# Ractor on Ruby 3.4

Koichi Sasada
STORES, Inc.

# Today’s topics

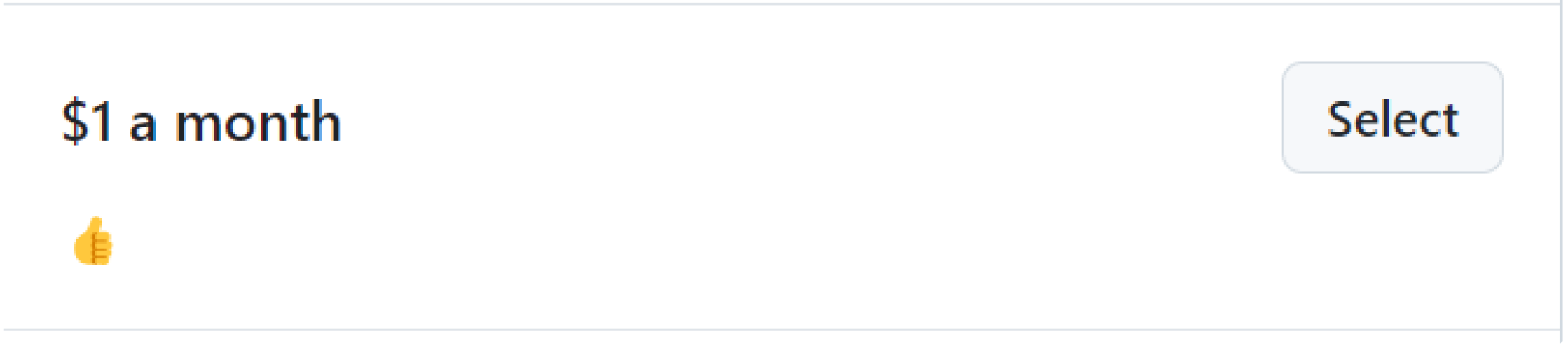
- What is and why “Ractor”?
- What is current situation of Ractor?
- What’s new on Ruby 3.4?
- What’s next?

**Apologies** 😢

**This year I was unwell for a long time, so I didn't achieve much. For that reason, there is a lot of background talk. Please listen to my talk as a refresher.**

# Koichi Sasada

- Ruby interpreter developer employed by **STORES, Inc.** (2023~) with @mametter
  - YARV (Ruby 1.9~)
  - Generational/Incremental GC (Ruby 2.1~)
  - Ractor (Ruby 3.0~)
  - debug.gem (Ruby 3.1~)
  - M:N Thread scheduler (Ruby 3.3~)
  - …
- Ruby Association Director (2012~)
- Ruby Hack Challenge workshop (tomorrow)
  - 12:00-13:30 Ask the Ruby Committers session




https://github.com/sponsors/ko1

Shortest description ever.

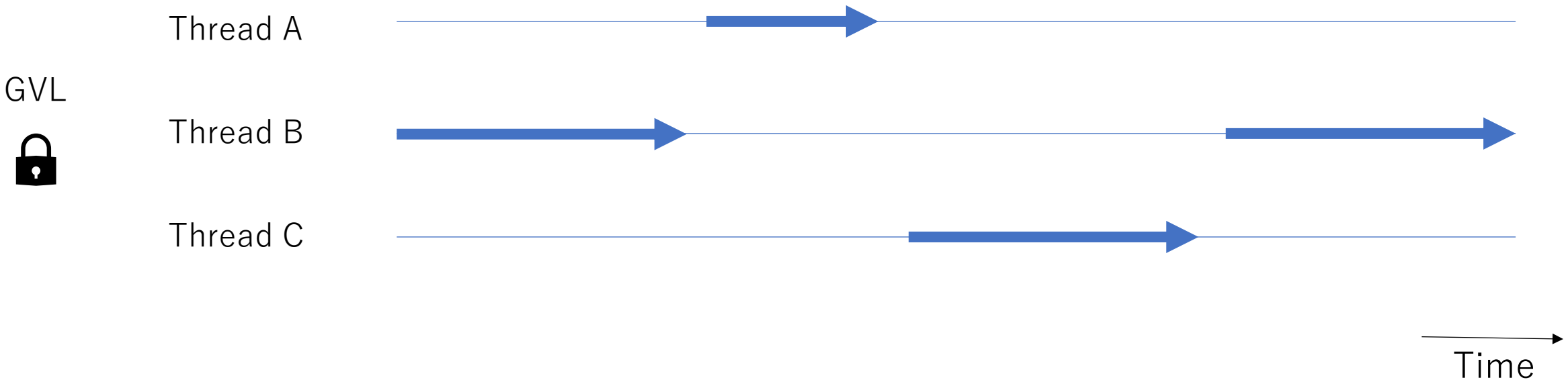
# What is and why “Ractor”?

