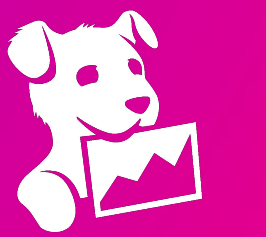
*Understanding the Ruby*

# Global VM Lock (GVL)

*by 👀 observing it*

Ivo Anjo / ivo.anjo@datadoghq.com



DATADOG

# Who am I

**Ivo Anjo**

Senior Engineer @ **Datadog**

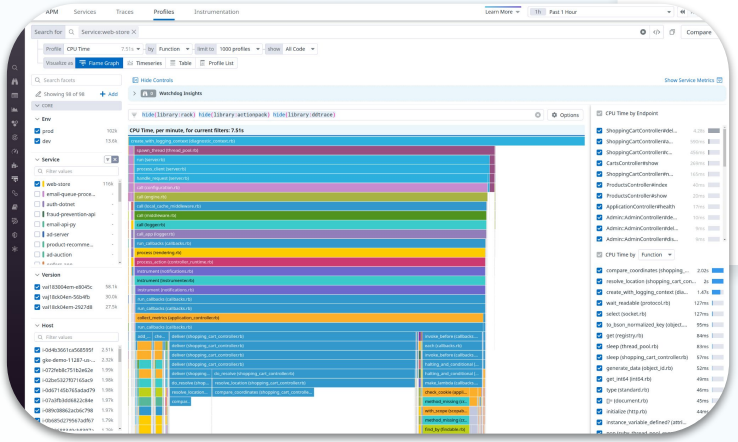
- ❤ Ruby since 2014
- ❤ Exploring language runtimes
- ❤ Application performance
- ❤ Making viz tools to uncover new insights 🎉

# Who am I

**Ivo Anjo**

Senior Engineer @ **Datadog**

I build **Datadog's** open-source **Continuous Profiler** for **Ruby**: **dd-trace-rb** 🎉



*this thing!*

# Outline

00 My story

01 What is the Global VM Lock (GVL)?

02 What is the GVL **not**?

03 Observing the GVL

04 Measuring the impact of the GVL

05 What do you do with this information?

2014: Started using Ruby at my first job

Someone tells me that's why Ruby doesn't use multiple cores

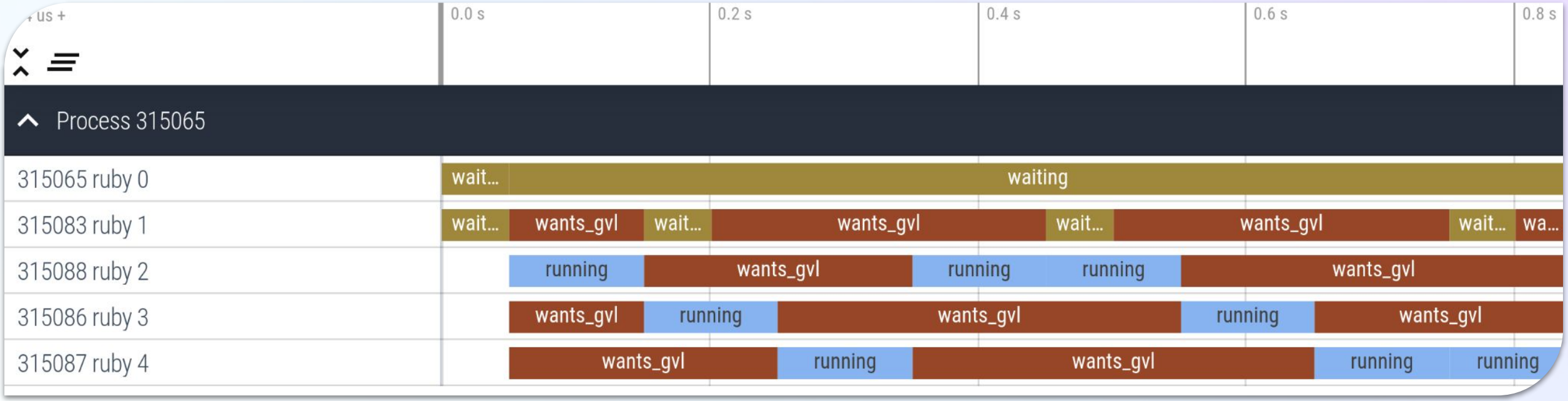
Saw some references to this GVL (GIL?) thing online

*ah... cool? I guess*

- Took me a long time to really start to understand what the GVL means to Ruby application **performance**, especially **latency**

- It also took the work of **Jean Boussier** (@_byroot) and **Matt Valentine-House** (@eightbitraptor) at Shopify to make the GVL much easier to 👀 observe
  *(You rock, Jean && Matt!)*

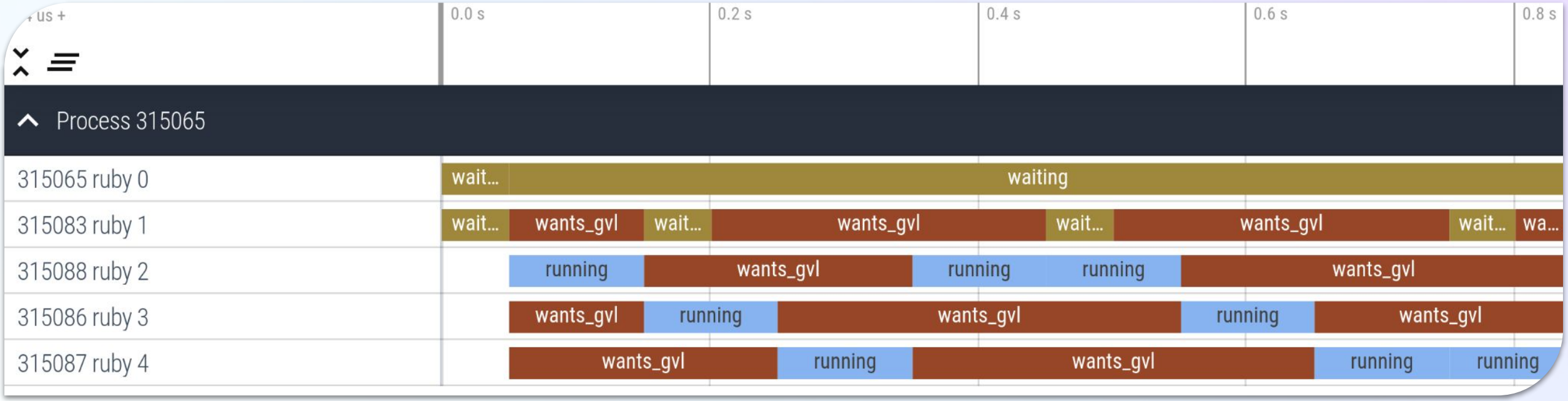
# …this led me to create the `gvl-tracing` gem





and get retweeted by Matz 😁

# …this led me to create the `gvl-tracing` gem

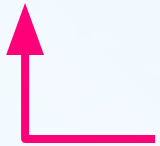



Here's what I learned so far in my saga!

# *What* **is** *the* **Global VM Lock (GVL)?**

# What is the Global VM Lock (GVL)?

```
$ ruby hello.rb
```



Ruby VM is a big program that is written in **C**
*(+ Rust, + Ruby)*

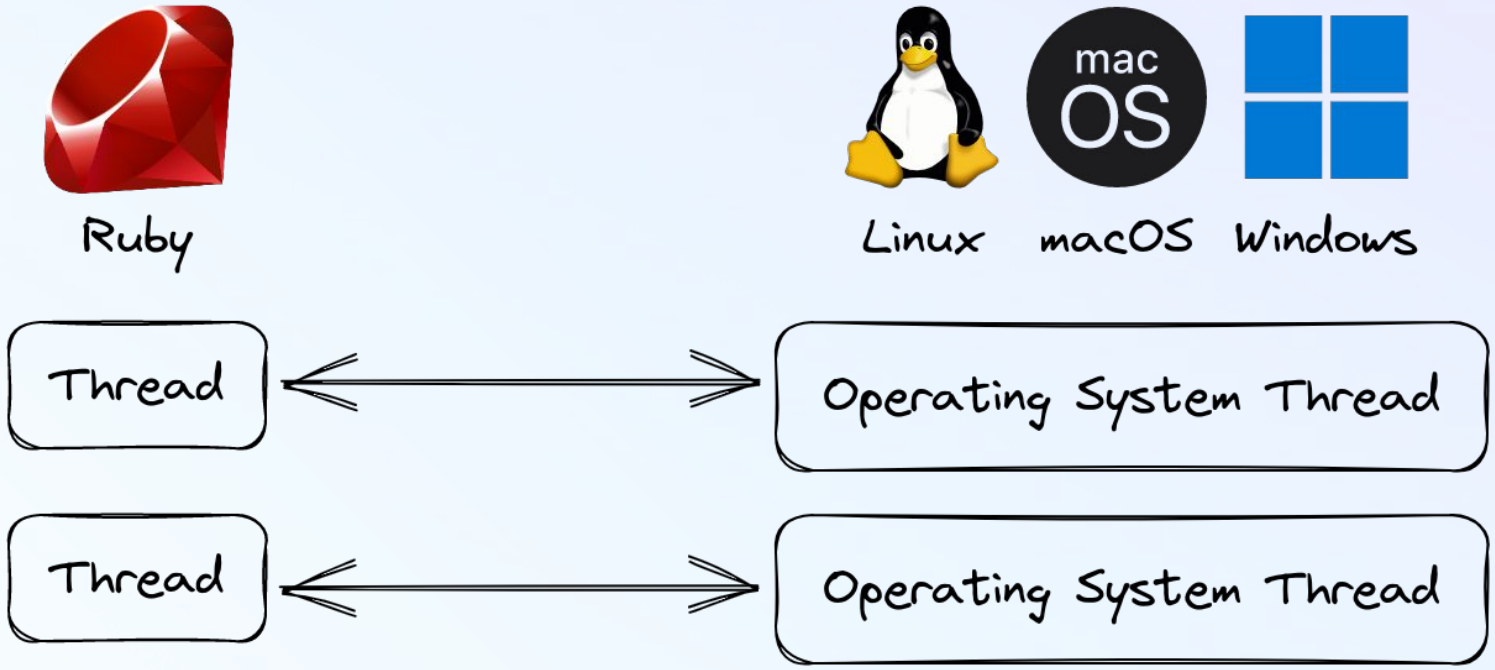
You may also hear it called **CRuby** or **MRI**

```
Thread.new { puts “Hello” }
Thread.new { puts “World” }
```

