Ractor Enhancements, 2024

Koichi Sasada
STORES, Inc.

STORES

Today’s topics

- Support important features on Ractors
  - “require”
  - “timeout”
- Memory management issues on Ractors
- Future enhancement plans
  - GC strategy
  - Proposed APIs

STORES

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~)

STORES

Glad to be here again
this year!

“Ractor” is

- introduced from Ruby 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

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
      - Immutable objects
      - Some special objects
        - Class/Modules
        - Ractor 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.
  - …

Issues on Ractors

- 😢 Lack of important features
  - “require” on child Ractors
  - “timeout” on child Ractors
  - …
  - …
- 😢 Performance degression
  - on memory management
  - …
  - …

Issue of “require”
Child Ractors can not call “require”

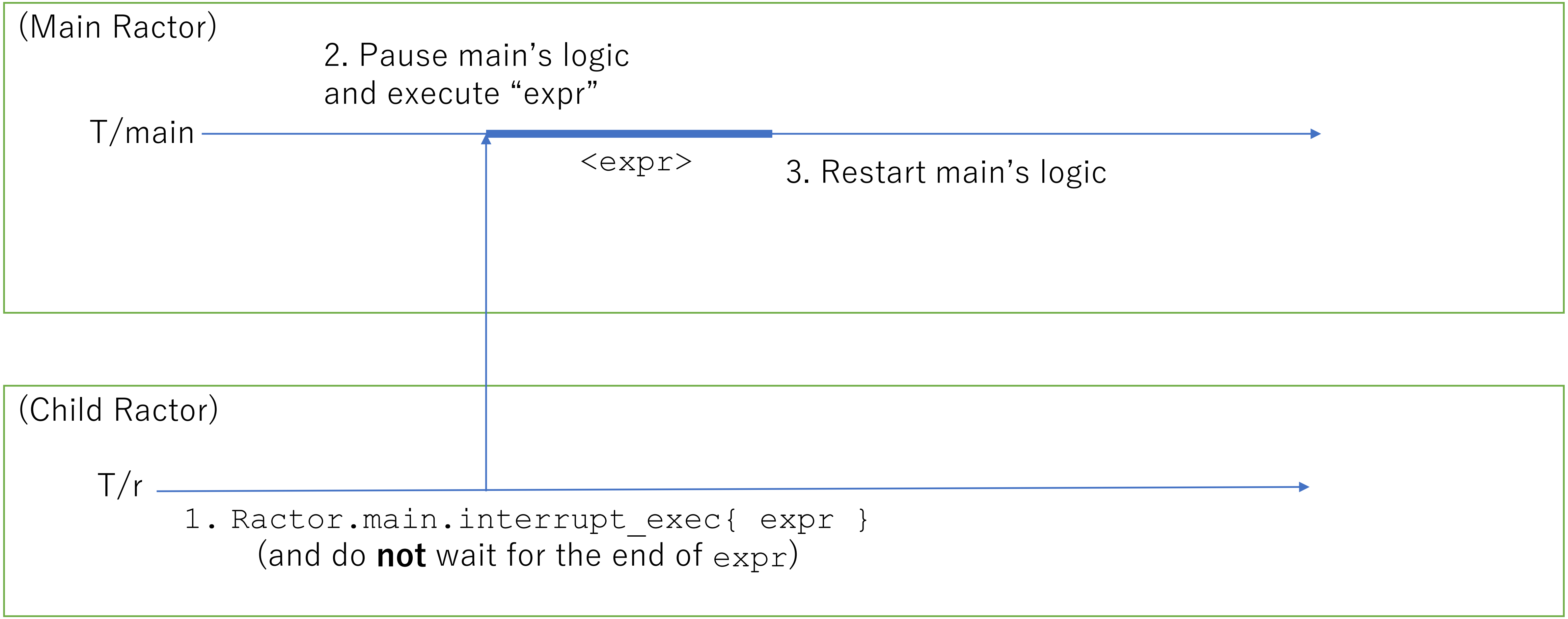
- Some code requires library in a method on child Ractors
  - “autoload” case
  - `def foo = (require “foo”; Foo.foo)`
    - “pp” case – require “pp” when the first “pp” is called
  - So, we need to allow “require” on child Ractors
- “require” is prohibited by child Ractors because
  - It access $LOAD_PATH, $LOADED_FEATURES and so on
  - The loaded code can define constants with unshareable objects

    ```
    STR = “str” # set an unshareable object
    ```
  - Complex logics on RubyGems
- **“require” should be done on main Ractor**

“require”
Solution: “require” on main Ractor

- Introduce new API “**`Ractor#interrupt_exec{ expr }`**”
  - Run `expr` on a receiver Ractor’s main thread asynchronously
    - The block will be translated to the shareable Proc (can’t access outer scopes)
    - The return value of `expr` will be ignored so should be sent explicitly
    - It likes trap handler and sending a signal (therefore it is danger too)
  - The main thread will be interrupted any methods such as IO blocking and so on (like signal handling)
- `Ractor.main.interrupt_exec{ $g=1 }` runs “`$g=1`” on the main thread of the main Ractor
  - Useful to access resources which are limited to main Ractor
  - Need some overhead to interrupt main thread