# “Ractor” is

- introduced from Ractor 3.0
- designed to enable
  - 😄 **parallel** computing on Ruby for more performance on multi-cores
    - It can make faster applications
  - 😄 **robust** concurrent programming
    - No bugs because of object sharing

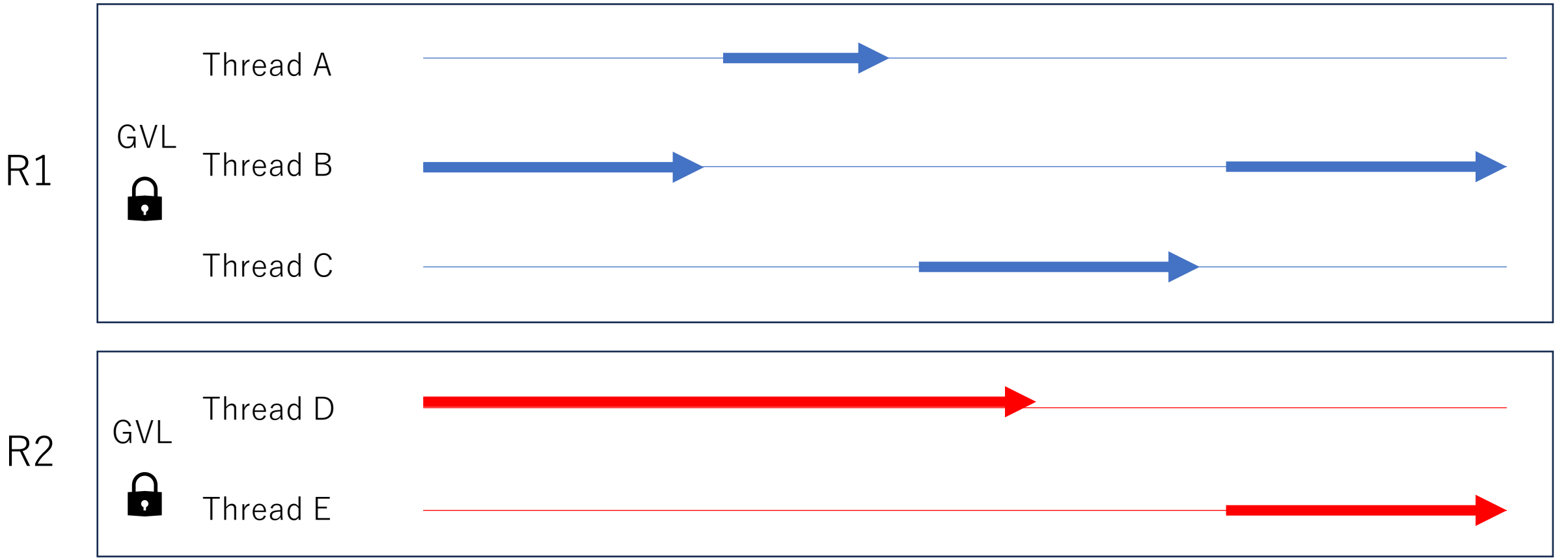
# What is parallel?

- Threads of Ruby (CRuby) doesn't run them in parallel
  - Only one thread acquiring "GVL" can run at a time
  - Even if the computer (CPU) has multiple cores



# What is parallel?

- Threads of different Ractors can run in **parallel**
  - They can utilize CPU cores on your machine



# Takeuchi function on 4 Ractors

```
def tarai(x, y, z) =
  x <= y ? y : tarai(tarai(x-1, y, z),
                     tarai(y-1, z, x),
                     tarai(z-1, x, y))
```

```
require 'benchmark'
Benchmark.bm do |x|
  # sequential version
  x.report('seq'){ 4.times{ tarai(14, 7, 0) } }

  # parallel version
  x.report('par'){
    4.times.map do
      Ractor.new { tarai(14, 7, 0) }
    end.each(&:take)
  }
end
```

```
      user     system      total        real
seq 53.674715   0.001315  53.676030 ( 53.676282)
par 57.916671   0.000000  57.916671 ( 14.544515)
```