When
- you
- your webserver (puma?)
- your background job processor

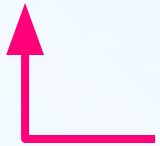
create threads...

# …they get mapped 1:1 to Operating System threads

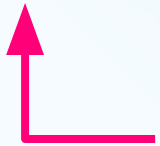


# What is the Global VM Lock (GVL)?

```
$ ruby hello.rb
```



Ruby VM is a big program that is written in **C**



That uses multiple OS threads (“multi-threaded”)

*The*
**Global VM Lock (GVL)**
*is a*
**mechanism**/strategy
*to ensure*
**correctness**
*of Ruby the multi-threaded C program*

# Aside #1

What's a lock?

*It's a mechanism that allows*

***only one** thread (holding the lock)*

*to work at-a-time*

Image credit: Pxfuel.com

*When a thread inside the Ruby VM is*

- Running Ruby code
- Interacting with Ruby VM data

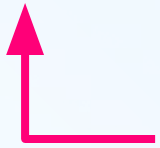
*it **needs** to be holding the GVL*

**No** other thread can be doing the same

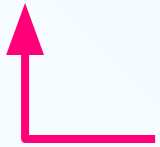
They need to **wait** for their turn

# Aside #2

What about the **Global Interpreter Lock** (GIL)?

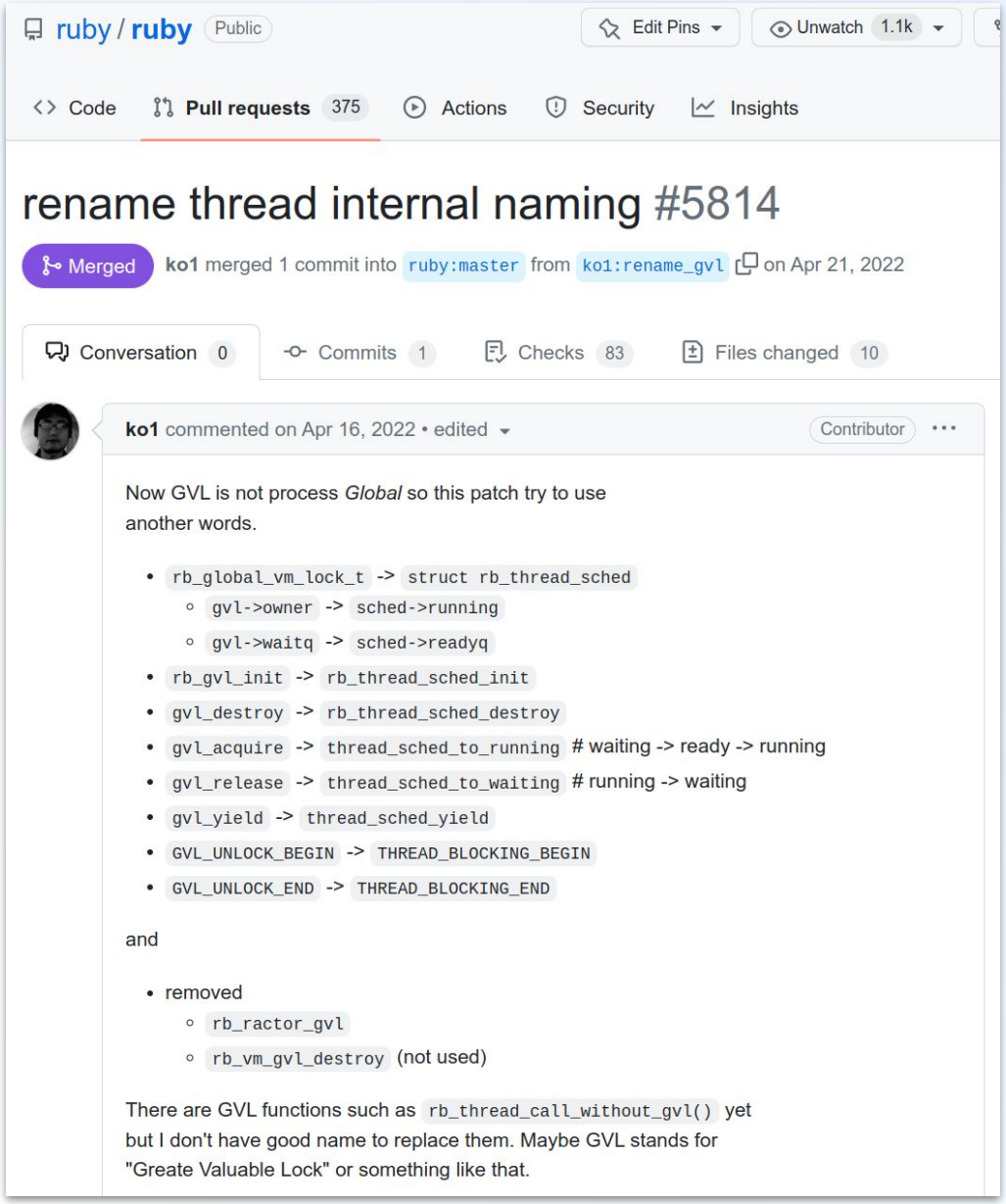


It’s what **Python** calls their GVL!



Old versions of Linux actually had the
**Big Kernel Lock** (BKL) – similar idea, but inside Linux!

# Aside #3: Great Valuable Lock?



ruby / ruby Public

Edit Pins | Unwatch 1.1k

Code | Pull requests 375 | Actions | Security | Insights

## rename thread internal naming #5814

Merged ko1 merged 1 commit into `ruby:master` from `ko1:rename_gvl` on Apr 21, 2022

Conversation 0 | Commits 1 | Checks 83 | Files changed 10

**ko1** commented on Apr 16, 2022 • edited — Contributor

Now GVL is not process *Global* so this patch try to use another words.

- `rb_global_vm_lock_t` -> `struct rb_thread_sched`
  - `gvl->owner` -> `sched->running`
  - `gvl->waitq` -> `sched->readyq`
- `rb_gvl_init` -> `rb_thread_sched_init`
- `gvl_destroy` -> `rb_thread_sched_destroy`
- `gvl_acquire` -> `thread_sched_to_running` # waiting -> ready -> running
- `gvl_release` -> `thread_sched_to_waiting` # running -> waiting
- `gvl_yield` -> `thread_sched_yield`
- `GVL_UNLOCK_BEGIN` -> `THREAD_BLOCKING_BEGIN`
- `GVL_UNLOCK_END` -> `THREAD_BLOCKING_END`

and

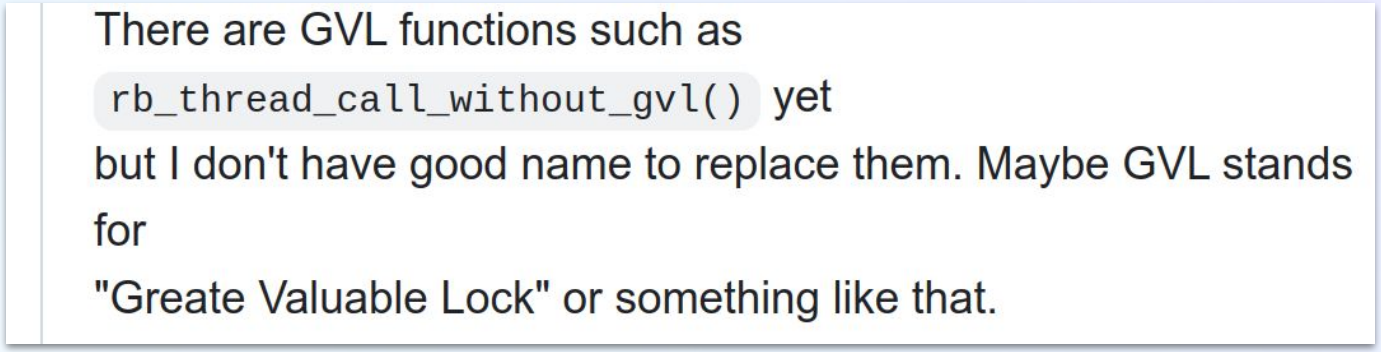
- removed
  - `rb_ractor_gvl`
  - `rb_vm_gvl_destroy` (not used)

There are GVL functions such as `rb_thread_call_without_gvl()` yet
but I don't have good name to replace them. Maybe GVL stands for
"Greate Valuable Lock" or something like that.