Implement “require” with `Ractor#interrupt_exec`



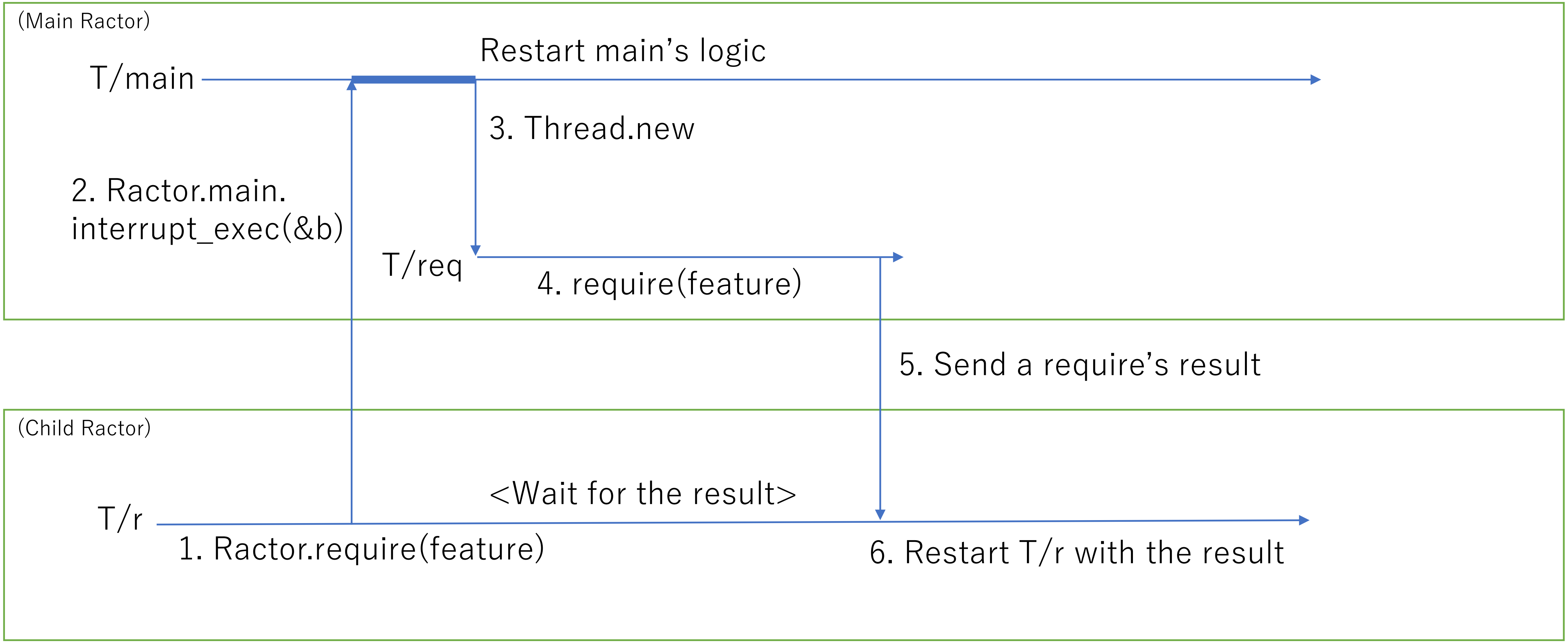
Implement “require” with `Ractor.require(feature)`

```
class Ractor
  def self.require(feature)
    c = Ractor::Channel.new
    Ractor.main.interrupt_exec do
      Thread.new do
        c << require(feature)
      rescue Exception => e
        c << e
      end
    end
    c.take
  end
```

```
def Ractor::Channel.new =
  Ractor.new{loop{ Ractor.yield Ractor.receive }}
```

- Ask main Ractor to require
- Call “require” on another thread because of recursive lock (dead lock)
- Caller Ractor should wait for the result of “requie”

Implement “require” with `Ractor.require(feature)`



Implement "`Kernel#require`" with `Ractor.require`

```
module Kernel
  def require(feature)
    return Ractor.require(feature) unless Ractor.main?

    # original require on main Ractor
  end
end

def Ractor.main? = Ractor.current == Ractor.main
```

Issue of Ractor supported "require"

- Some library override "Kernel#require()"
  - Rubygems
  - Bundler
  - …
- All of them need to insert non-main Ractor guard

```
def require(feature)
  return Ractor.require(feature) unless Ractor.main?
  …
```

- Can we ask to add this line at the beginning of all overriding definitions?

Issue of Ractor supported “require” Provide prepended module?