x 3.7 faster!! 😆

**Write this kind of code in Ruby!!**

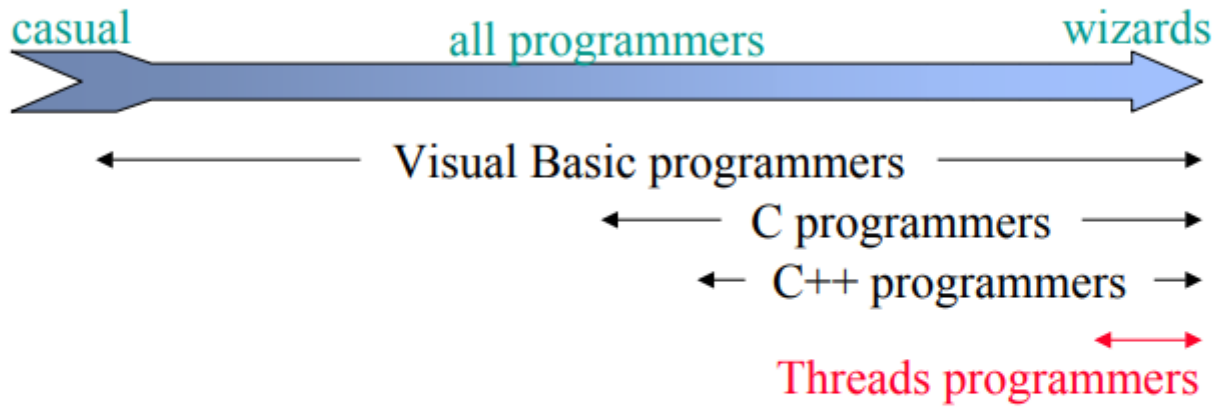
# BTW: GVL

- GVL **had meant** “Global/Giant VM Lock”, per VM lock
  - GIL (Giant interpreter lock) in Python
  - We use the term “VM” because we tried multiple VMs in one process and wrote some academic papers
    - Sub-interpreter in Python now a day
- With Ractor, GVL is not “Global”. Giant?
  - Each Ractor has GVL
  - The Ruby VM has many GVL now a day
- Good Valuable Lock?
  - We uses the term “GVL” in C API and don’t change them ☺
  - Feel free to tell me if you have a good idea what GVL stands for

# Why not parallel on threads?
# 1. Difficulty of programming

- Writing parallel programming is too hard in general
- We need to synchronize all **shared mutable objects** between threads
- On threads, it is very **easy** to share an object, but it is hard to trace, furthermore we can not trace over the libraries
- It is same on current (C)Ruby's concurrent threads, but the context switch point is very limited, so the degree of difficulty is better than parallel threads.

# What's Wrong With Threads?




- **Too hard for most programmers to use.**
- **Even for experts, development is painful.**



"Why Threads Are A Bad Idea (for most purposes)" by John Ousterhout (1995)

# Why not parallel on threads?
# 2. Quality of the Ruby interpreter

- 2.1 Reliability of the interpreter
  - (C)Ruby doesn't allow to stop by critical error (SEGV, for example)
  - Accessing all mutable objects should be **thread-safe** in parallel run
    - Array, Hash, String, … many basic Ruby objects written in C
  - Thanks to GVL, we don't need to care such difficult things
  - Ruby is OSS and development productivity is important
- 2.2 Performance of the interpreter
  - Introducing thread-safety needs fine-grained locking and it introduces overhead

BTW: Python community decided to tackle this difficult issue ([PEP 703]).

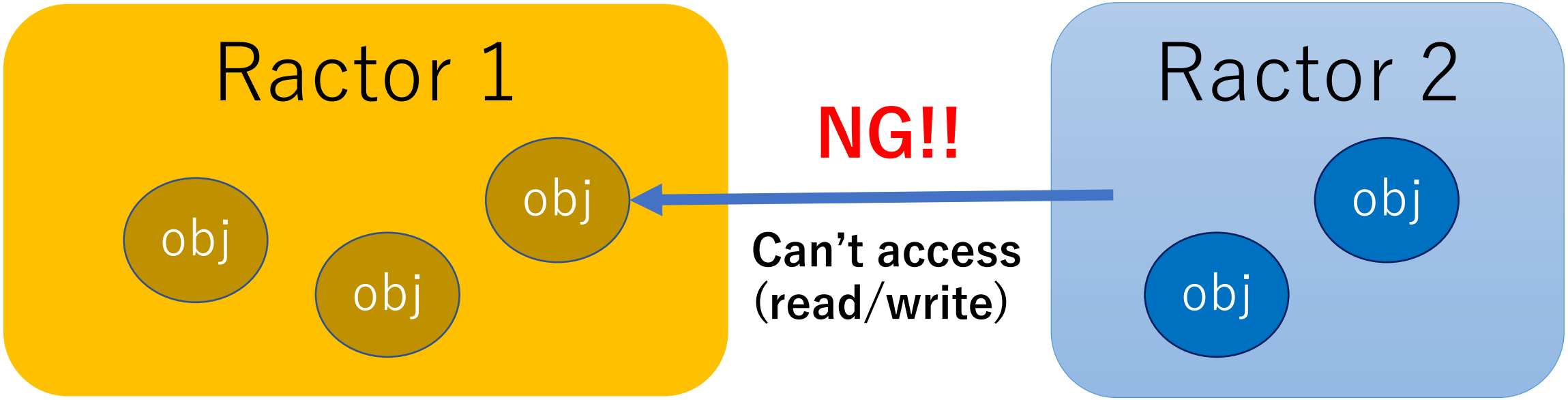
I respect the decision and I'm looking forward to see the conclusion.

# Again: Why parallel thread programing is difficult?

- Difficulty to manage **shared mutable objects** between threads
  - Off course there are other reasons, but it is one of biggest issue.
- Ideas
  - **Prohibit mutable objects!**
    → Erlang, Elixir, functional languages
  - **Trace them by type!**
    → Rust and other ownership support languages
  - **Mange/limit mutations!**
    → Clojure, transactional memory libraries
  - **Trace all mutation and locking by tooling**
    → thread sanitizer (clang), helginrd (valgrind), …
  - **Prohibit sharing objects!**
    - → Forking processes (shell), micro-services, dRuby, Ruby MVM project, …
  - **Separate shareable and unshareable objects! → Ractor!!!**

# Ractor, an isolated object space

- Introduce Ractor as isolated object heap



# Ractor, a better choice of thread-safety

- Split **all objects** into “Shareable” and “Unshareable”
  - Most of objects are “Unshareable” (Array, String, …)
  - Some special shareable objects
    - Immutable objects
      - **The object is frozen**, and **they only refers shareable objects**
    - Classes/Modules
    - Special cared objects such as Ractor, TVar and so on
- We **only need** to make shareable objects thread-safe

