What is a reduction and why Fibers are the answer for Ruby concurrency

Feb 6, 2021

In the Ruby 3 features, a lot of attention went to Ractors - a new parallelism primitive which provides what can best be described as "Web Workers" - separate threads of execution with memory isolation from the spawning thread. However, there was also a release of a seemingly "nerdy" feature which is the FiberScheduler.

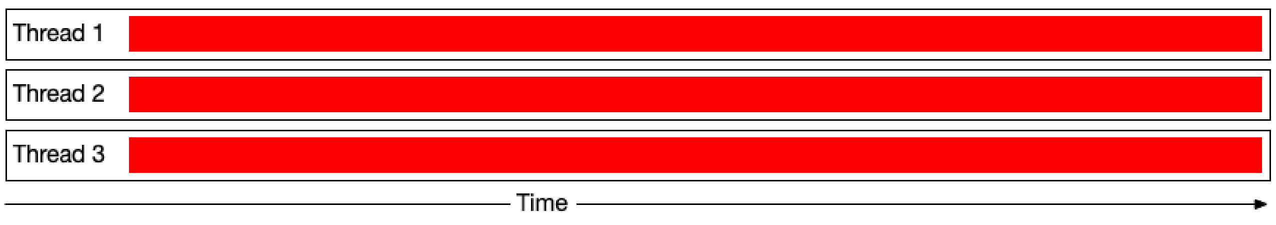
Ractors still have to prove their own (my dear friend Kir Shatrov has done some exploration into designing a Ractor-based web server) but I would like to highlight that second feature, and the concept of scheduling and reduction in general.

I strongly believe making good use of Fibers is instrumental to Ruby staying relevant for creating web applications. As a community we are in a tight squeeze now, between Go on one side and Node.js on the other side, and the concurrency story in Ruby isn't that good compared to either of the two - however we stand a measurable, substantial chance of once more getting ahead of the game. Especially considering that Python3 has chosen for the "colored" functions model with its `asyncio` setup. See Kir's article for an interesting perspective on this too.

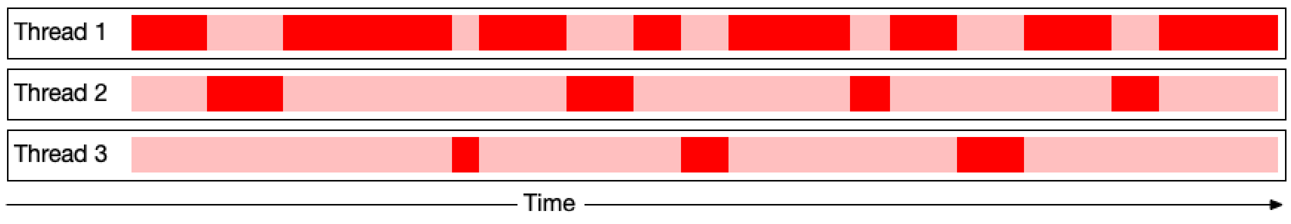
See, when people talk about Ruby parallelism and concurrency usually the conversation is quickly curtailed by the first person who screams "But the GIL! So there is no parallelism!" and then the room falls silent. In practice the situation is much more nuanced, and gaining a better understanding of the nuances will make your Ruby programs faster, more efficient and let you use less compute. If you know where to look.

A whirlwind tour of Ruby concurrency (and parallelism)

The current situation in Ruby using threads (the native `Thread` primitive) is basically as follows. Imagine we have multiple threads (listed vertically) which are performing some work concurrently, and the time is on the horizontal axis:



The red sections are **executing MRI opcodes** - some kind of Ruby code work. This can be Puma threads, or Sidekiq threads - anything you can meaningfully parallelize. However, there is a GVL (Global VM lock) which ensures that no two Ruby opcodes can execute in parallel. The reasons for that are curious, but mostly they have to do with a *very* thin layer between raw C pointers - which native code uses inside MRI, for things like memory allocations, IO and calls to libraries such as Oniguruma. Since Ruby threads are *pthreads* - POSIX threads - if this lock gets removed multiple threads can perform the native calls at the same time. This can call out into not-thread-safe APIs also, and more of them exist than you could think of. For instance, `ENV[...]` - the `getenv()` call - is *not* thread safe, and without having a mutex you would have to contend with that in your Ruby code. So, in practice, when all your threads are executing Ruby code, what you get is so-called **between the lines concurrency** - similar to what you have in Node and in the browser Javascript engines:



When a thread is executing Ruby code other threads cannot obtain the GVL and have to wait. Now, if you are at a roundtable of grumpy developers who can't wait to switch your entire team to Java or Go this is where the conversation usually ends. "There is no real concurrency, so there is nothing to discuss in here" lauds the answer, and plans get drawn to rewrite perfectly workable software for a different runtime. However we need to dig a little deeper - not only to have a healthier, more informed conversation about the topic, but also to understand how your program gets executed.

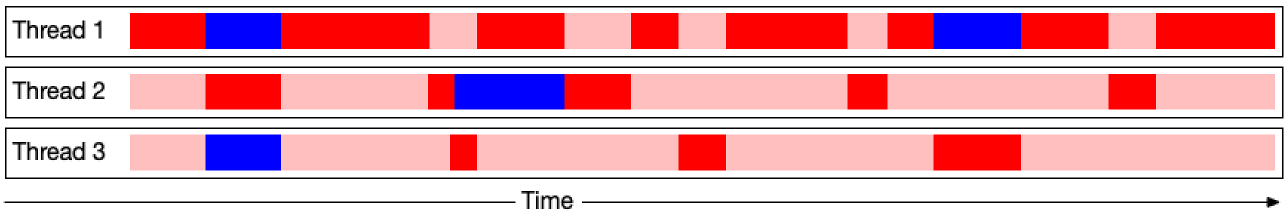
In reality, in addition to the Ruby code there is also **native code** - code which is provided by either bindings to the system calls of the operating system or to the calls of an extension (like a gem, or like a native standard library component such as Oniguruma). These calls **may** at their discretion **release** the GVL, by using code like this:

```
ret = rb_thread_call_without_gvl(
        (void *(*)(void *)) native_func, native_func_argument,
        session_ubf_abort, (void*)some_native_func_state
      );
```

Native code has the choice whether it wants to do this for the following reason: when the GVL is released the code must be able to guarantee that it **won't be touching any MRI data structures** such as `Strings` (`RSTRING`), arrays, or any other `RVALUE` objects - or call any MRI functions which manipulate these structures. In practice this is possible and highly desirable, because when the GVL is released there *will be* some degree of parallelisation. Usually it is done like so:

```
data_copied_to_buffer = copy_data_from_ruby_structure_to_native_struct(struct_ptr
rb_thread_call_without_gvl(...)
```

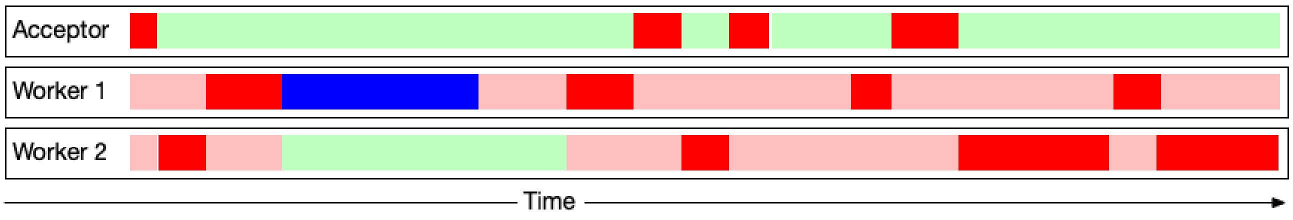
and the picture of our thread flow will then look markedly different:



The blue sections denote where our thread enters native code which is able to release the GVL. In that situation there actually *will be* true parallelism, as nothing prevents the native code from executing in parallel.