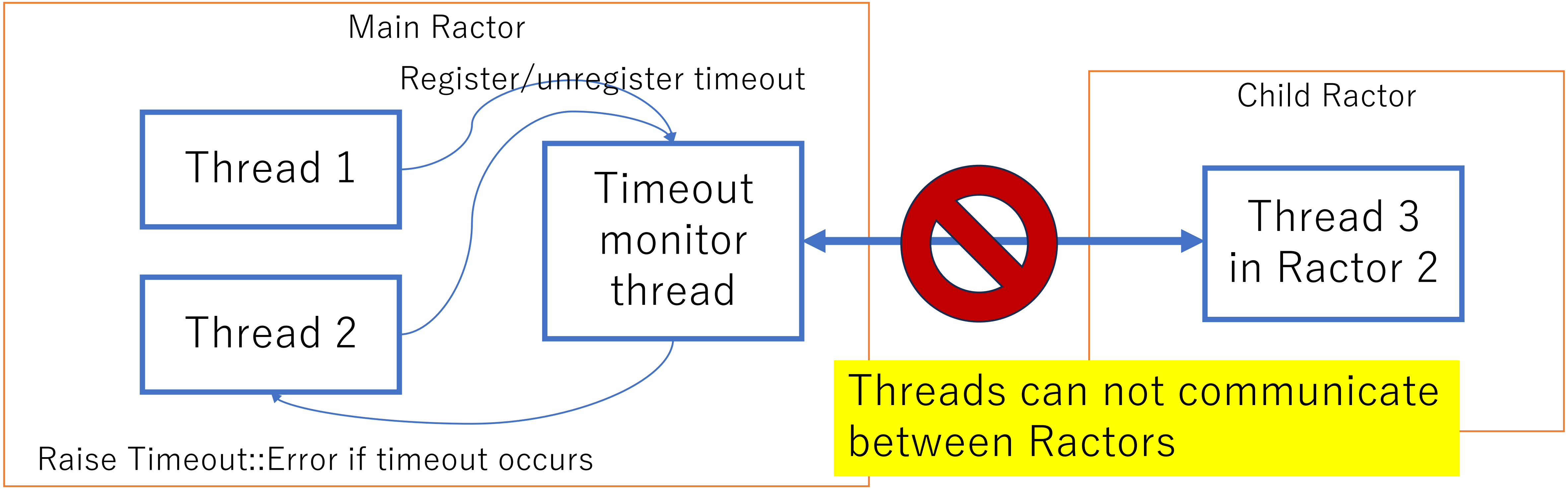
```
# prepend a module to Kernel can solve it
module RactorAwareRequire
  def require(feature) =
    Ractor.main? ? Ractor.require(feature) : super
end
module Kernel
  prepend RactorAwareRequire
end

# but all ancestors of classes contains it
p ''.class.ancestors
# => [String, Comparable, Object, RactorAwareRequire, Kernel, BasicObject]
```

Off-topic `Ractor/Thread#interrupt_exec`

- This feature is useful for debuggers to stop all threads
  - Current debugger doesn't support Ractors → Key feature for it
  - Current debugger implementation using "line" TracePoint to stop all threads, but we can't stop threads running "blocking operation" (I/O waiting and so on) and can not access to the thread information.
- This feature is danger like trap handlers because it can be interrupts any code such as cleanup code in ensure
  - With great power comes great responsibility
  - Difficult to introduce Ruby's features? (C-API?)

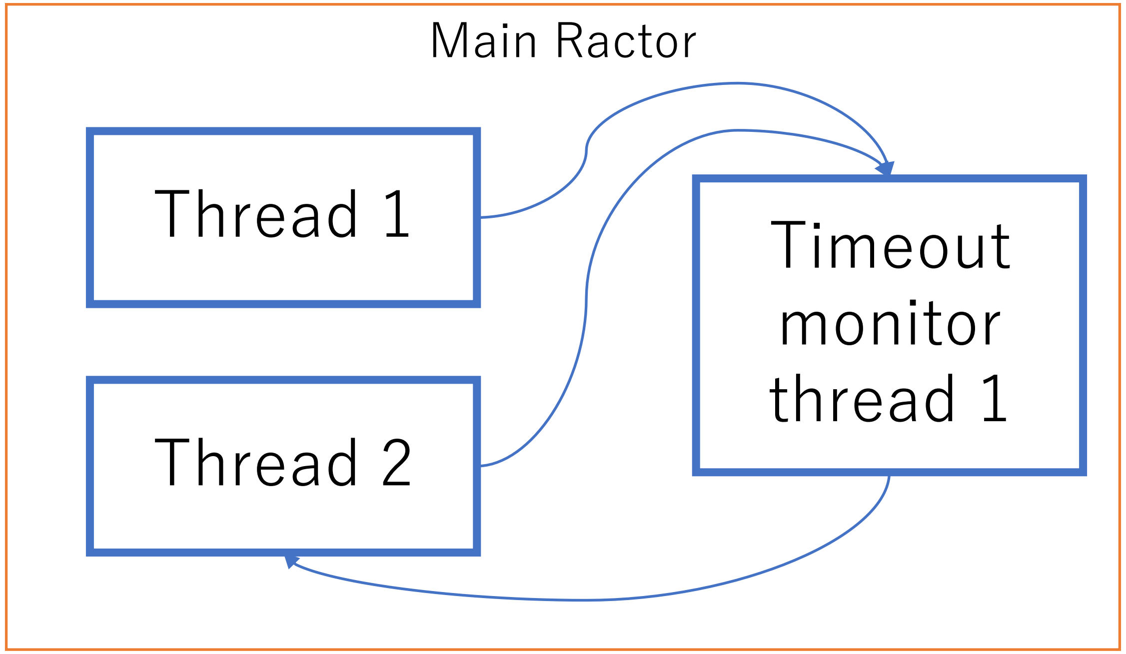
Issue of "timeout"

