like this in code:

```
@counter = 0

# Get the current value of `@counter`.
current_value = @counter

# Increment the retrieved value by 1.
new_value = current_value + 1
```

```
# Assign the new value back to `@counter`.
@counter = new_value
```

You probably wouldn't write this code, but this illustrates the underlying steps involved.

The problem with this multi-step operation is that it's not atomic. It's possible that two threads race past step 1 before either gets to step 3. In that case, both threads would think the `current_value` is 0 and assign the value of 1 to `@counter` in step 3. A correct implementation should end up with `@counter` equal to 2 with two threads.

The old guard would solve this problem with a mutex:

```
# ./code/snippets/locking_plus_equals.rb

@counter = 0
@mutex = Mutex.new

@mutex.synchronize do
  current_value = @counter
  new_value = current_value + 1
  @counter = new_value
end
```

This implementation would ensure only one thread can perform this multi-step operation at a time, preserving correctness by limiting the ability of things to happen in parallel. It's safe to use with multiple threads.

This problem can be solved a different way using CAS. It would look something like this:

```
# ./code/snippets/atomic_plus_equals.rb

require 'atomic'

@counter = Atomic.new(0)

loop do
  current_value = @counter.value
  new_value = current_value + 1

  if @counter.compare_and_set(current_value, new_value)
    break
  end
end
```

This version is a bit more code and uses more control structures. However, it doesn't use any locks and is still safe to use with multiple threads. I'll walk you through the logic here, then show you a shorthand to make things much more readable. Since this is Ruby, the `Atomic` class *does* provide a much more elegant API; however, the set of primitive operations you're seeing here should be possible in any environment that supports CAS.

```
require 'atomic'

@counter = Atomic.new(0)
```

Here, you wrap the value that you want to perform CAS operations on inside of an `Atomic` object. In this case, our counter value, starting at 0, is wrapped by an `Atomic`.

```
loop do
  current_value = @counter.value
  new_value = current_value + 1
```

Here's the main part of the logic. First you enter a loop. The reason for this is so that the whole operation is retried if the `compare_and_set` fails. I'll repeat that again: **the whole operation is retried if the `compare_and_set` fails**. That means that your operation must be idempotent. In other words, it can be run multiple times without having any additional effects outside of its scope. If something is idempotent, there's no harm in doing it multiple times, which is exactly what could happen in this case!

Next you grab the current value by using `Atomic#value`, then calculate the new value by adding 1 to it.

```
  if @counter.compare_and_set(current_value, new_value)
    break
  end
end
```

Now for the punch line. You call the `compare_and_set` method; the first argument is what you expect the value to be; the second argument is the new value that you want to set it to. If the underlying value *has not* changed, the update will succeed, returning true. In this case nothing needs to be retried so you `break` out of the loop.

Now for the failure case. Between reading the value into `current_value` and setting the new value here, it's possible that another thread has updated this variable. In that case, the operation would fail. The first argument would *not* be equal to the real current value. In that case, this method returns false and the loop is restarted.

Notice that there's no locking involved. With the happy path, when the update succeeds, there's no locking that needs to be done. If the update fails, there's a different cost to be paid: the whole operation needs to be retried.

What's the point of this? The point is that for some kinds of operations, locking is very expensive. Since only one thread can be in a critical section at any given time, all other threads need to be put to sleep to wait their turn when they try to execute the same code path. Using a CAS operation, all of the threads remain active. If the cost of retrying the operation is cheap, or rare, it may be much less expensive than using a lock. All this is to say that sometimes a lockless algorithm is much faster than a locking variant, but other times this isn't the case.

Let's compare the running times of the locking and lock-free implementations of the `+=` operation. But first, I want to show you the Ruby sugar that `Atomic` provides.

Since this construct of using a loop to retry operations is so common, `Atomic` supports an `update` method that encapsulates this. We can re-write our previous CAS example like so:

```
# ./code/snippets/elegant_atomic_plus_equals.rb
```