And it is not only the handcrafted C extensions which are able to release the GVL - inside MRI there is a number of functions which will also release the GVL when they wait for IO. For example, when you call `IO.select` the VM is going to enter a select call in your thread, and register an **unblock function** for when the `select()` call returns. If you are calling a database, most likely your database access gem does exactly the same - it dispatches an IO call and then polls on the socket while your database chugs along. The same happens in Puma - imagine we have one thread which is the “acceptor”, which takes incoming connections. This thread then passes the file descriptor for the socket to one of the worker threads of the web server.

If the “acceptor” thread would be holding the GVL all the time, Puma would not be able to do any useful work at all because all the time the acceptor is listening for a connection. So the flow of execution will look something like this (light green sections is waiting on IO):

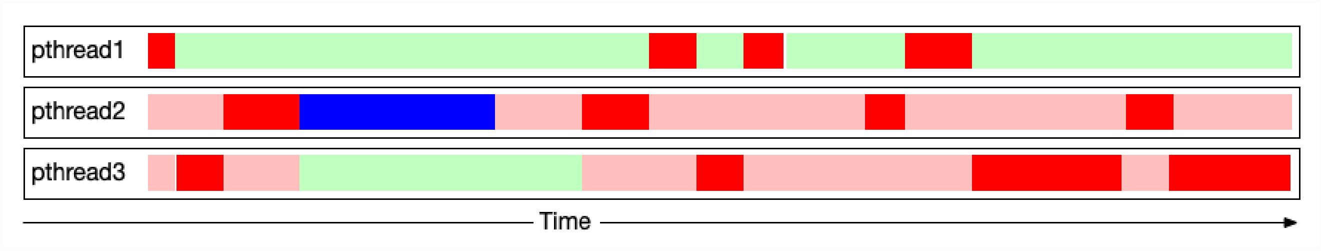


So, the first important takeaway: yes, Ruby (or YARV/MRI more specifically) **does** have concurrency, and **does** have parallelism (although it is limited to IO waits and native code sections).

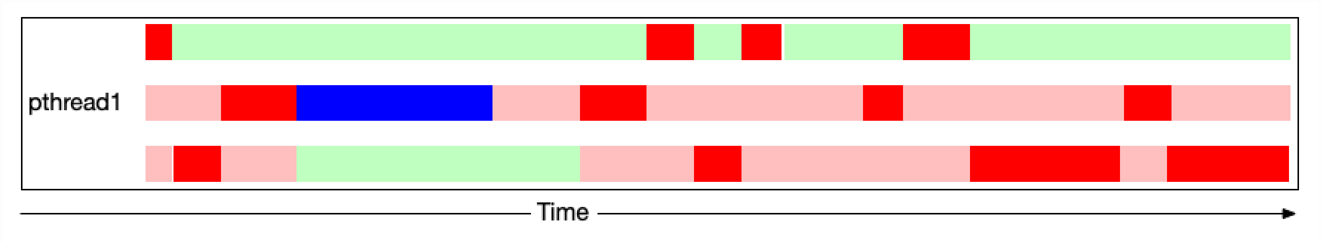
The second important takeway: even though there is a GVL, it **is possible** for multiple Ruby threads to be executing at the same time (some threads are in an IO wait, some are in a GVL unlocked state, and one will be running MRI opcodes).