**ko1** commented on Apr 16, 2022 • edited — Contributor

Now GVL is not process *Global* so this patch try to use another words.



There are GVL functions such as `rb_thread_call_without_gvl()` yet
but I don't have good name to replace them. Maybe GVL stands for
"Greate Valuable Lock" or something like that.

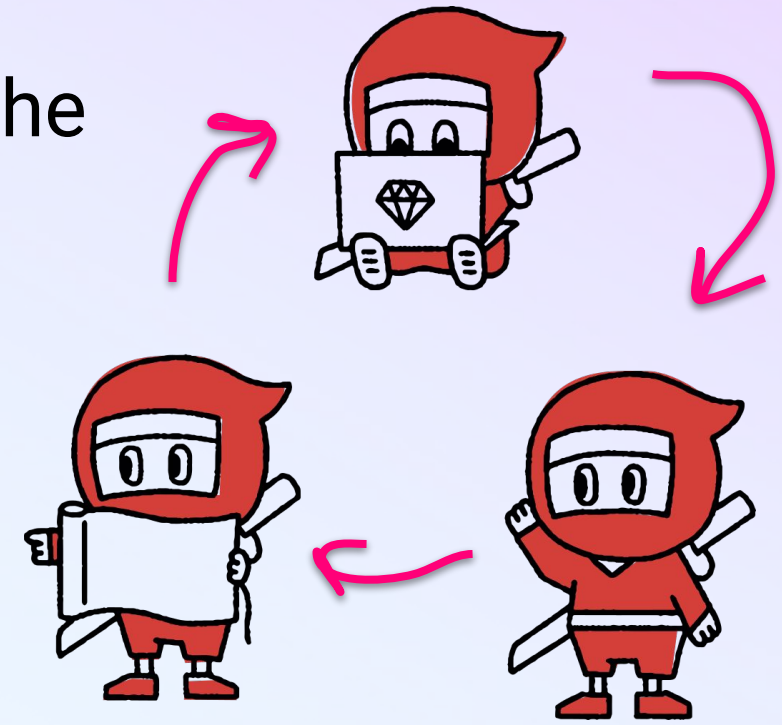
Renamed in Ruby 3.2

*So what is the GVL*

**not?**

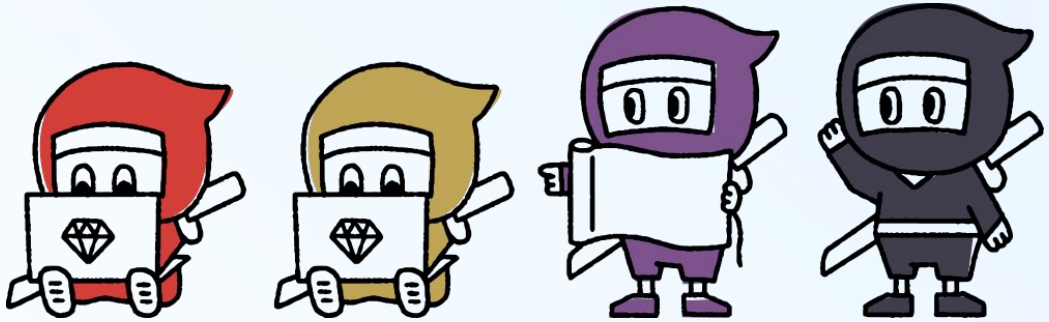
GVL allows **concurrency**

- Illusion of working on multiple things at the same time by switching between them



But not **parallelism**

- Actually working on multiple things at the same time




…like on a 90s single core PC
COMPAQ
COMPAQ
COMPAQ
LLORCA
MALLO
Image credit: …


…like on a 90s single core PC
COMPAQ
COMPAQ
COMPAQ
LLORCA
MALLO
Image credit: …me

If the GVL exists **why** do we need Ruby concurrency APIs?

- `Mutex`
- `ConditionVariable`
- `Queue, SizedQueue`
- `concurrent-ruby` gem
- …

Because the GVL only protects the Ruby VM **data/state**!

Data from your application **may not end up what you wanted**!

```ruby
from = 100_000_000
to = 0

50.times.map do
  Thread.new do
    while from > 0
      from -= 1
      to += 1
    end
  end
end.map(&:join)

puts "to = #{to}"
```

```
$ ruby counter-example.rb
to = 100000011
```

- **Valid** result for Ruby
- Probably **not what we wanted**!

```ruby
from = 100_000_000
to = 0
lock = Mutex.new

50.times.map do
  Thread.new do
    while from > 0
      lock.synchronize do
        if from > 0
          from -= 1
          to += 1
        end
      end
    end
  end
end.map(&:join)

puts "to = #{to}"
```

```
$ ruby counter-example.rb
to = 100000000
```

Code is **correct** now!

# Key Insight #1

- The GVL **does not** protect Ruby code running in multiple threads (only protects the Ruby VM!)*

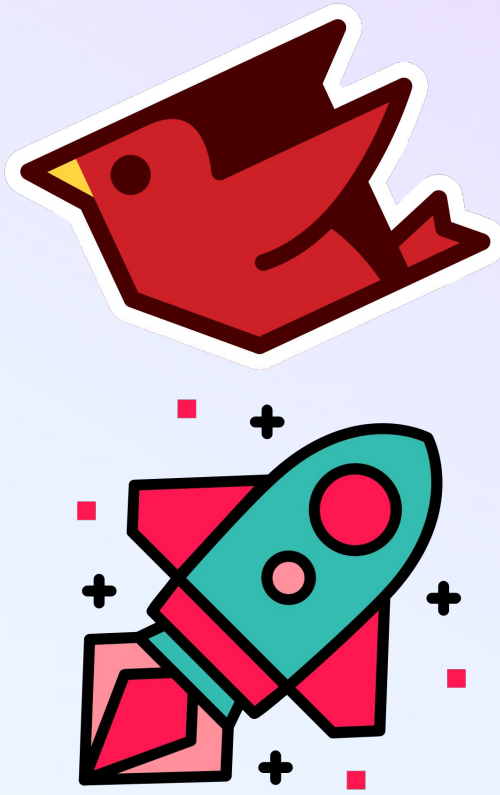
** Except when it does, often by accident or implementation detail that you shouldn't rely on*

# Aside #4

JRuby and TruffleRuby **don’t use a GVL**

…they use other mechanisms to protect the VM data

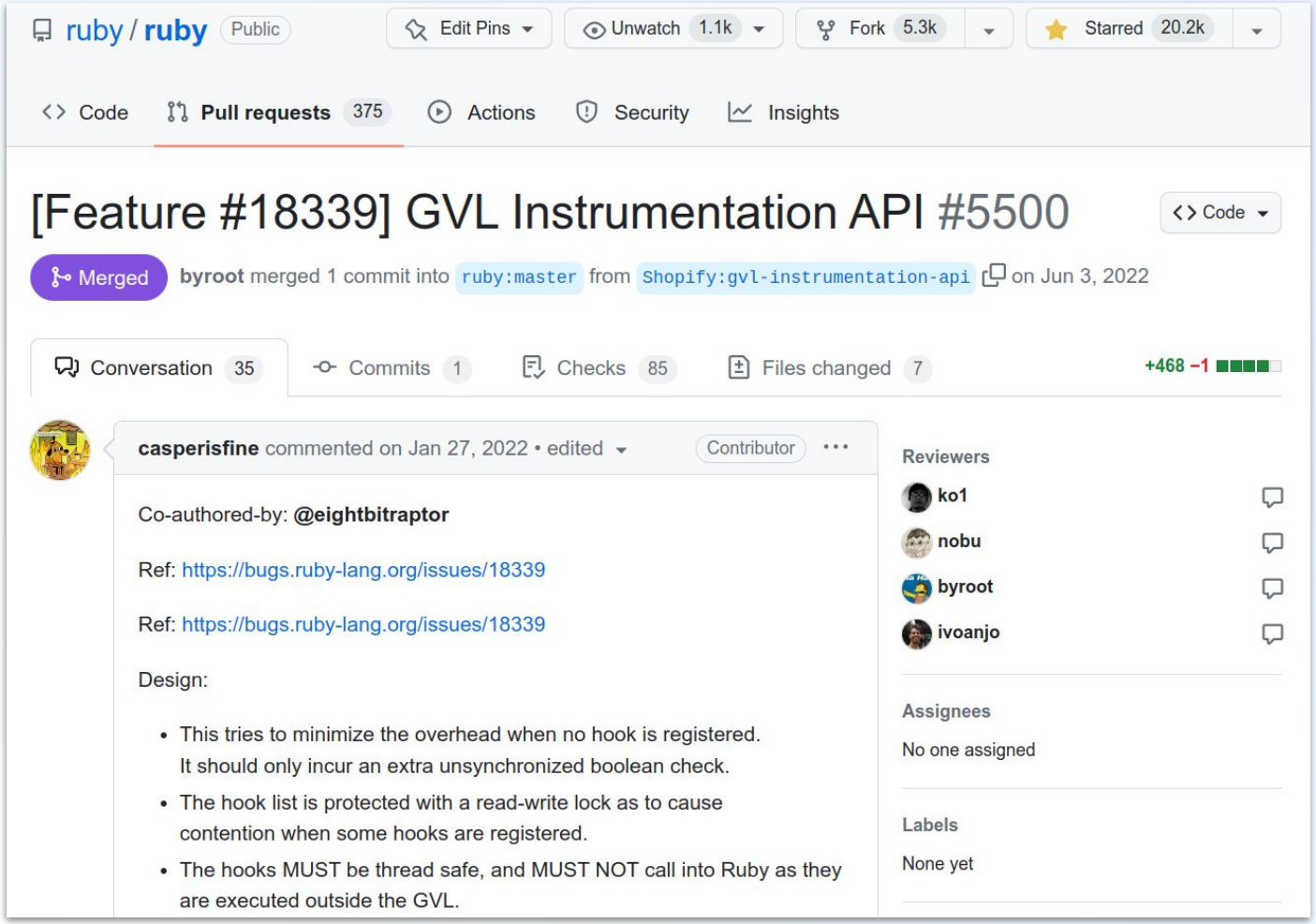
➜ Thus they get **concurrency AND parallelism**