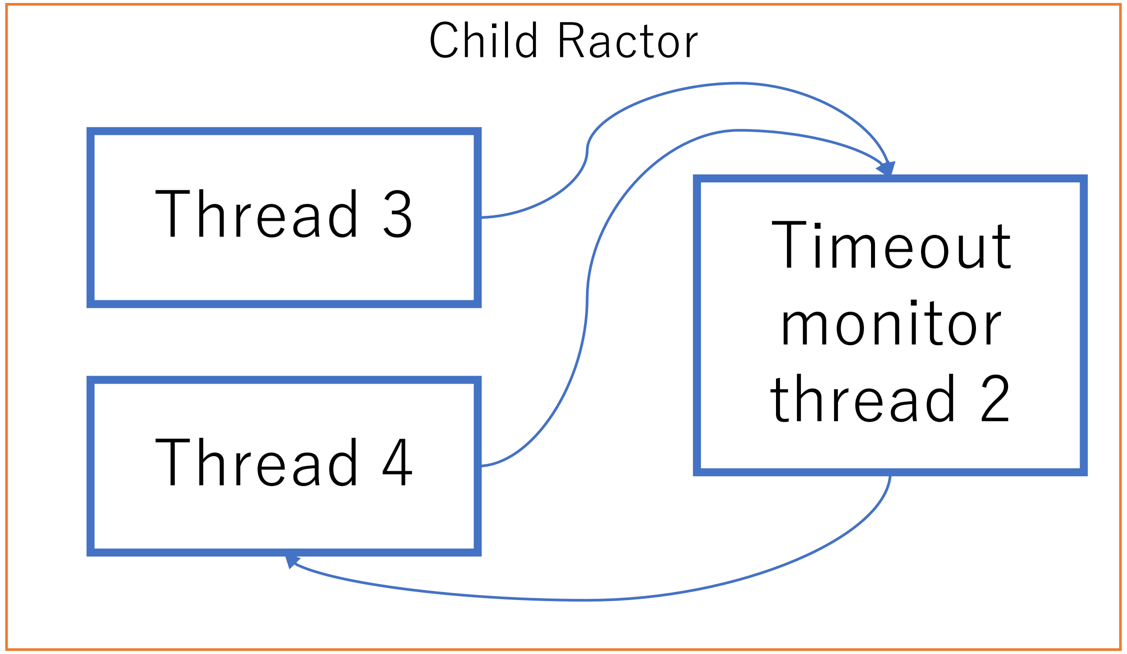
- "timeout" library uses Thead to send asynchronous exception to the timeout thread
- Can not communicate between Ractors



"timeout"
Solution 1: Prepare a monitor per each Ractor

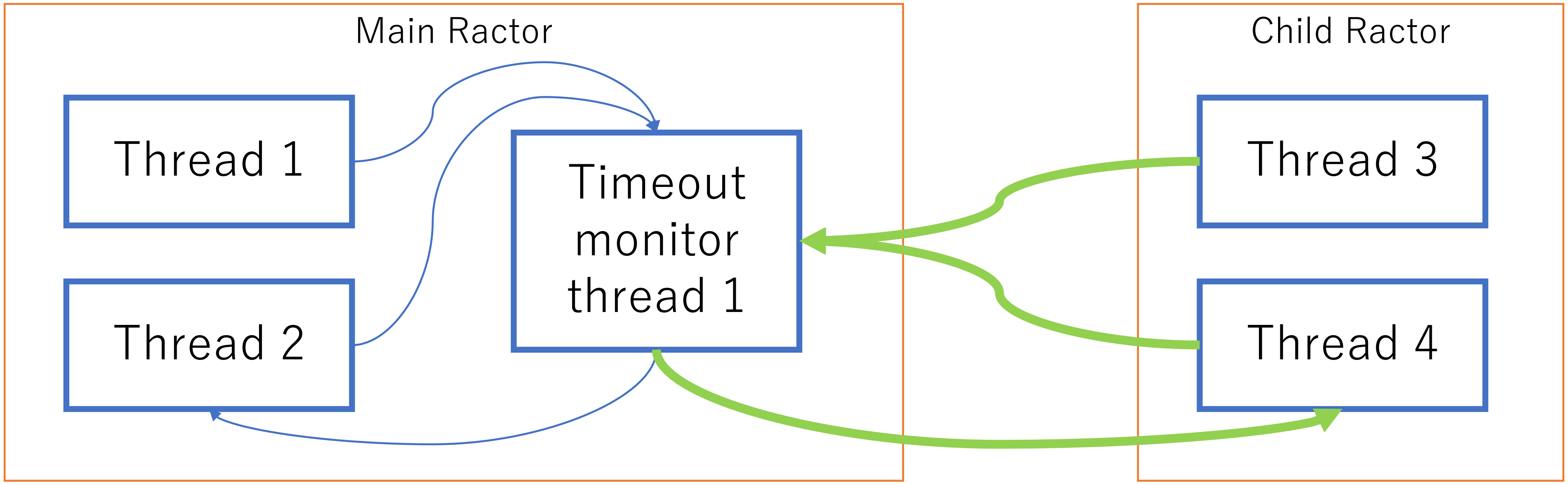
- Provide monitor threads per a Ractor
- 😀 Easy implementation with Ractor local variable (30min)
- 😢 Need monitor threads