Why you likely want concurrency more than you want parallelism

Remember what the original premise of Node.js was? "Threads are crap, we need to use events.". However, events and a standard Node setup present the same "between the lines concurrency" functionality. You will have chunks of waits (on timers or on external events such as "this file descriptor was scanned and is now known to be ready to take writes"). No two opcodes/VM instructions will be executing at the same time either. But there is a second angle to this, namely - in Ruby every VM thread is *also* an operating system POSIX thread. These threads are expensive to create and manage, and rely on layered scheduilng - the logic that decides when the threads will run. So, if performance were exactly the same, this situation:



costs us more system resources than being in this situation:



Note that in both cases we are only running **one** "blue" section - native code. But with a network server - like your web application in Rails - those IO waits will be *much more frequent* than the other code. The main scaling point for our applications is the database, and while the database is doing work your HTTP worker thread will be sitting in an IO wait block, idling until the database starts replying with the resultset. So, all things considered, **cheaper concurrency is more important for web applications than raw parallelism.**

There is also a second reason for this: the amount of parallel compute that your machine can do is limited by the number of CPU cores. Even if you have 24 or 64 cores - like on the latest AMD processors, which are absolute BEASTS of a box, you can give every customer one core and not more than that. Otherwise computation for this customer will have to pause every now and then. Do you have a situation where you need to service 64 customers (and no more) per box, with a web application? Rarely so I'd wager. Yes, there are certainly situations where you want extreme parallelism:

- Doing one very expensive task and making it complete as fast as it can. Like rendering a 3D scene, whereby you will be ready to dedicate a number of cores just to that task
- Doing a known, fixed number of not very expensive tasks which absolutely must get the entire core at their disposal - for example, realtime control of a system with multiple parameters (hard- and soft-realtime systems)

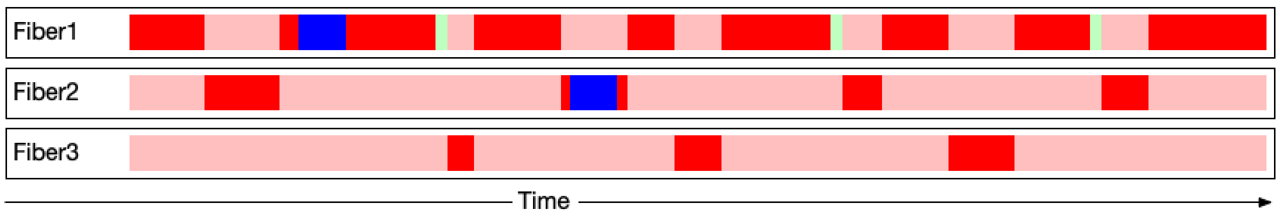
But being able to service 10000 people from one computer **is not one of those.** What you do want is a way to service those 10000 people the fastest, which will usually mean - being able to *switch* between them very quickly when servicing.

This aspect is very important to remember: in your web app you want concurrency more than you want parallelism. Both are nice, but concurrency wins.

Threads are **preemptive** - they can be externally paused. With pthreads, it is the combination of the MRI smarts and the OS thread scheduler which manage them. For example, within a Ruby thread you can call `Thread.pass` which will yield control back to the scheduler and allow another thread to be resumed. Also when you call `IO.select` inside a thread, the Ruby mechanics will allow other threads to execute code which requires the GVL to be unlocked and locked by those threads. You can also do `Thread#kill` and other interesting stuff, but the important point remains - Ruby has some amount of automatic, outside-in control of Threads.

What are Fibers (coroutines?)

Fibers are blocks of code which can be *paused.* In effect they are the same as our second picture - but instead of executing within a process and being Threads they execute *sequentially* within a same thread. Even though Fibers are called “a concurrency primitive” I like to call them “a sequencing primitive” - with Fibers only one Fiber is “alive” and executing Ruby code, other Fibers within the same thread are asleep:



You will notice that again - only one Fiber can be executing Ruby code or native code at a time. There is a big difference in terms of resources - all those Fibers live inside of a single Ruby thread (and thus inside of a single pthread), so much less resources are wasted. Also, since we do not have a way to do an “IO wait” inside of a Fiber the light green sections are where a Fiber discovers it cannot proceed due to IO blocking and **pauses itself.** This is a very important distinction - if Ruby threads will automatically “sleep” or “give way” once they enter a blocking section, a Fiber has to attempt an operation that would block, and if the operation would block the Fiber has to deliberately pause itself and “give way”. Every fiber must participate in cooperative multitasking - it must be able to “yield back control”. This is what `Fiber.yield` does.