*Observing the*

# Global VM Lock (GVL)

# Observing the GVL




**Ruby 3.2** introduced the GVL Instrumentation API

(Thanks Jean && Matt!)

# Observing the GVL

38 ■■■■■ include/ruby/thread.h

Viewed

```
@@ -190,6 +190,44 @@ void *rb_nogvl(void *(*func)(void *), void *data1,
 */
#define RUBY_CALL_WO_GVL_FLAG_SKIP_CHECK_INTS_

+#define RUBY_INTERNAL_THREAD_EVENT_READY      0x01 /** acquiring GVL */
+#define RUBY_INTERNAL_THREAD_EVENT_RESUMED    0x02 /** acquired GVL */
+#define RUBY_INTERNAL_THREAD_EVENT_SUSPENDED  0x04 /** released GVL */
+#define RUBY_INTERNAL_THREAD_EVENT_MASK       0x07 /** All Thread events */
+
+typedef void rb_internal_thread_event_data_t; // for future extension.
+
+typedef void (*rb_internal_thread_event_callback)(rb_event_flag_t event,
+              const rb_internal_thread_event_data_t *event_data,
+              void *user_data);
+typedef struct rb_internal_thread_event_hook rb_internal_thread_event_hook_t;
+
+/**
+ * Registers a thread event hook function.
+ *
+ * @param[in]  func    A callback.
+ * @param[in]  events  A set of events that `func` should run.
+ * @param[in]  data    Passed as-is to `func`.
+ * @return     An opaque pointer to the hook, to unregister it later.
+ * @note       This functionality is a noop on Windows.
+ * @warning    This function MUST not be called from a thread event callback.
+ */
+rb_internal_thread_event_hook_t *rb_internal_thread_add_event_hook(
+        rb_internal_thread_event_callback func, rb_event_flag_t events,
+        void *data);
+
```

C level API only:
For Ruby gems with native extensions

ivoanjo / **gvl-tracing** Public

Get a timeline view of Global VM Lock usage in your Ruby app

MIT license

60 stars 3 forks

Star Unwatch

Code Issues 4 Pull requests 1 Actions Security Insights ...

master ...

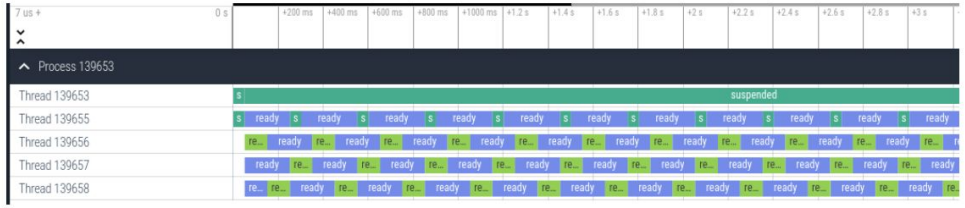
**ivoanjo** Prepare for next development iteration, 1.2.0.dev ... on Mar 15 51

View code

README.adoc

# gvl-tracing

A Ruby gem for getting a timeline view of Global VM Lock usage in your Ruby app that can be analyzed using the Perfetto UI.




See my blog post tracing ruby's (global) vm lock for more details!

| Note | This gem only works on Ruby 3.2 and above because it depends on the GVL Instrumentation API. |
|---|---|

## Quickest start

# `gvl-tracing` gem

Draw GVL usage in a visual timeline

# `gvl-tracing` gem: 3 steps

*Step 1:*

```
gem "gvl-tracing" if RUBY_VERSION >= "3.2."
```

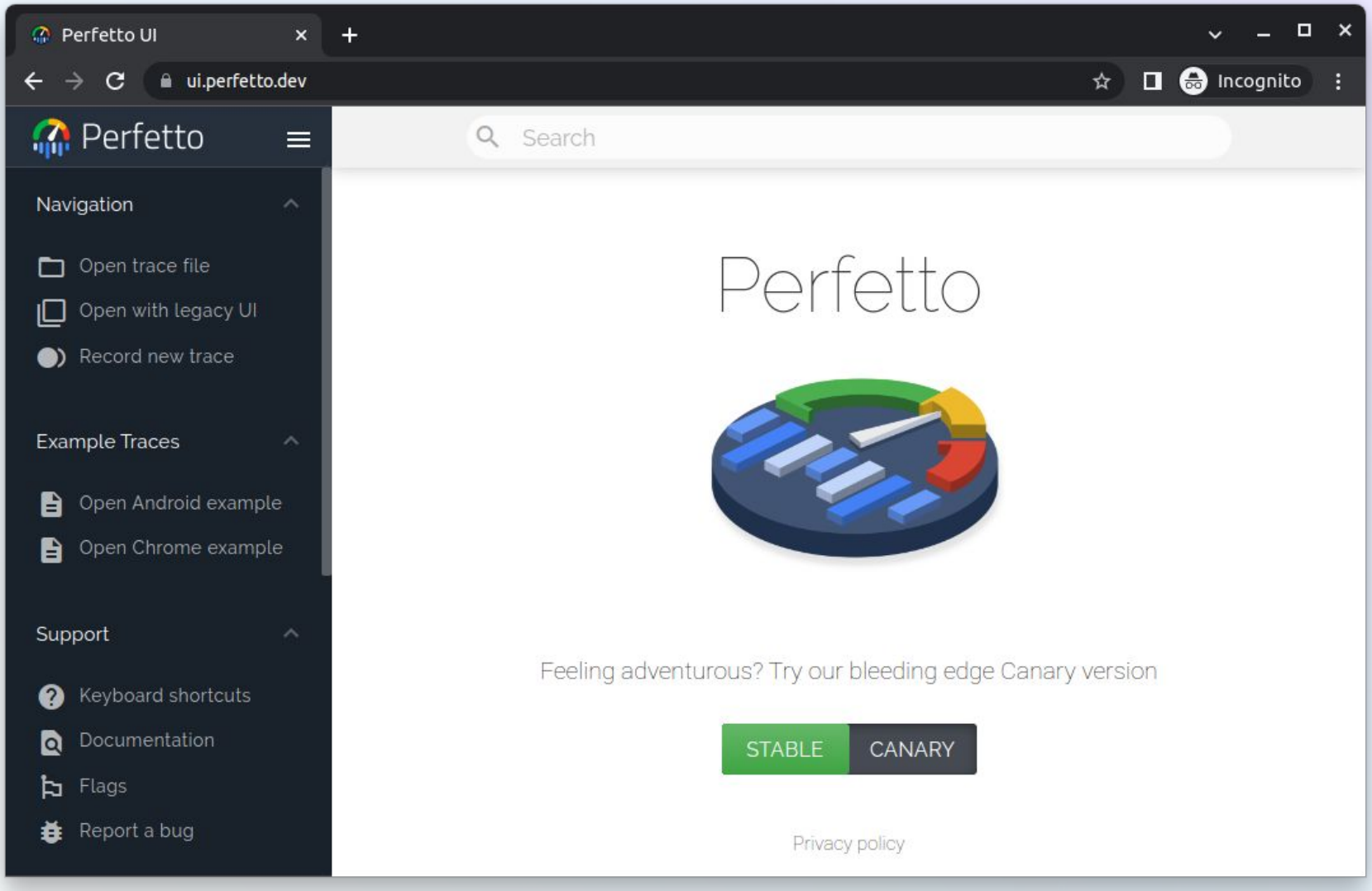
*Step 2:*

```
GvlTracing.start("my_example.json")
# …some code…
GvlTracing.stop
```