# But wait, how to share the state…?

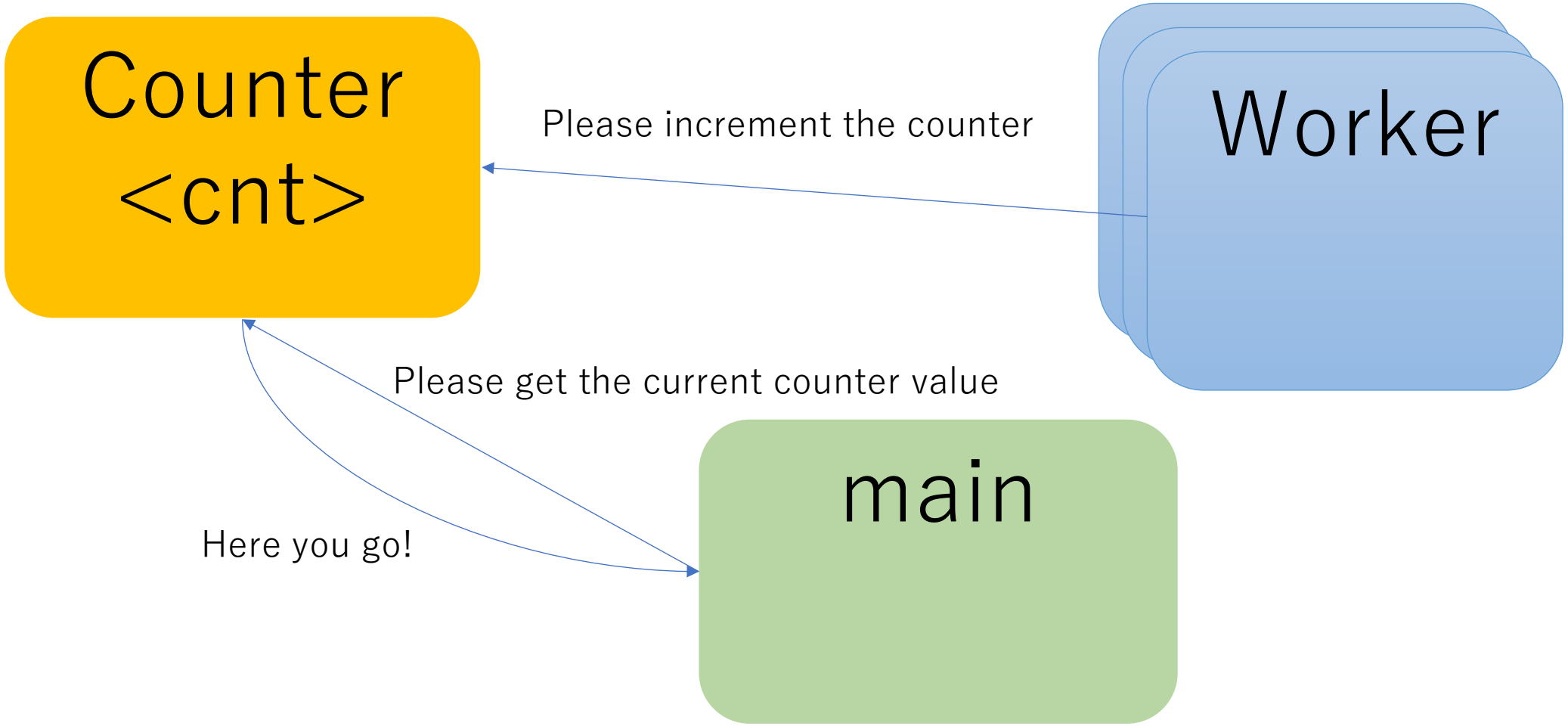
```
# easy example
cnt = 0
WorkerNum.times{
  Thread.new{ N.times{do_task(); cnt += 1} }
}
…
p cnt #=> Total count of “do_task()” called

# Note: “cnt” should be protected by Mutex
#       This is why thread programming is difficult
```

# How to share the state between Ractors?

- 3 options (but there are more)
  1. Use external database (RDB, redis, etc)
  2. Use “Actor” programming model (Ractor was comes from)
  3. Use “TVar”, Transactional Variable

# 2. “Actor” programming model
# Enclose the state in the Actor



# “Actor” programming model
# Enclose the state in the Actor

```
Counter = Ractor.new{
  cnt = 0
  loop{
    case Ractor.receive
    when :get
      Rator.yield cnt
    when :inc
      cnt += 1
    end
  }
}
```

```
WorkerNum.times.map{
  Ractor.new{
    N.times{
      do_task()
      Counter.send :inc
    }}}
…
p Counter.send(:get).take
```

# Using TVar – Transactional Variable

- “Transactional” is come from DB and research from STM
- Optimistic locking
  - If the thread safety was broken (mutate by other threads), then roll-back (rerun)
- Composable (nested transactions run atomically)
- ractor-tvar.gem (Advertisement)
  - Transactional variable library for Ractors
  - It is guaranteed to keep the mutation atomically