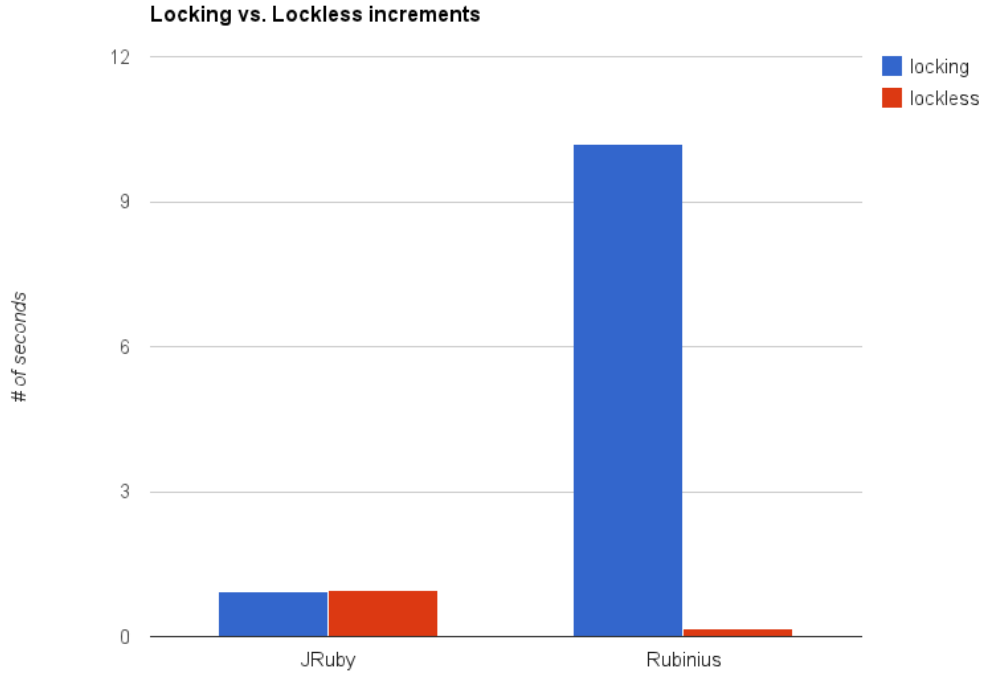
```
require 'atomic'

@counter = Atomic.new(0)

@counter.update { |current_value|
  current_value + 1
}
```

You pass a block to the `update` method. Under the covers it will fetch the current value, then call this block, passing the current value as a block variable. The result of evaluating the block is taken to be the new value you want to set, which is then used for the `compare_and_set` operation. If the operation fails, the block is called again, passing in the updated current value. As I mentioned, it's important that this block is idempotent, as it may be called multiple times.

I think we can all agree that this version is more elegant, with more of a Ruby-ish feel to the API. On to the benchmark.

Benchmark

I simply took the locking and lockless variants of the counters presented in this chapter and benchmarked them against each other. In this simple benchmark, I had 100 threads each incrementing the counter 10,000 times. To be sure, this is an abnormally high workload for a counter, normally the threads would also be engaged in other behaviour. Nonetheless, here is a plot of the results from running `code/benhcmarks/locking_vs_lockless.rb`.


Locking vs. Lockless increments
12
9
6
3
0
# of seconds
JRuby
Rubinius
locking
lockless

As expected, the results differ wildly between implementations. With JRuby, the locking variant was consistently on par with the lockess variant. With Rubinius, however, the lockless variant was consistently faster by about 10x. Don't expect these results to hold in all situations. As always, the only surefire way to know what will work for your code is to measure.

CAS as a primitive

We've been looking at CAS operations as an alternative to locking solutions. They certainly can be used this way, but research has shown that CAS are perhaps more useful as a lower-level abstraction. Indeed, it's been proven that with this one primitive, you can *any* lock-free algorithm.[2]

Some new technologies, such as Clojure, take full advantage of this and are showing what's really possible with multi-threaded concurrency. In the end, CAS operations don't provide a silver bullet. In some situations it can provide speed increases, in other situations, new functionality.

As far as Ruby goes, there's not much support for these kinds of approaches. There's the 'atomic' rubygem, and a few experimental data structures[3] floating around. However, now that we have a CAS primitive available in Ruby, many academic papers outlining algorithms should be applicable.[4]

---

2. http://www.podc.org/dijkstra/2003-dijkstra/
3. One can be found at https://gist.github.com/jstorimer/5298581, the other is an implementation of the Disruptor algorithm at https://github.com/ileitch/disruptor

4. Here are a few that are good introductions to these topics: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.3363&rep=rep1&type=pdf, http://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html

Chapter 19

Appendix: Thread-safety and Immutability

You learned earlier that the main thread-safety issue that's going to come up is concurrent modification. When two threads are trying to modify an object at the same time, things get unpredictable.

But immutability provides a way around this. An immutable object is one which can't be modified after it's created. If an object can't be modified, then, by definition, two threads can't modify it at the same time. So, by definition, immutable objects are thread-safe!

On the surface, this sounds like an easy path to concurrency, but writing programs with no mutability is very difficult. Pure functional programming languages, like Haskell, implicitly support this programming model, and are notoriously hard to master. In Ruby, everything is mutable by default, so attempting to write fully immutable programs involves going against the grain and the language idioms.

But there's no need to throw out the baby with the bath water. If writing fully immutable programs is difficult, there are certainly places for immutable objects or immutable data structures that can provide an easier path to thread safety.

Immutable Ruby objects

Immutability is actually supported in core Ruby using the `Object#freeze` method.

```
# This is a mutable Array
comics = []

# Appending to the array mutates it
comics << 'random'

# Freezing the Array makes it immutable
comics.freeze

comics << 'random'
#=> RuntimeError: can't modify frozen Array
```

The `freeze` method makes the Array immutable, but then it effectively becomes unusable. How do you append to an Array that can't be updated?

The typical method signature for immutable objects is: methods that would typically mutate the object instead return a new version of the object with the mutation applied.