*Step 3:*

Go to https://ui.perfetto.dev/ and open `my_example.json`

# `gvl-tracing` gem: 3 steps



**Code** + **perfetto** links @

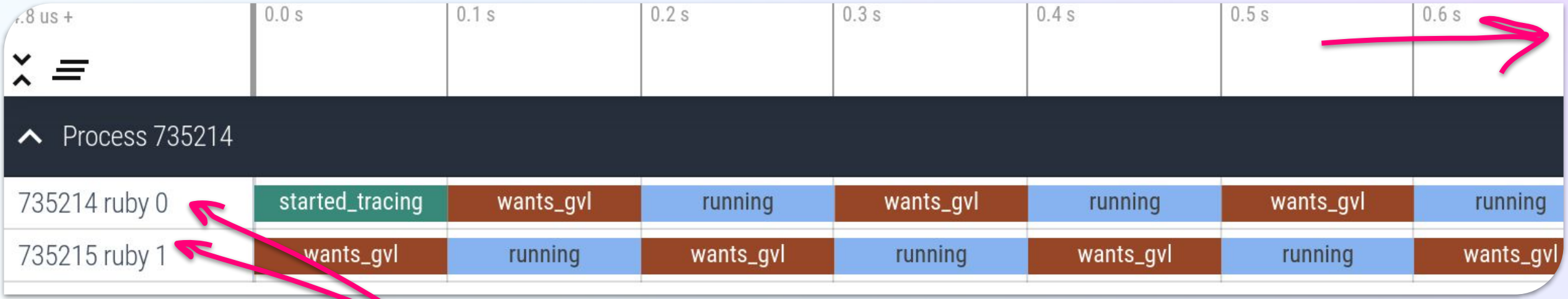
www.ivoanjo.me/rubykaigi2023

*(I'll repeat at the end of the talk,*

*no need to rush to record the link!)*

# Observing the GVL #1

Time flow




Threads

States shown by gem:

started_tracing

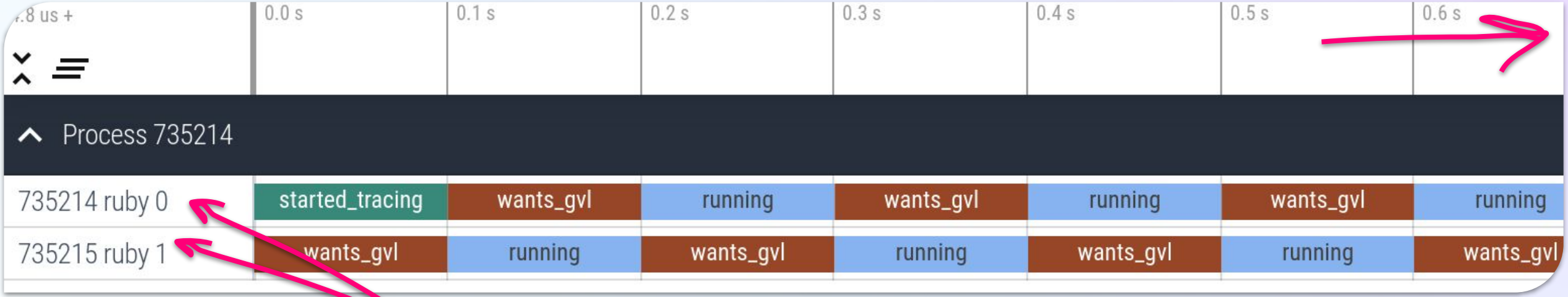
started

died

Starting and stopping of threads/tracing

# Observing the GVL #1

Time flow



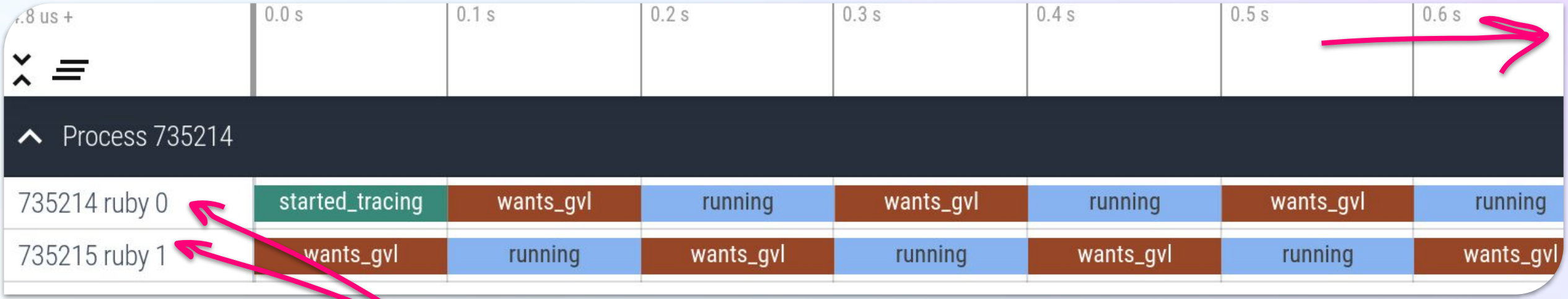

Threads

States shown by gem:

- waiting — Waiting for work / network / IO / sleep…
- running — Doing work!
- wants_gvl — 🔥Has work to do, needs the GVL to do it 🔥
- gc — Garbage collection

# Observing the GVL #1

Time flow

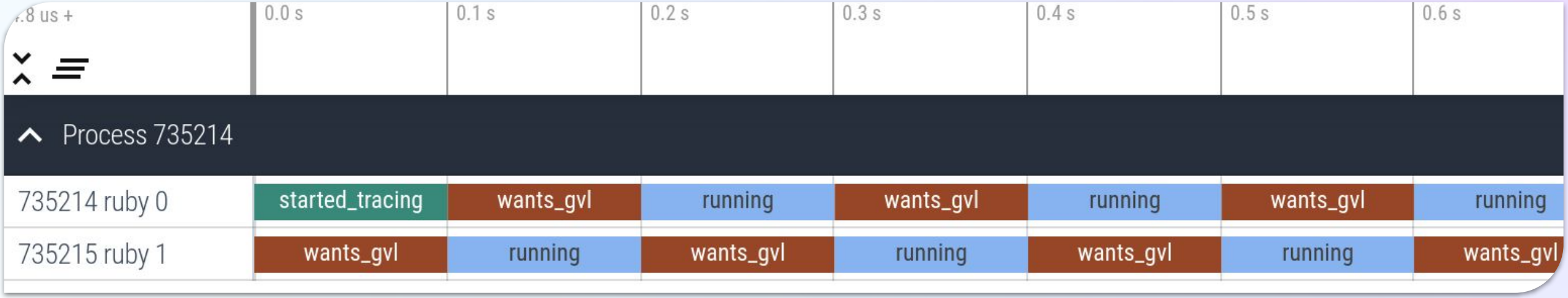



Threads

```ruby
def counter_loop
  counter = 0
  counter += 1 while counter < 1_000_000_000
end

Thread.new { counter_loop }
```

# Observing the GVL #1




Two threads, each incrementing their own counter

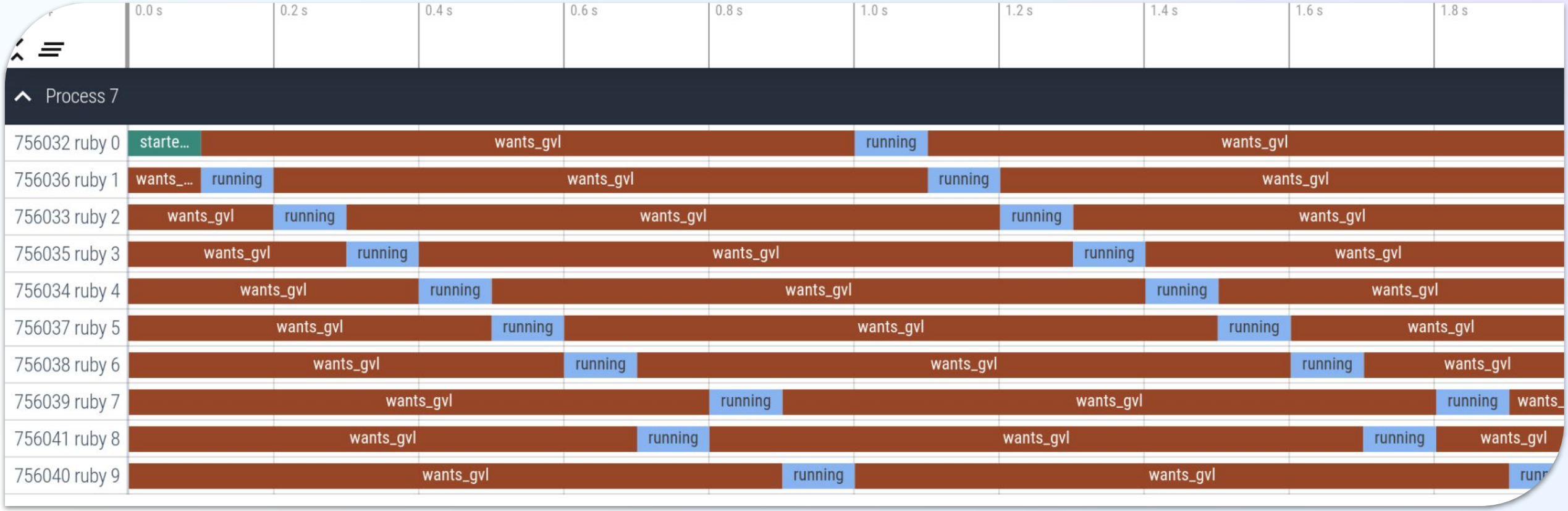
➜ **Concurrency** but not **parallelism**

➜ Threads take turns to execute

➜ When multiple threads want to work, Ruby switches between threads every 100ms

# Observing the GVL #2

Adding more threads to example makes each thread wait longer between executions (2 threads => 10 threads)



# Key Insight #2

Adding more threads to a Ruby app **impacts latency** when multiple threads want the GVL **at the same time**


Runs for 100ms

Waits for 900ms

Runs again...