Fibers are indeed **much more** useful for where concurrency is more important than parallelism. They are much cheaper (a fiber is just a datastructure holding a call stack), they do not use OS resources (pthreads) and do not imply that the native code they call into has to be thread-safe as they only get executed sequentially. With Fibers it becomes feasible to service many thousands of clients from a single Ruby process, **provided these clients do not require continuous computation.**

If you want to know more, Samuel Williams has written Fibers Are the Right Solution just about that.

The “play” button - who resumes Fibers if you don’t do it manually?

Although the yielding (the “pause button”) is cooperative (happens from the inside of the fiber) the resumption (the “play button”) gets activated from the *outside* of the Fiber. Something (or someone) has to call `resume`. For instance:

```
Thread.new do
  loop do
    readable_socket.wait_readable # Using "require 'io/wait'"
    # At some point the thread scheduler will resume us automatically,
    # that will happen once "wait_readable" returns
    str = readable_socket.read_nonblock(65*1024)
    do_something_with_str
  end
end
```

however, with Fibers:

```
Fiber.new do
  loop do
    if readable_socket.readable?
      str = readable_socket.read_nonblock(65*1024)
      do_something_with_str
    else
      Fiber.yield # We pause and something has to resume us deliberately
    end
  end
end
```

We can use something like nio4r to resume our Fibers for us:

```

# Our Fiber will copy some bytes from a socket to
# a destination, and then "yield"
socket_fiber = Fiber.new do
  loop do
    some_destination.write(socket.read)
    Fiber.yield # Pause the fiber after doing one write
  end
end

monitor = @selector.register(socket, :r)
monitor.value = socket_fiber

loop do
  @selector.select { |monitor| monitor.value.resume }
end
```

But this, of course, is a bit cumbersome - it would be much nicer if there was *something* that would automatically register resources we need to watch (from inside of our Fibers) and then manage to `resume` us once the socket is ready. This is what the scheduler is for.

The "pause" - who yields from Fibers if you don't do it in your code?

Any code you run inside your Fiber may do a `Fiber.yield`. This is difficult from the point of view of polymorphism (your Fiber may return *literally anything at all*), but we won't get into that.

Normally, you would `yield` manually. For example:

```
f = Fiber.new do
  ....
  if !socket.writable?
    Fiber.yield
  end
end
```

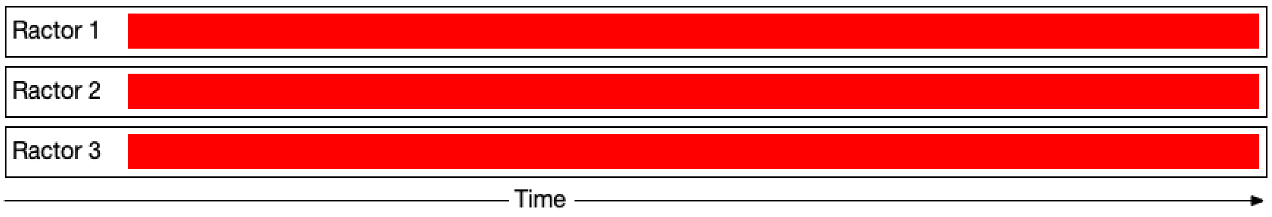
But in some libraries - such as async - there are extra features. For example, Async gives you a `Writable` and a `Readable` instead of `IO` objects. When send them `write()` or `read()` messages and the socket is not readable/not writable, the `Writable/Readable` will actually do `Fiber.yield` for you - effectively making any non-async Ruby program into an async one. When it works, it is akin to magic - but it does require you use those objects instead of your standard IOs.

```
f = Fiber.new do
  ....
  socket.write(data) # If the socket is not writable, the write method calls Fiber.
end
```