# TVar by ractor-tvar.gem

```
Counter = Ractor::TVar.new(0) # 0 is init value
WorkerCounts.times.map{
  Ractor.new{
   N.times{
     do_task()}
     Counter.atomically{ Counter.value += 1 }
     # Counter.increment is also supported for short
   }}}
…
p Counter.value
```

# What is current situation of Ractor?

# Good news

- Ractor was released with Ruby 3.0!!
- Ractor supported debug.gem aim to support Ractor
- M:N threads for lightweight Ractors/Threads creation
- Few usage reports which achieve performance improvements
- … (and detailed additional features and improvements)

# Bad news

- 😢 Ractor supported debug.gem aim to support Ractor, but not supports yet…
- 😢 M:N threads for lightweight Ractors/Threads creation, but not tuned yet…
- **Only Few** usage reports which achieve performance improvements

- Difficulties of Ractor – We need to change the usage of Ruby

→ Not enough Ractor supporting libraries

- Quality of Ractor implementation

# Strict Ractor rules to make safer concurrent programming

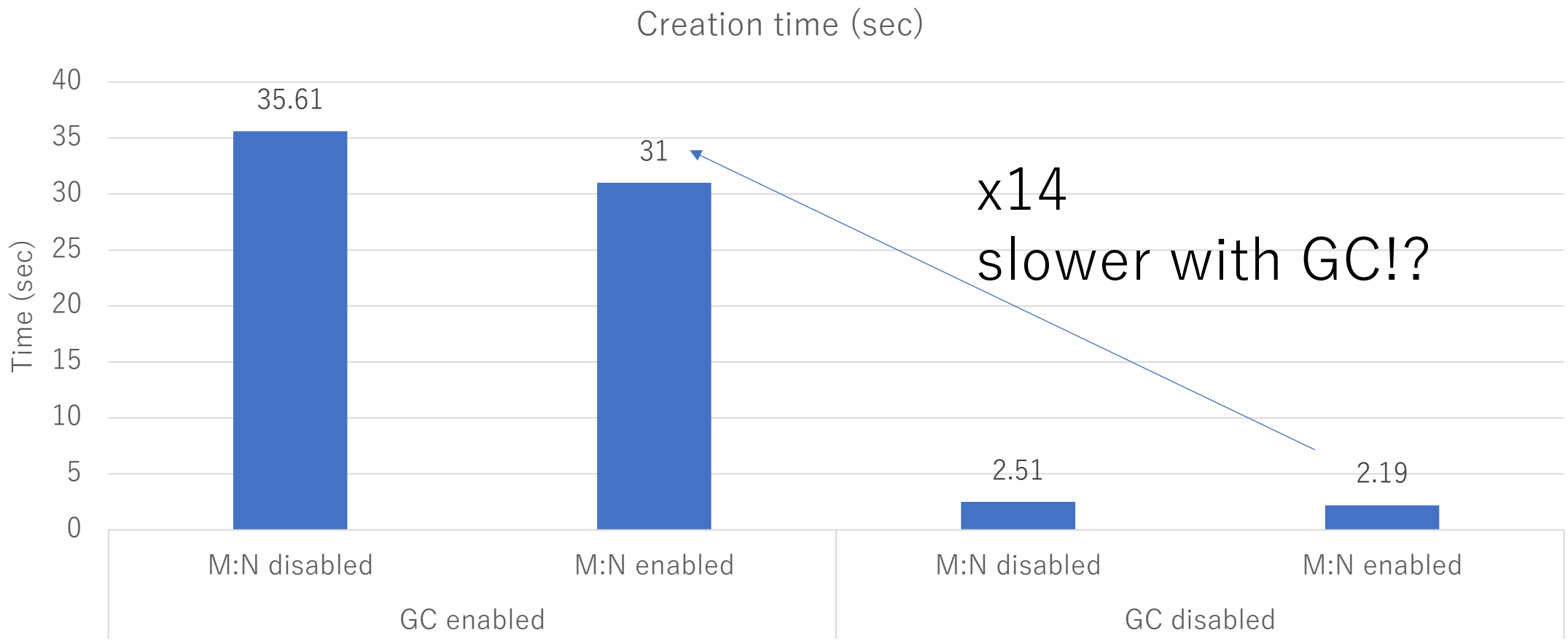
- 😟 Limiting object sharing features between Ractors
  - Unshareable and shareable objects
    - Unshareable objects – most of objects
    - Sharable objects – some special objects
  - Constants (and so on) can not get/set unshareable objects by child Ractors (= non main Ractors).
  - Global variables are not accessible from child Ractors.
  - …
- We need to rewrite existing libraries
  - For example, Rails doesn’t run at all
  - We need huge effort to run Rails on Ractors.