Here's an example using the 'hamster' rubygem.

```
# ./code/snippets/hamster_vector.rb
```

```
require 'hamster/vector'

mutable = Array.new
immutable = Hamster::Vector.new

mutable #=> []
mutable.push(nil)
mutable #=> [nil]

immutable #=> []
immutable.add(nil) #=> [nil]
immutable #=> []

# This is typical of immutable data structures,
# re-assign the reference to the result of an
# operation.
immutable = immutable.add(nil)
immutable #=> [nil]
```

Notice how the `immutable` object retained its original state even after an element was pushed onto it? Instead of mutating itself, its `push` method returned an updated version of itself containing the new element. It's a sneaky way to avoid mutating objects!

If you want to use immutable data structures in your Ruby program, you'll want to check out `Hamster`. For one, it's got mature immutable implementations for Hash, List, Vector, and others. It doesn't match Ruby's API 100%, but it provides what makes sense for an immutable data structure. In terms of efficiencies, you might, at first glance, think there's a lot `dup`ing of Ruby objects happening under the hood to create

new versions of these data structures, but Hamster has a much more efficient implementation that will be friendly to the garbage collector.[1]

Integrating immutability

You've had a quick look at how an immutable object works. Now how is it typically used? The simplest use case is this: **when you need to share objects between threads, share immutable objects**. If you need to share objects with other threads, it's always preferrable to share an immutable object. If you share a mutable object with another thread, then you need to be concerned about thread safety, probably introducing synchronization.

This kind of complexity literally disappears if you share an immutable object. When passing an object to another thread, making it immutable is a simple win. But what about other situations?

Let's look at a classic multi-threaded arrangement: a producer and consumer. This problem involves one thread distributing work to another group of threads, so you definitely need a thread-safe data structure.

Here's how you might do it with a Hamster data structure.

```
# ./code/snippets/hamster_producer.rb
```

1. The README outlines the API and links to the paper describing the implementation: https://github.com/harukizaemon/hamster#readme.

```
require 'hamster/queue'

@queue = Hamster::Queue.new

10.times do
  @queue = @queue.enqueue(rand(100))
end
```

I've started with just the producer side of things because I see an issue already. In the examples from the beginning of the chapter, you updated the immutable object by re-assigning it to a new version of itself. That's what's happening here, but that's not safe in the presence of multiple threads.

There's no synchronization being used here, so it's possible that between the time that the producer thread evaluates the right hand side of its assignment to get a new version of the queue and the time it actually assigns that value, another thread could have removed an element from the queue and updated the reference. When the producer thread completes its assignment, it could overwrite the change made by the consumer thread.

So we can't get away from using some form of synchronization here. That's generally the case with immutable objects. **It's very easy to pass out immutable objects to share, but if you need to have multiple threads modifying an immutable object you still need some form of synchronization.**

In this case, immutable data structures work great with CAS operations. Here's an updated version making use of both:

```ruby
# ./code/snippets/hamster_producer_consumer.rb

require 'hamster/queue'
require 'atomic'

@queue_wrapper = Atomic.new(Hamster::Queue.new)

30.times do
  @queue_wrapper.update { |queue|
    queue.enqueue(rand(100))
  }
end

consumers = []

3.times do
  consumers << Thread.new do
    10.times do
      number = nil

      @queue_wrapper.update { |queue|
        number = queue.head
        queue.dequeue
      }

      puts "The cubed root of #{number} is #{Math.cbrt(number)}"
    end
  end
end

consumers.each(&:join)
```

Notice how well these two approaches work together?

The immutable approach looked out of place before, but now, inside of the `update` block, it looks like regular mutable code. Remember the `update` block only cares about the return value, and the immutable operations are returning new versions of the queue to replace the existing version. Since the underlying queue is immutable, the operations in the block are idempotent and can be run any number of times.

I'll take a minute to address the consumer side of the code, particularly how the dequeuing happens. Here it is again:

```
@queue_wrapper.update { |queue|
  number = queue.head
  queue.dequeue
}
```

With a `Hamster::Queue`, popping a value from the queue is a multi-step operation.

The `dequeue` method returns a new version of the queue with the first element removed. This is what you'll want to assign back to the underlying queue reference. But you also need a way to retrieve that first element before updating the list. `Hamster::Queue#head` will give you that first element without modifying the queue. One you have a reference to head, you can try to update the queue with the new dequeued version.

Wrap up

Like almost everything else you've seen, this approach has its pros and cons. On the one hand, immutability is a nice guarantee to have, it's the simplest path to thread safety when sharing objects. However, when you need to collect data from immutable objects, it sometimes requires some re-thinking and some new techniques.

In terms of performance, the comparisons are a little unfair. Ruby has no native immutable collections, so the native mutable collections are generally faster than what Hamster can offer. However, if you're looking for the flexibility and guarantees that immutable collections offer, the tradeoff is a good one to make.

I'll leave you with the 4 rules to safe concurrency again. Rule #3 is of particular relevance to this chapter.

The safest path to concurrency is:

1. Don't do it.
2. If you must do it, don't share data across threads.
3. **If you must share data across threads, don't share mutable data.**
4. If you must share mutable data across threads, synchronize access to that data.