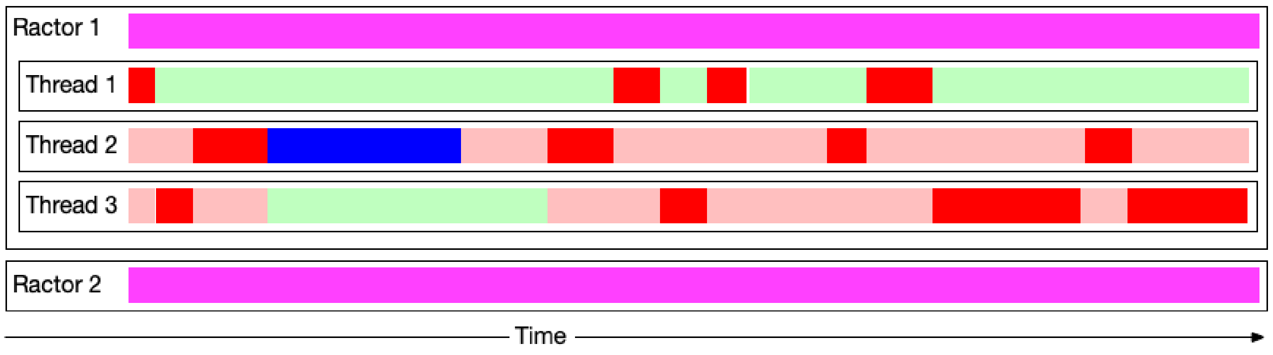
However, if you use the newly added `Fiber.set_scheduler`, you can assign a scheduler which will monitor *every* `IO#read` or `IO#write` and intelligently suspend any Fiber where such an operation will block. This would allow creating evented, highly concurrent Ruby servers which will make things like ActionCable, MessageBus or live-view-like solutions (with long-lived connections) much easier and much more pleasant. In terms of raw impact, the addition of the Fiber scheduler to Ruby 3.x is **much more valuable** than Ractors, which we'll examine below.

Where do Ractors fit in all this

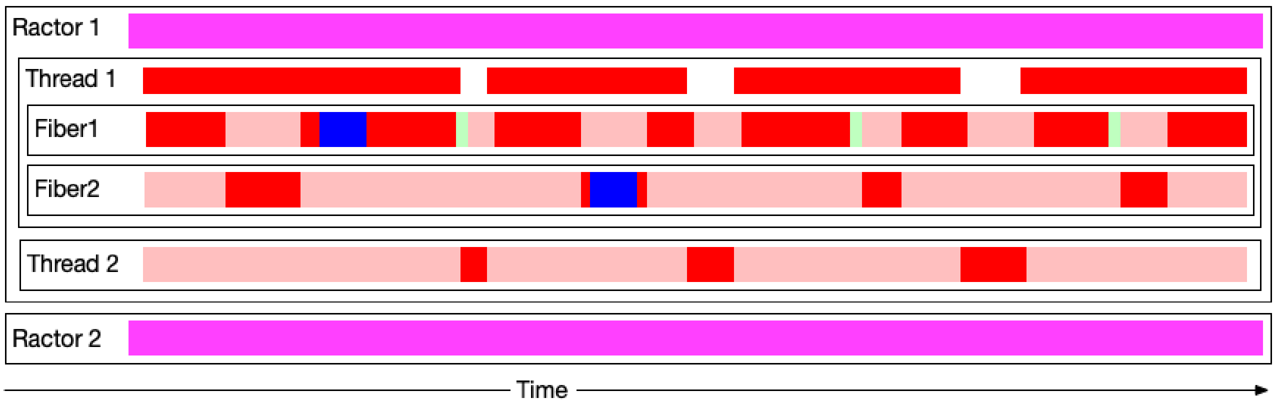
Ractors are "outside" of Threads, and thus outside of Fibers. If we use our drawings from the previous example, on paper Ractors are supposed to deliver "The holy grail" of concurrency *and* parallelism:



That is, we now have concurrent *and* parallel flows of execution which do not have to share a GVL - every Ractor gets its own GVL. If we superimpose our threads illustration over them, our picture will look like so:



and if we add Fibers, like so 🤯:



However, Ractors are not Threads. The closest relative to a Ractor is in fact the WebWorker - it is an entirely separate process. You pass it messages through a kind of a "messagebox".