# Key Insight #2

Adding more threads to a Ruby app **impacts latency** when multiple threads want the GVL **at the same time**


❗ Note that Ruby **web** apps are usually not like this…

…they have network calls (I/O) and not as much processing

# Key Insight #3

➜ If you’re using threads, your app latency **does not depend only** on each request/methods’s performance

➜ Even highly-optimized Ruby code may take longer if it needs to wait for the GVL

# Key Insight #3

*A specific Ruby web request may be slow*

**not because of what the request does**

*but instead*

**because of some other request/thread**

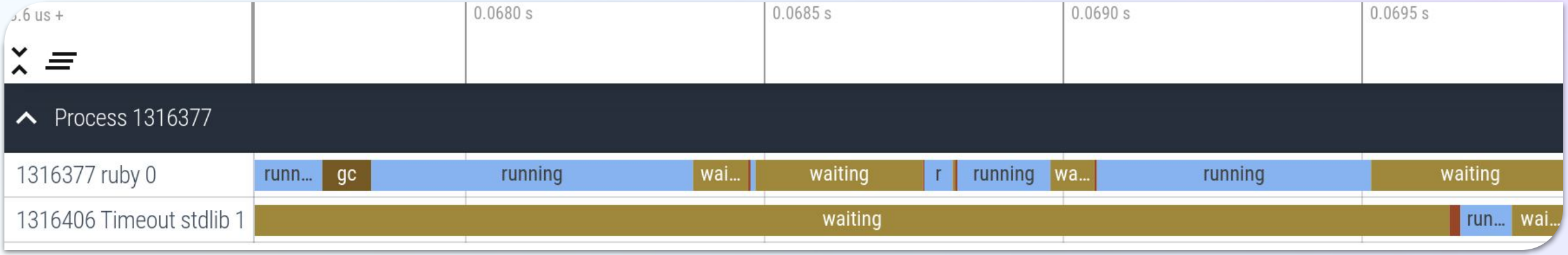
*that was doing a lot of work at the same time*

➜ **Yup, Ruby threads are “noisy neighbours”**

# Observing the GVL #3


When only one thread has work to do, Ruby will give it the GVL for as long as needed

# Observing the GVL #4




When one thread is doing network calls, **it will give up** the GVL while waiting for a response

➜ Other threads can run in the meanwhile!

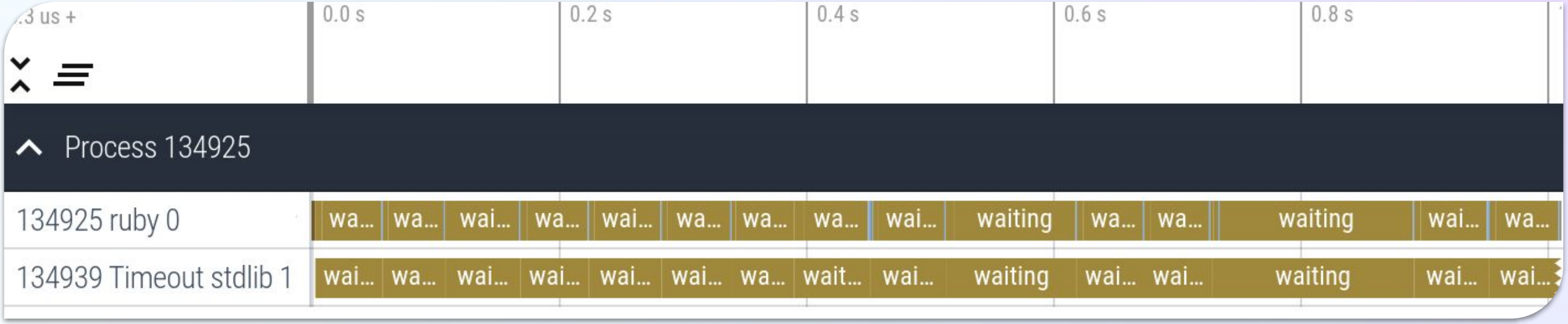
*Threads each get 100ms: that seems*

**fair**

*Are there situations where Ruby is*

**unfair?**

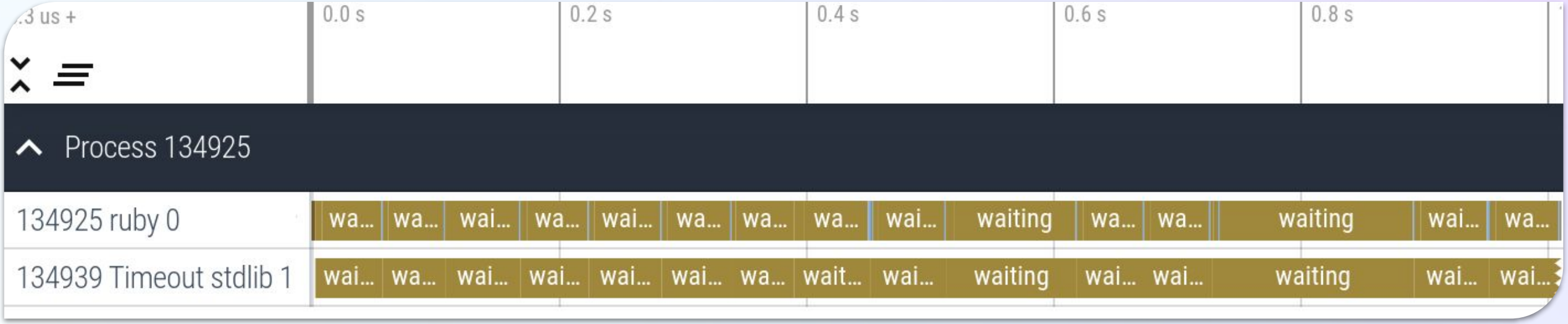
# Observing the GVL #5: Network call example again




```ruby
def perform_request = \
  Net::HTTP.start('www.google.com', ...) { |it| it.get('/') }

Benchmark.ips do |x|
  x.config(time: 1, warmup: 0)
  x.report("request") { perform_request }
end
```

# Observing the GVL #5: Network call example again




```
$ bundle exec ruby rk-example5.rb
Calculating -------------------------------------
request     16.703  (± 6.0%) i/s -     17.000  in   1.022726s
```

# Observing the GVL #6: Adding counter thread

```ruby
def perform_request = \
  Net::HTTP.start('www.google.com', ...) { |it| it.get('/') }

Benchmark.ips do |x|
  x.config(time: 1, warmup: 0)
  x.report("request") { perform_request }
end
```

+

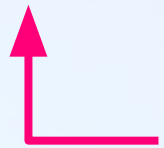
```ruby
def counter_loop
  counter = 0
  counter += 1 while counter < 1_000_000_000
end

Thread.new { counter_loop }
```

= ?