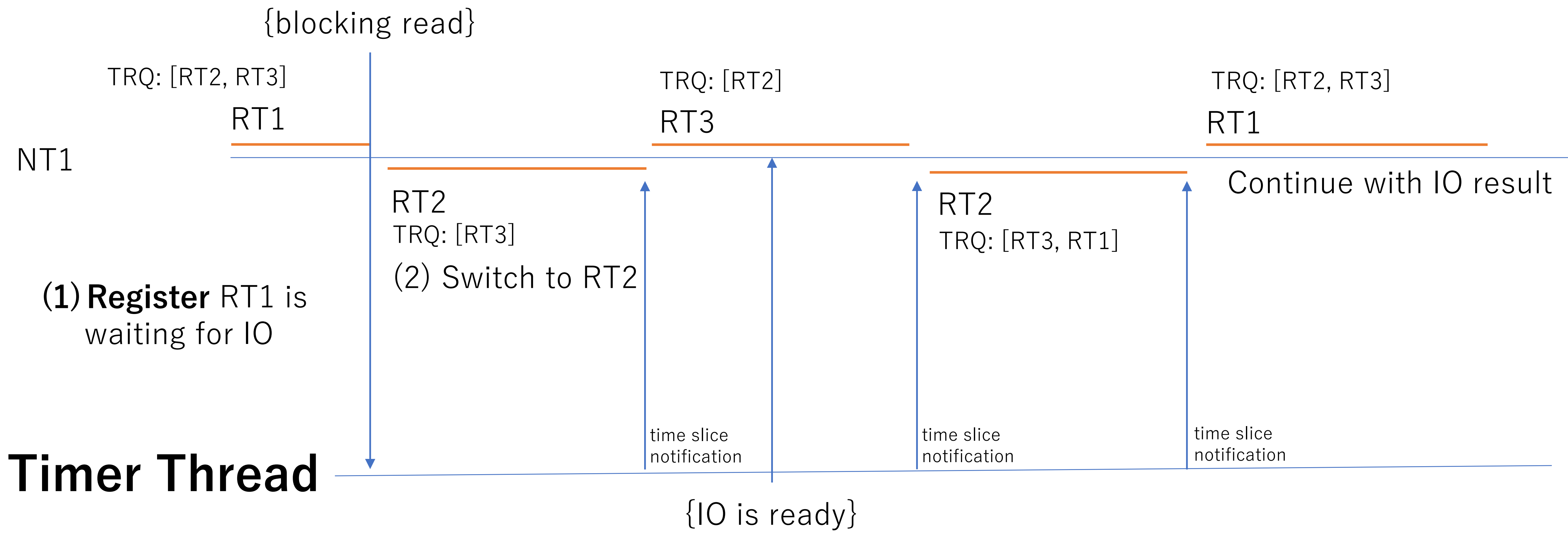


"timeout"
Solution 2: New communication path

- Use new communication path between Ractors
- 😀 1 monitor process in a Ruby (hard for massive Ractors)
- 😢 Difficult API design



Handle managed blocking operations



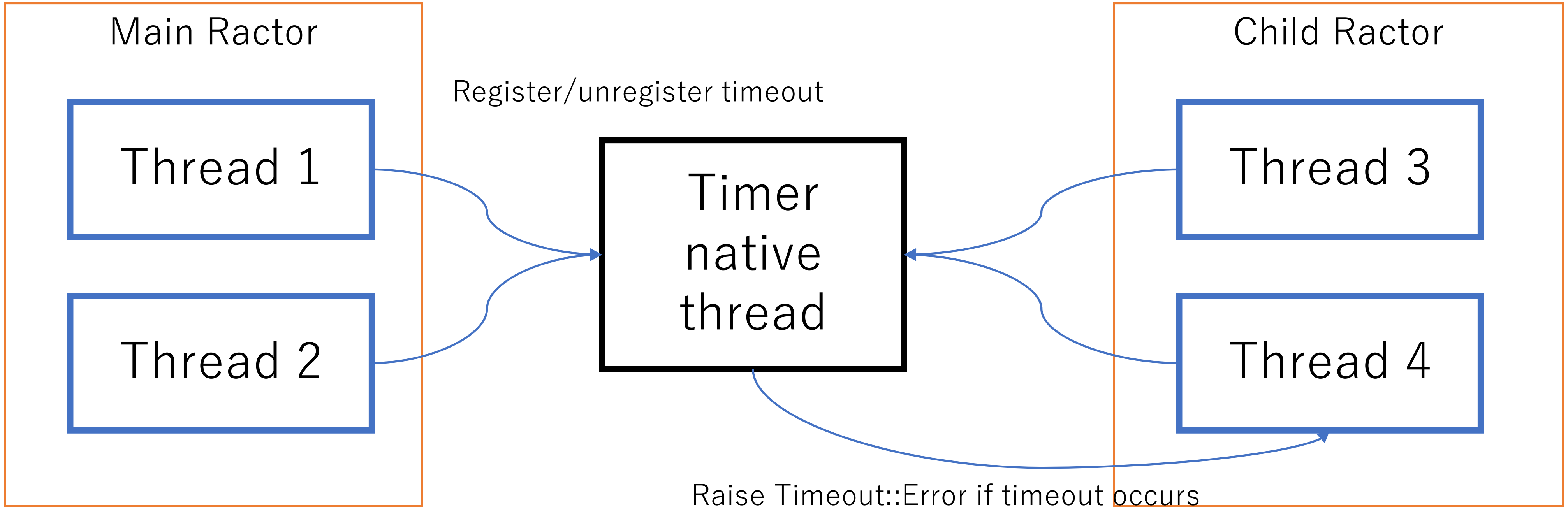
(3) Add RT1 to ready queue
→ TRQ: [RT2, RT1]

Start status:
Ruby threads RT1, RT2, RT3 are there
TRQ (Thread Ready Queue) is [RT2, RT2]

❓ RT1 can be scheduled early

"timeout"
Solution 3: Use native timer thread

- Use a timer thread for M:N thread scheduler
  - Timer thread already managing timeout for sleep, etc
- 😃 No need Ruby's timer thread / Better performance because of C impl.
- 😢 Need to support not M:N supported platforms