# Simple example which doesn’t run

```
Ractor.new do
  pp 1
end

# because
#  the first pp require ‘pp’ library
#  but require on a Ractor is prohibited
```

# Ring example benchmark results




Data from “M:N Thread implementation on Ruby”, PPL2024

We have many issues
We need to improve step by step

# What’s new on Ruby 3.4?

# Off-topic
# Ruby 3.4 feature: Unused block warning

```
# Have you written such code?
p foo do
  …
end

# and “foo” doesn’t receive a block
# it is easy to detect if foo raises an exception,
# but not easy if foo change the behaver
# if it accepts a block or not
```

# Off-topic
# Ruby 3.4 feature: Unused block warning

- From Ruby 3.4, if a block will be passed to a method which doesn't seem to use a block, show warning on "-w"

```
def foo
end
foo{}
#=> t.rb:5: warning: the block passed to
'Object#foo' defined at t.rb:2 may be ignored
```

# Off-topic
# Ruby 3.4 feature: Unused block warning

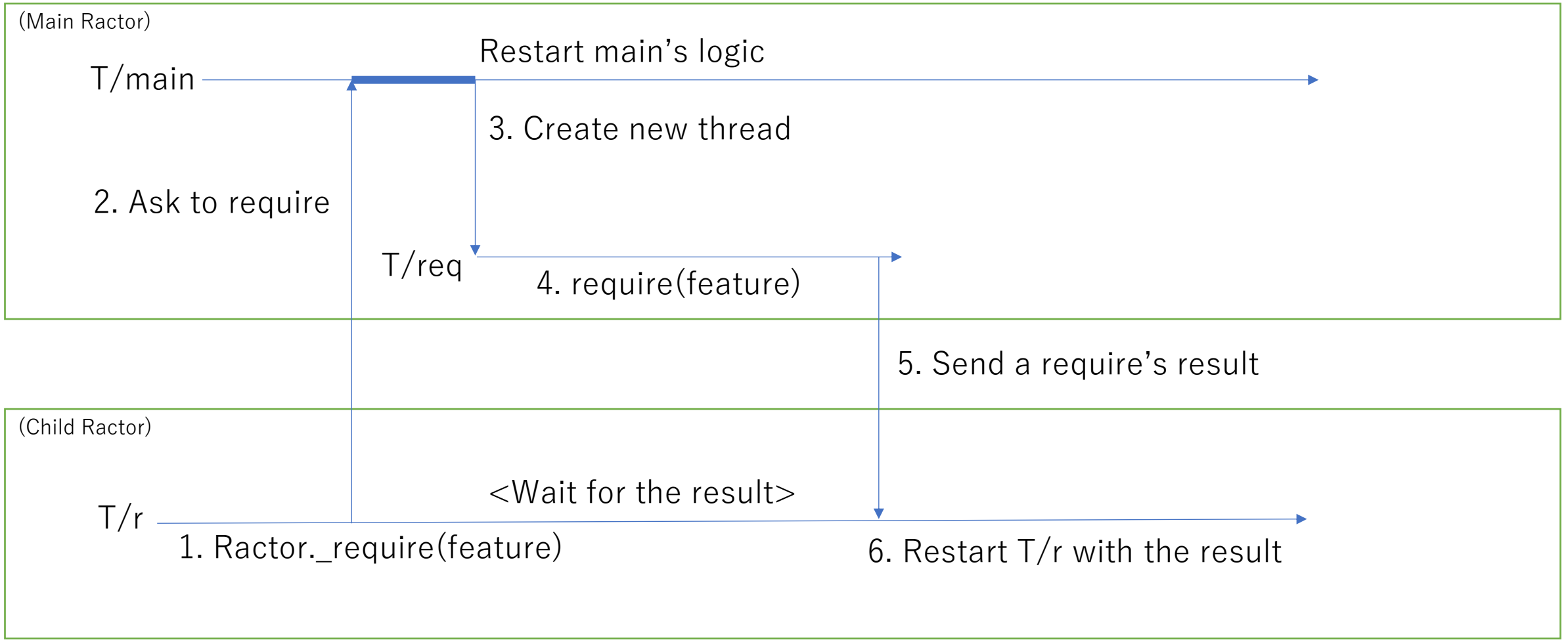
- There are many false positive on duck typing, so we relaxed the warning condition: Warn when there is not a method which has same name and accepts a block
- If you want to warn without this condition, use `Warning[:strict_unused_block] = true`

```
def foo = nil
def "".foo = yield
foo{} #=> No warning
```

# Require on Ractors

- “pp” is good example
- Autoload is also good example to require on Ractor
- And Ruby 3.4 can support “Require on Ractors”!!
- Issues
  - $LOAD_PATH, $LOADED_FEATURES are mutable data
  - Many library expect to run on the main Ractor
    - `STR = "str"` is not allowed in non-main Ractors
  - Rubygems and other many tools can run only on main ractor
- Solution
  - Ask main Ractor to require and wait the result → Ractor._require()