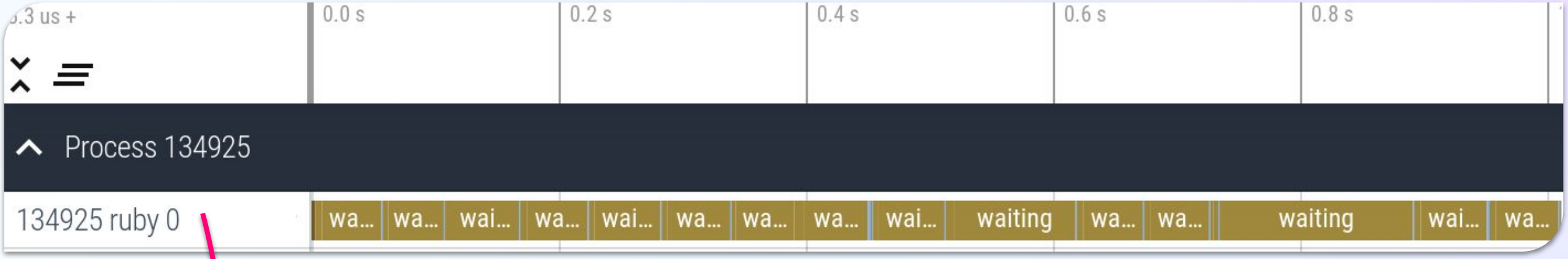
# Observing the GVL #6: Adding counter thread

```
$ bundle exec ruby rk-example6.rb
Calculating -------------------------------------
request      1.450  (± 0.0%) i/s -      2.000  in   1.406056s
```



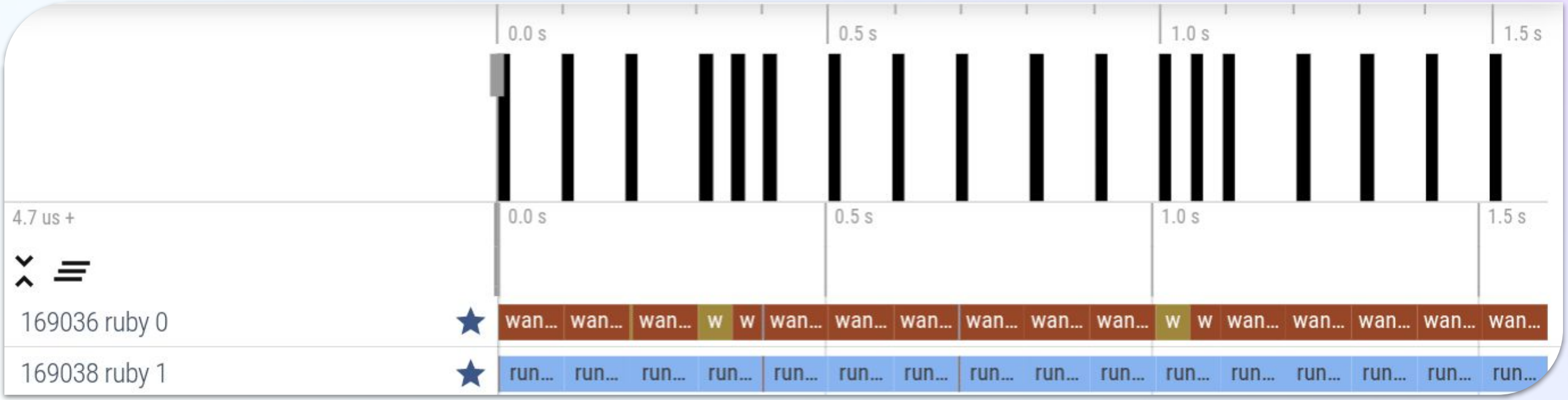
# Observing the GVL #6: Adding counter thread

Before




After

# Observing the GVL #6: Adding counter thread



# Key Insight #4

The Global VM Lock is not always **fair**

# Key Insight #4

*Ruby threads and fibers get you more*

➜ **Utilization**

➜ **Throughput**

*but potentially penalize* ***latency***

# Key Insight #4

Ruby switches threads

➜ Every 100ms spent running

➜ When thread blocks (I/O, network, locks, sleep, pass, …)

➜ At certain points when running C/native code (VM/extensions)

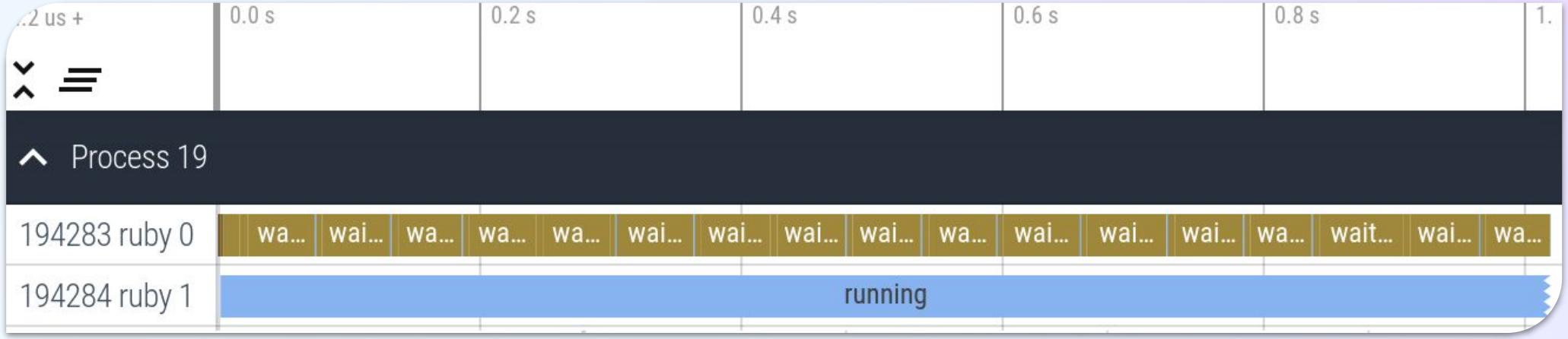
Ractors were introduced in Ruby 3 as a new concurrency primitive

Ractors provide **parallelism** and communicate via **messages**

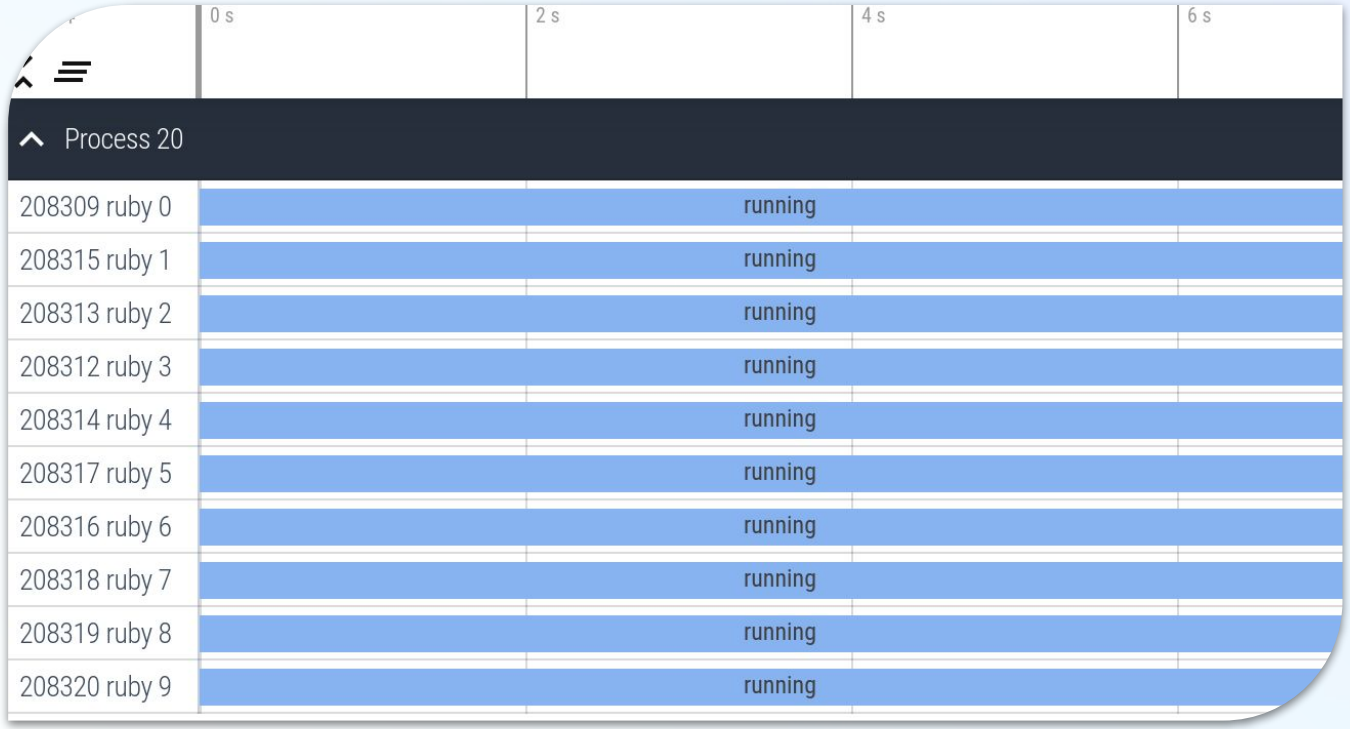
What happens if we modify some of our examples to use Ractors?