"timeout"
Solution 3: Use native timer thread

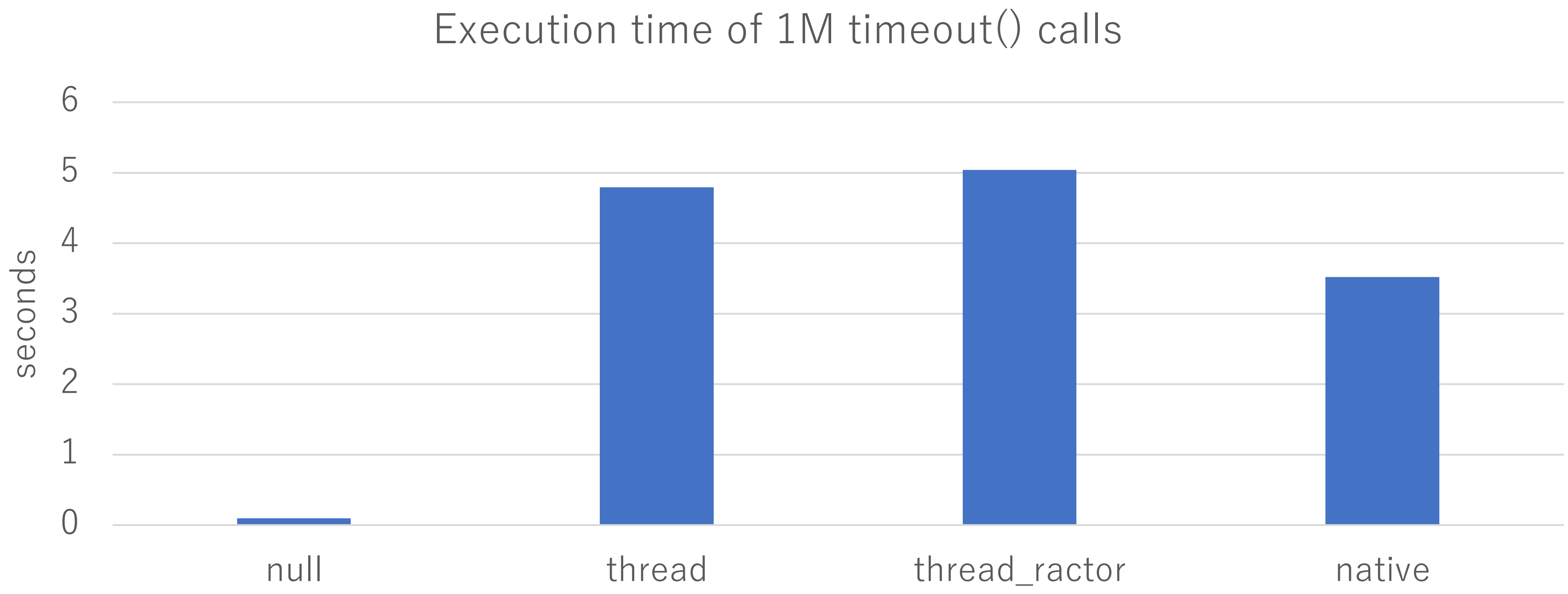
```
module Timeout
  # simplified version
  def timeout(sec, exc = Timeout::Error, msg = "…")
    RubyVM.timeout_exec(
      sec, proc{Thread.current.raise exc, msg}) do
      yield
    end
  end
```

- **`RubyVM.timeout_exec`** will call given Proc when times out by same mechanism of **`Ractor#interrupt_exec`**
- Can we introduce general API like **`Thread.timeout_exec`**?

“timeout” Benchmarking

- 99% of “timeout” call does not timeout

  → measure: `N.times{timeout(1){null_task}}`



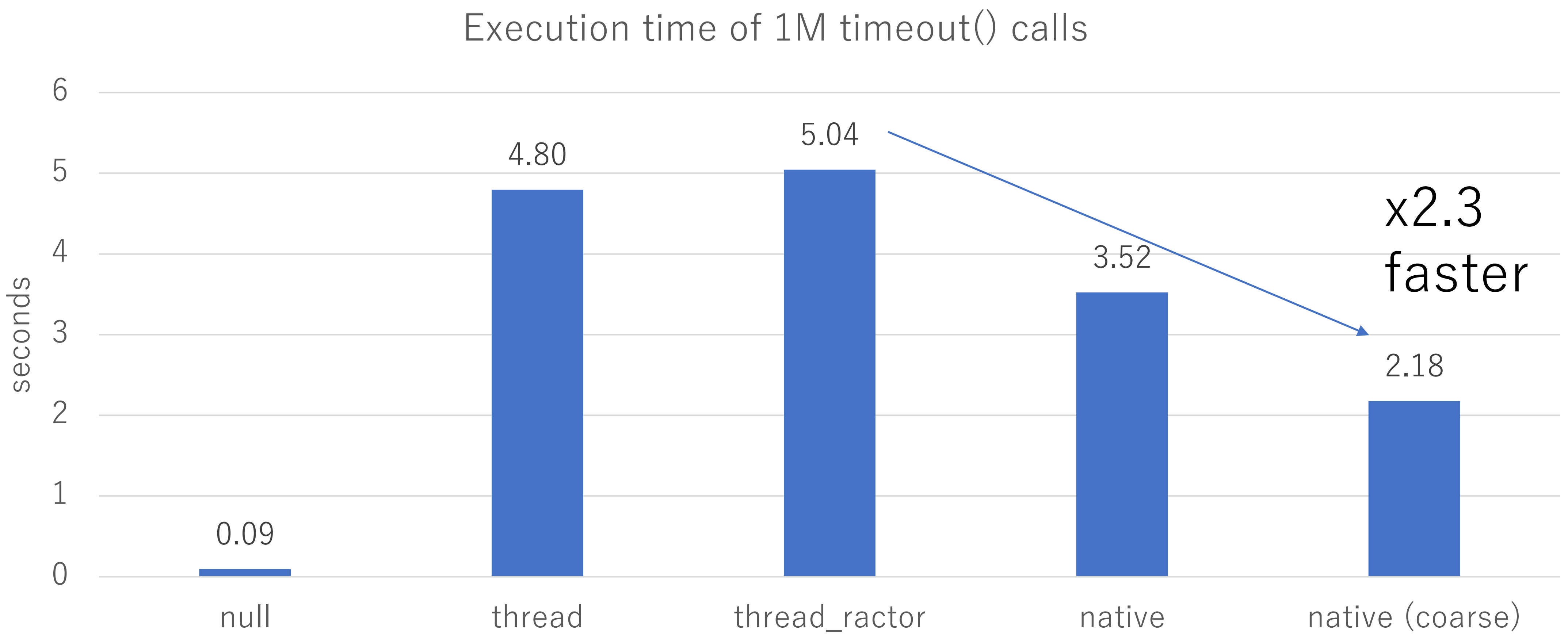
- null: no timeout (== tap)
- thread: original timeout
- thread_ractor: solution 1
- native: solution 3