This works well, but it doesn't take into account a very important property of the Ruby language - that **at runtime any program structure can be modified.** Class and module variables can be added and assigned, new constants can be defined, constants can get removed and redefined - all sorts of wild things. So a lot of stuff Ruby programs normally do - such as calculating a constant:

```ruby
SOME_MAGIC_STR = [200, 102, 210, 10].pack("C*")
```

become problematic inside a Ractor - all of this becomes "forbidden". So in practice what you may run inside a Ractor is very, very limited - at least at the current stage. Also, a whole number of questions emerges:

- How do we implement code reloading? (constants/modules removed and then added again)
- How do we implement singletons? (these messageboxes must be stored somewhere)
- What about queues?
- What about mutable strings?
- How do we pass native data structures? (sockets, file descriptors, other handles to external resources?)

There is a lot of work - including Ruby itself - to turn things which were previously perfectly fine into things which are now-suddenly-not-fine-with-Ractors (see this commit and similar).

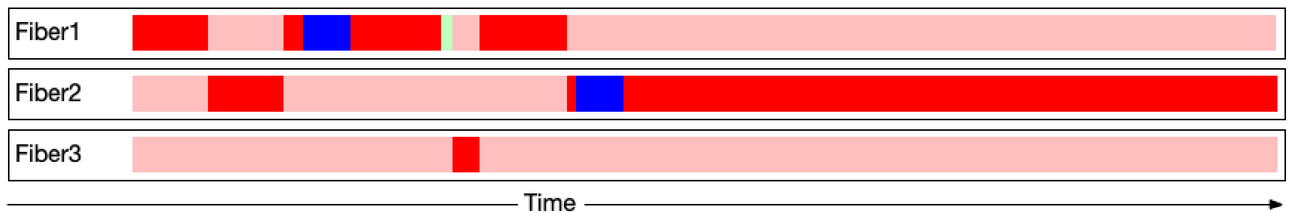
Ractors also try to introduce a concept of immutability into Ruby, which - starting with even Strings - is very hard to achieve (in Ruby one of the most sorry design issues is that strings are mutable byte buffers, which makes the system susceptible to memory bloat).

Given the current state of Ractor, I feel it is unlikely frameworks like Rails or servers like Puma will adopt them as the concurrency mechanism soon - simply too much of the existing Ruby library ecosystem would suddenly turn out to be highly "mutating", "forbidden" and "dynamic". I don't know what the solution to this might be, and I suspect some time needs to pass until such a solution might emerge.

Where Ractors are very likely to find use is in handling **one particular CPU-intensive process** which requires a certain degree of parallelism. For example, computing an ML model, or processing an image, or converting Markdown. I don't see it as highly likely that Ractors will be used as originally intended (as a wrapper of a large, long-lived group of Threads in turn containing Fibers).

The problem with overzealous Fibers

The difficulty with Fibers is scheduling. As I mentioned, the new Fiber scheduler handles the "ying" part of the equation - *resuming a Fiber which is paused.* But there is a "yang" to it - a Fiber has to *pause* - to `yield` somewhere. The concurrency model is, after all, called "cooperative" for a reason - the Fibers themselves must cooperate with each other. When it works it's fine. But imagine we have a Fiber which is *not* cooperative and enters some expensive computation, and does not `yield` on time:



In this example, Fiber2 has started running some expensive operation - such as blurring an image in pure Ruby, or running a regular expression of doom and your system is starved of resources. The scheduling has now become extremely unfair and one "bad actor" - in this case Fiber2 - has gobbled up all the execution time. Since Fiber2 cannot be paused externally (remember - **cooperative**) the only choice we have is to wait for it to complete.