# Require on Ractors
# `Ractor._require(feature)`



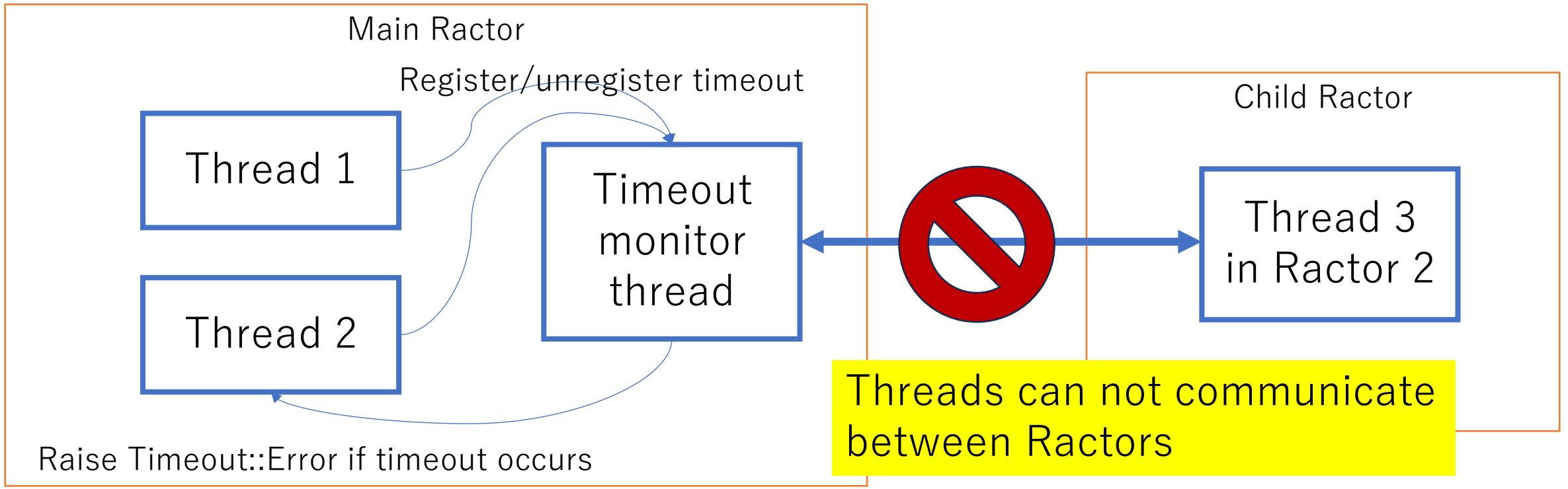
# Prepend `Kernel#require` to use `Ractor._require`

```
Kernel.prepend Module.new do
  def require(feature)
    return Ractor._require(feature) unless Ractor.main?
    super
  end
end

# and all of require(feature) checks main ractor or not
# Note: prepend when the first child ractor was created
```

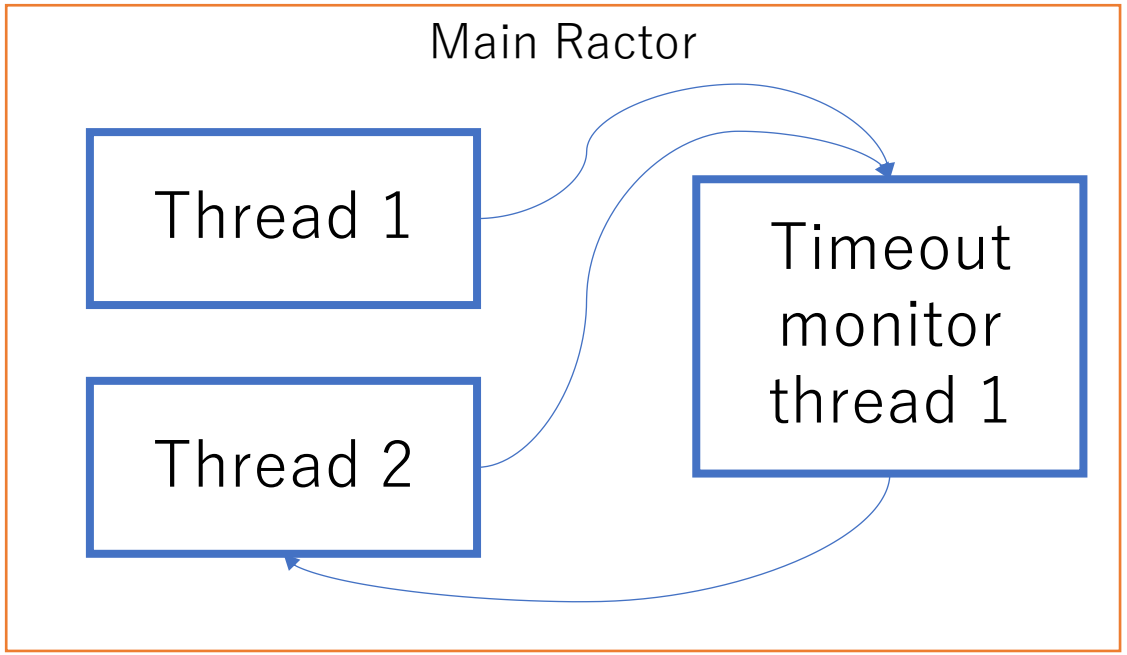
# Issue of “timeout”