- “native” is fastest, but not so fast?

“timeout” Benchmarking

- “perf” indicated that accessing hardware timer is an issue
  - To determine the sleep duration, `clock_gettime(CLOCK_MONOTONIC)` is used (1M times).
- Use `CLOCK_MONOTONIC_CORSE` (on Linux) can help
  - Faster, but not accurate (up to 4ms error on Ubuntu) and it is enough for this purpose.

“timeout” Benchmarking

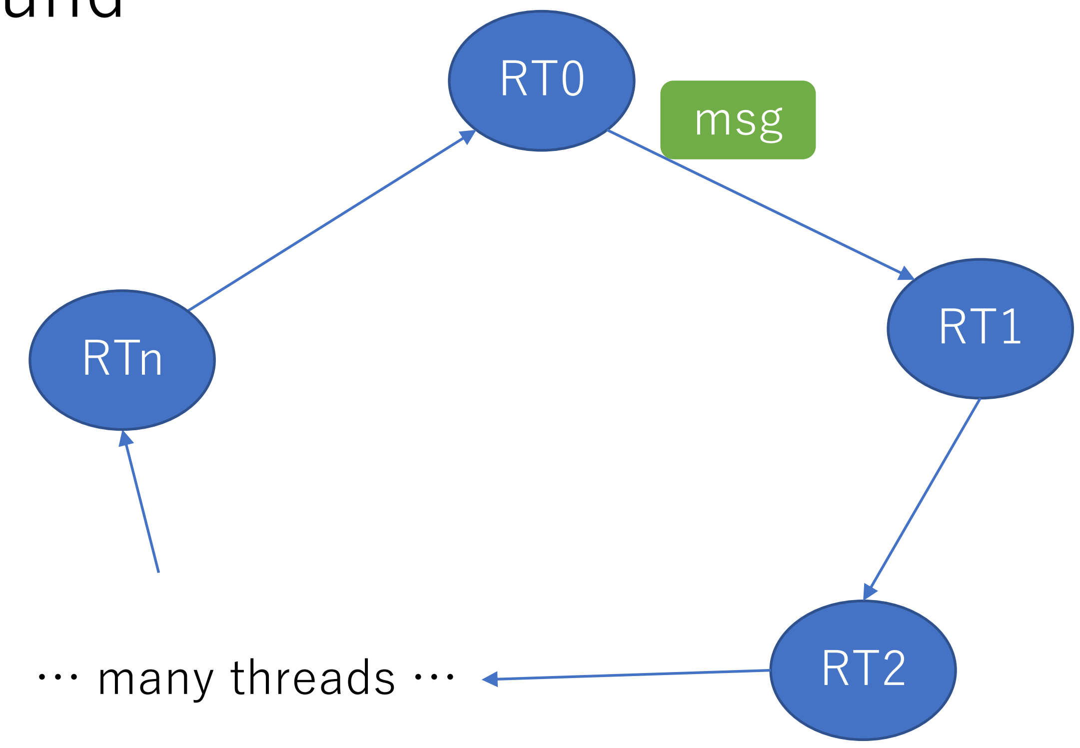


