WORKING with RUBY THREADS

Jesse Storimer

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

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.