# Observing the GVL #7: Ractors!

Network
+
Counters



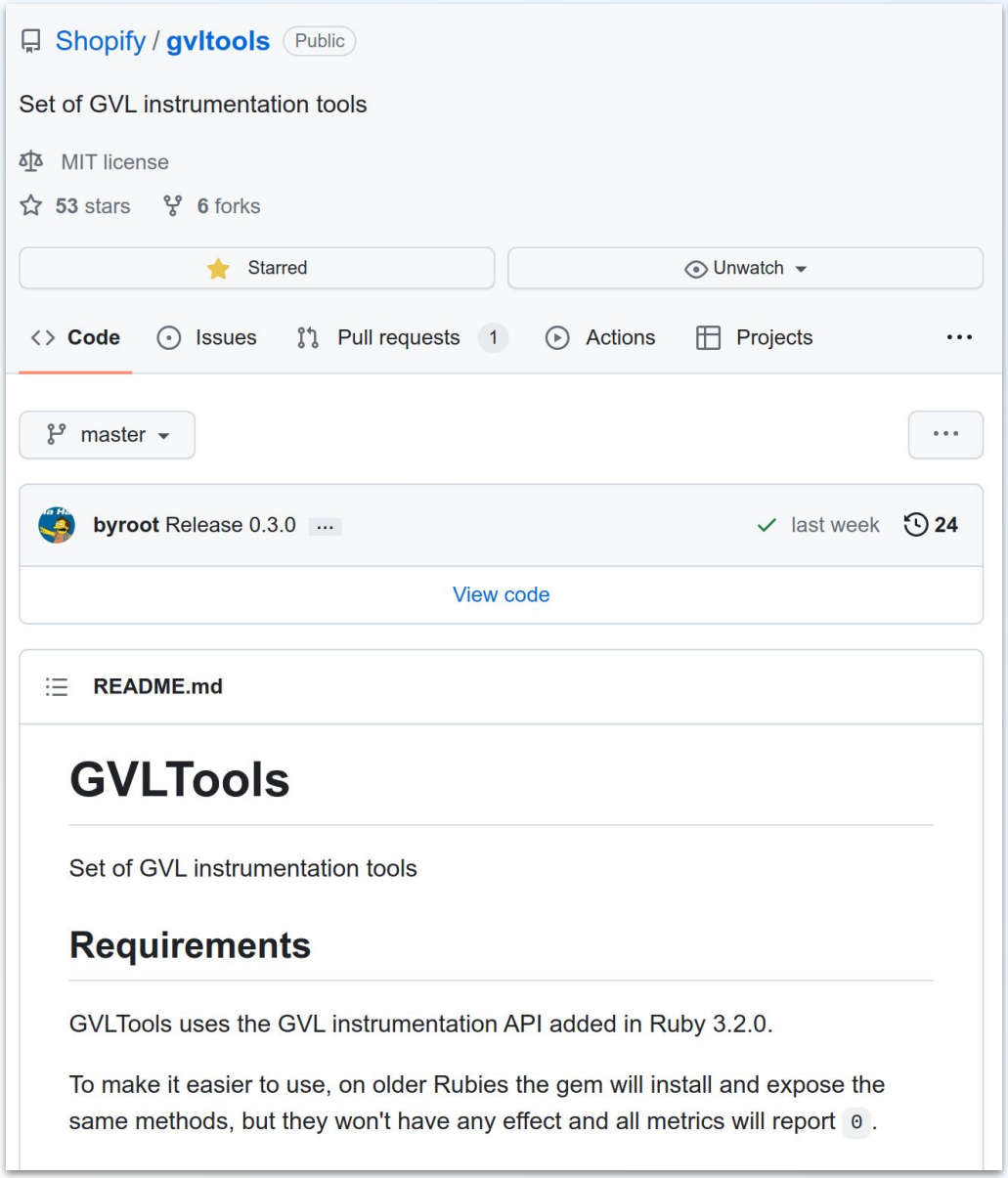

Many
counters



# Key Insight #7

Ractors are **parallel** 🎉

# Bonus: `gvltools` gem




## Provides GVL instrumentation as **metrics**

```ruby
before = GVLTools::GlobalTimer.monotonic_time
# ...
diff = GVLTools::GlobalTimer.monotonic_time - before
puts "Waited #{(diff / 1_000_000.0).round(1)}ms on the GVL"

# => Waited 4122.8ms on the GVL
```

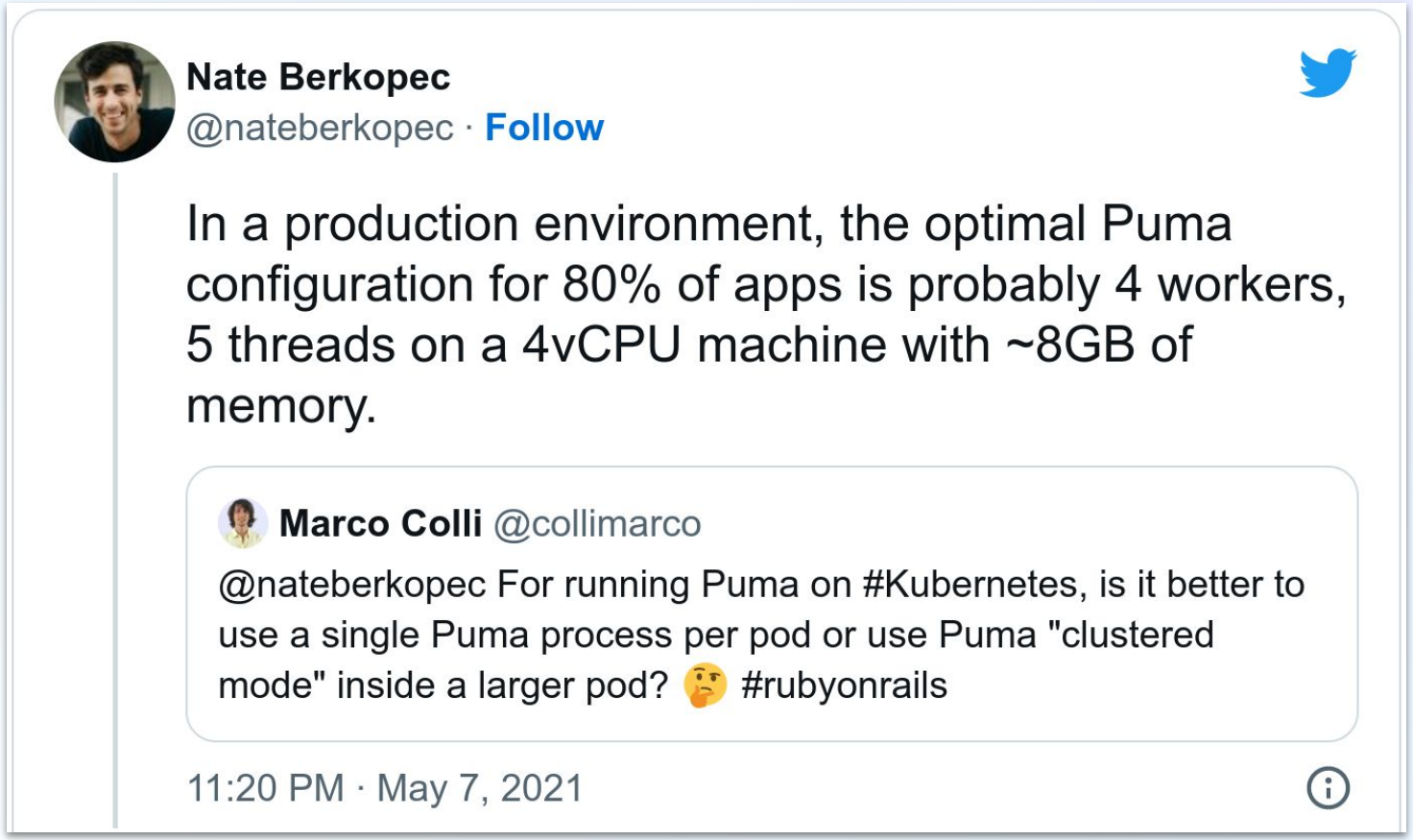
```ruby
Thread.new do
  10.times do
    sleep 0.001
    p GVLTools::WaitingThreads.count
  end
end
```

# What do you do with this information?

**Keep** using threads and fibers in most cases

➜ Worked so far, will keep on working

# **Avoid** using too many threads if you are latency-sensitive



**Nate Berkopec**
@nateberkopec · **Follow**

In a production environment, the optimal Puma configuration for 80% of apps is probably 4 workers, 5 threads on a 4vCPU machine with ~8GB of memory.

> **Marco Colli** @collimarco
> @nateberkopec For running Puma on #Kubernetes, is it better to use a single Puma process per pod or use Puma "clustered mode" inside a larger pod? 🤔 #rubyonrails

11:20 PM · May 7, 2021

**Separate** latency-sensitive code from other things

➜ Remember that performance of a Ruby web app endpoint impacts others in the same app

Don’t guess: **experiment** and **measure**!

➜ Profiling/benchmarking in production (or at least realistic)

➜ Use gvltools gem to measure, gvl-tracing gem to observe

➜ Try puma (or another webserver) in 1-thread-per-process and see how that behaves

➜ Ractors, JRuby and TruffleRuby can **unlock** performance + latency benefits

➜ Always **remember** the GVL is not meant to protect your code (but sometimes does accidentally)

# The future is bright:
# Ruby is always improving and evolving!

- ➜ Ractors
- ➜ YJIT
- ➜ MaNy project (M:N threading)
- ➜ 10ms thread switch?

# Thank you

**Ivo Anjo**
ivo.anjo@datadoghq.com / @knux
www.ivoanjo.me/rubykaigi2023 ➜➜➜

Datadog is **hiring!**