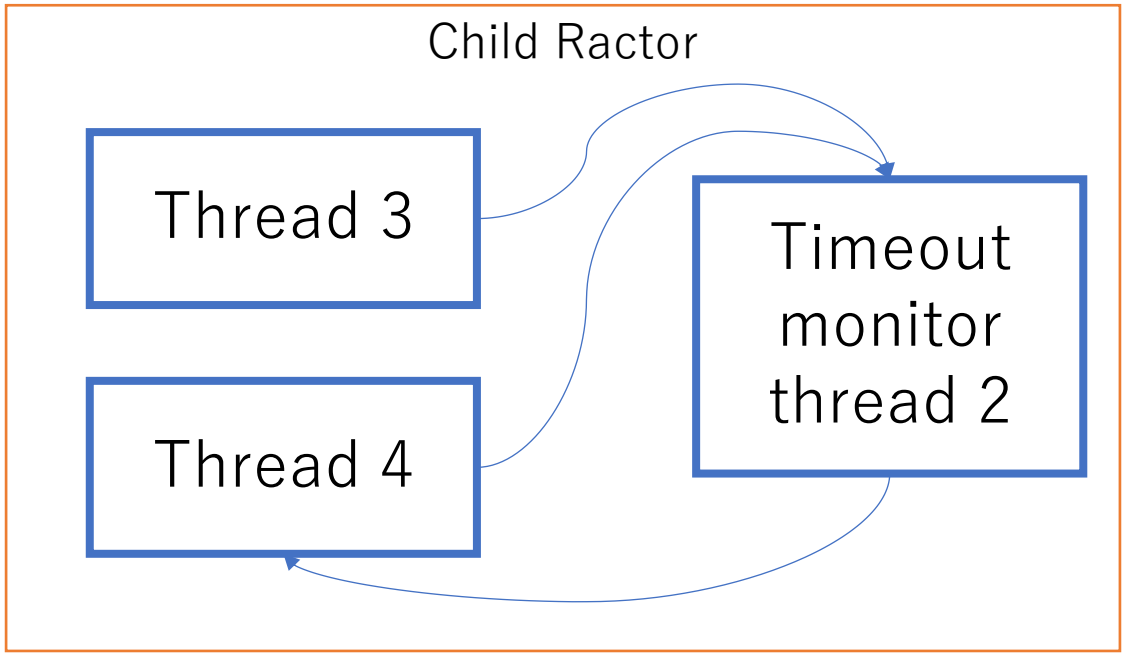
- “timeout” library uses Thead to send asynchronous exception to the timeout thread
- Can not communicate between Ractors



# “timeout”
# Prepare a monitor per each Ractor

- Provide monitor threads per a Ractor
- Easy to implement
- Not introduced yet, but I hope Ruby 3.4 has this change



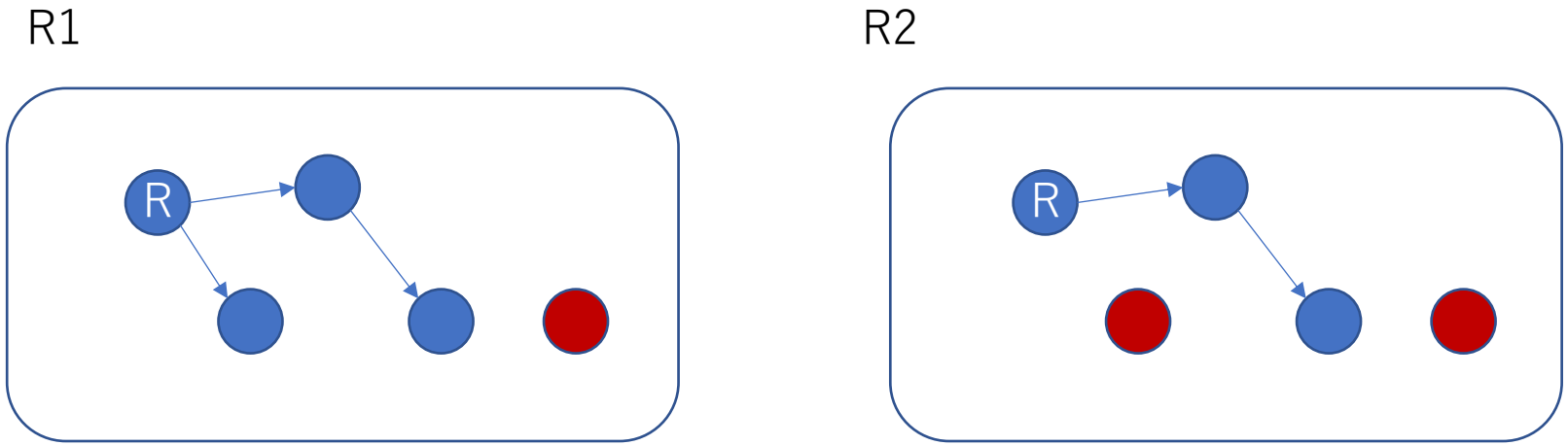
# Others

- Newly introduced methods
  - `Ractor.[]/[]=`, not yet but hopefully `Ractor.local_storage_init`
  - `Ractor.main?`
- (Not yet) Thread supports
  - Many Ractor operations (`Ractor.select` and so on) doesn't support threads (only 1 thread can run them), but it is not healthy

# What’s next?

# Future
# GC tuning

- Ractor aware GC tuning
  - Prepare enough pages for the number of Ractors
- Ractor local GC
  - Need distributed GC techniques
  - Need more memory vs. single heap



# Today's topics

- What is and why "Ractor"?
- What is current situation of Ractor?
- What's new on Ruby 3.4?
- What's next?

Questions and feedbacks are very welcome!

Thank you for listening, Koichi