There is some work going on currently to make it easier to trace which Fiber in your program has gobbled up all the CPU time - namely async-beacon but it doesn't solve the "preemption" issue. And this is where **reductions** enter the picture.

What is this "reductions" thing?

The solution for this lies in a really interesting contept which is used in Erlang/BEAM which is called **a reductions counter.** Imagine we have a function which runs a regular expression engine (I'll make it very simple in shape), and it *scans* through a string character by character and returns if a match is found:

```
def matches_regular_expression?(str, regex)
  str.each_char do |char|
    regex.continue_matching(char)
    if regex.match_detected?
      return true
    end
  end
  false
end
```

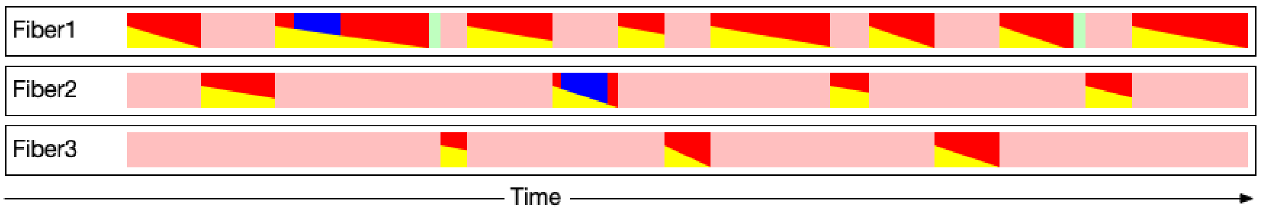
If the `regex` is written in such a way that will cause catastrophic backtracking, the function *will not return* before it has performed the millions of checks that it needs to do, and we will gobble up resources. However, what if the function were also receiving an object which is some kind of a *counter?* We would decrement this counter when we know we have consumed some amount of compute, and we will know that this counter will magically replenish when our caller `resume`s us:

```
def matches_regular_expression?(str, regex, consumed_operations_counter)
  str.each_char do |char|
    regex.continue_matching(char) # Perform something with a "computational cost" o
    consumed_operations_counter.decrement! # Decrement the counter
    return true if regex.match_detected?
    Fiber.yield if consumed_operations_counter.zero?
  end
  false
end
```

A contraption like this ensures that every call we do (a meaningful call) would be cooperative by default. The consumed operations counter can use just counts, but it can also monitor timeouts (act as a kind of a deadline) etc. The complexity, of course, lies in the fact that for this to be feasible the regular expression engine must **check this counter** after every scanned byte. So the counter must enter native code (in Ruby's case Oniguruma).

That counter is what a reductions counter is in BEAM. Since processes in Erlang are effectively fibers - coroutines - Erlang uses reductions as a tool to ensure fair scheduling even in the face of functions which might run for a very long time. This clever setup allows Erlang to truly be the kind of concurrent execution.

If we map the changes of the reductions counter to our timelines, we would see something like this:



Fiber2 exuausts its reductions counter at some point and yields control to other Fibers to continue. Every time a Fiber gets resumed, the counter gets reset.

My former colleague Wander is currently doing some research into reductions/interrupts for Fibers, which I hope he is going to publish soon.

In conclusion

In my view, for Ruby web apps to be viable **concurrency is vastly more beneficial than parallelism.** Ractors are, at this stage, **problematic with most existing Ruby programs.** The **fiber scheduling system** added to Ruby 3.x **is going to be extremely important to the story of Ruby on the server** (and of UIs, if that ever reemerges).

If you have strong feelings about Ruby and the Ruby ecosystem *surviving* it is **essential** that Fiber-based concurrency - and projects exploiting it, such as socketry and polyphony have your full support and attention. More so than various `ActiveSomething` Rails extensions or library gems. Test your libraries with these async frameworks, fix bugs, and if you can - try to develop production systems running on them.

* * *

Thanks to Kir Shatrov and Wander Hillen for reviewing the initial version of this post.