GC Performance issue

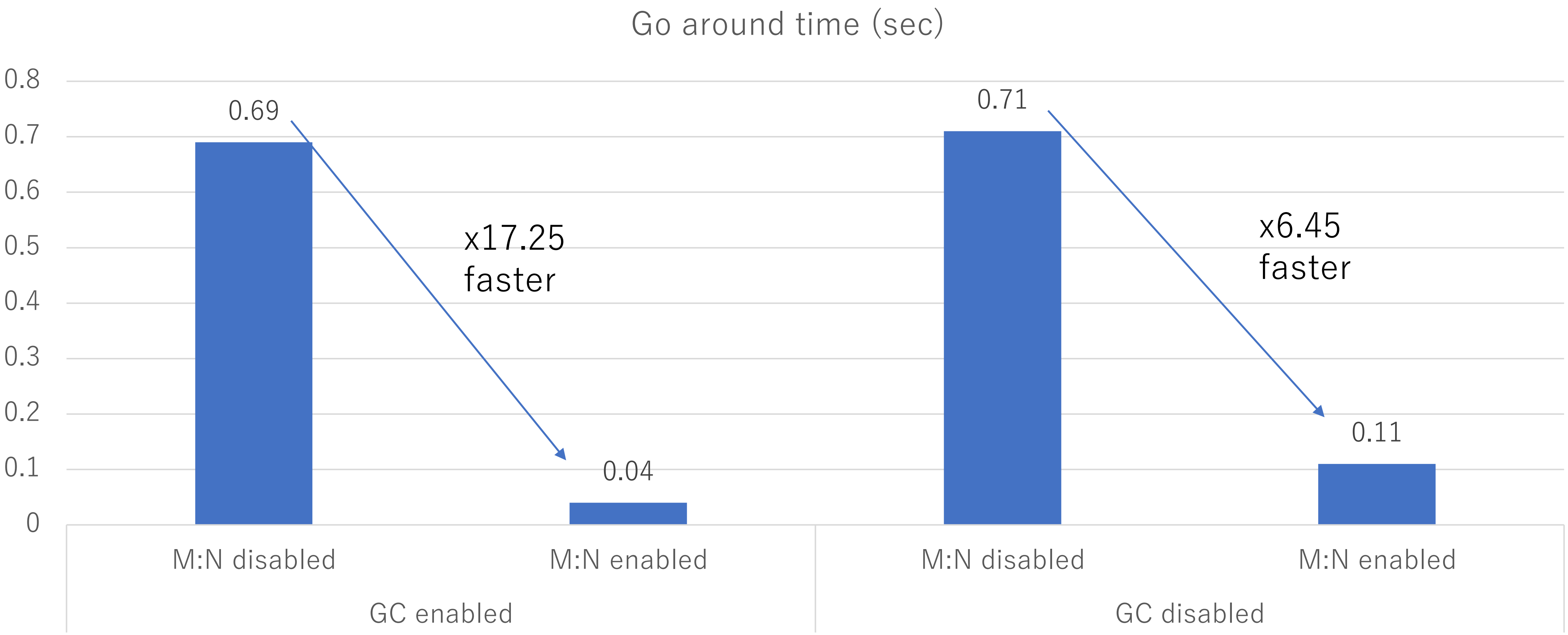
Performance survey and no proposals yet

Ring example

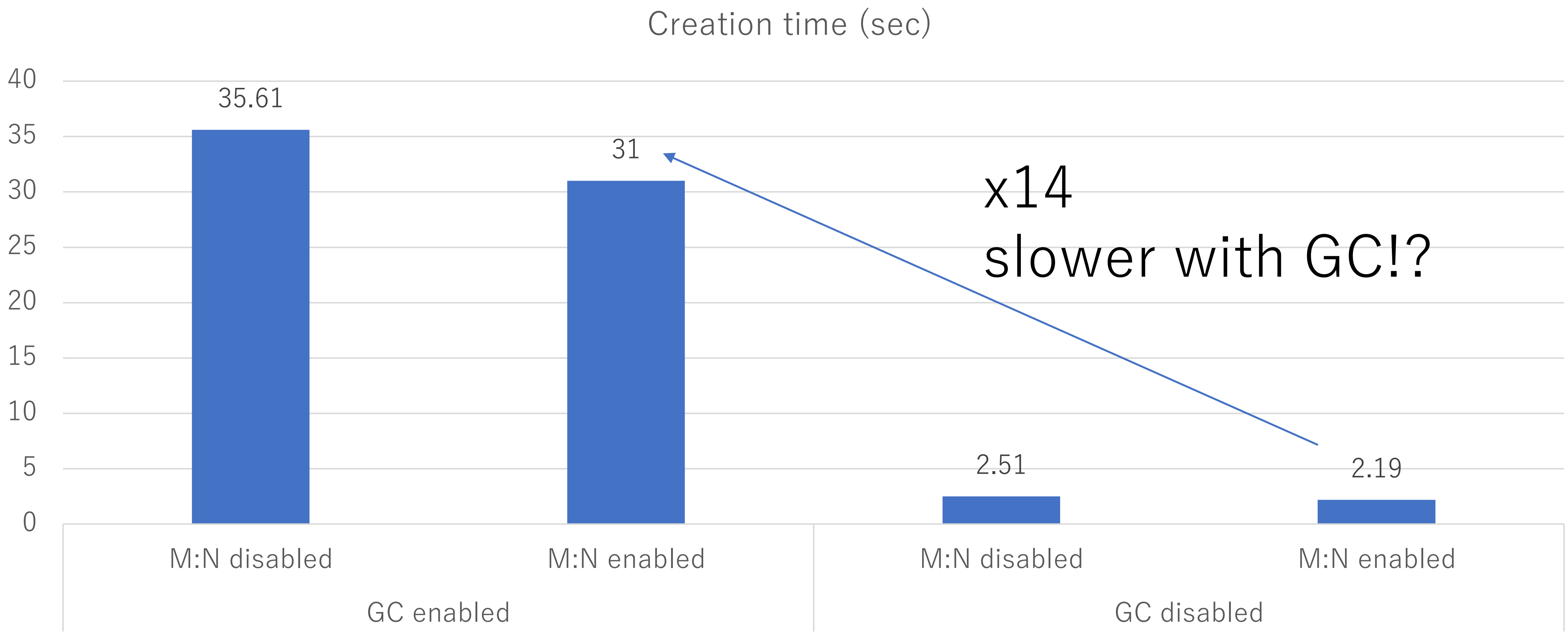
- Make 50,000 Ractors
- Send a message (object) to the next Ractor and measure the time to go around



Ring example benchmark results



Ring example benchmark results



Data from “Ruby におけるM:Nスレッドの実装”, PPL2024

GC performance issues on Ractors

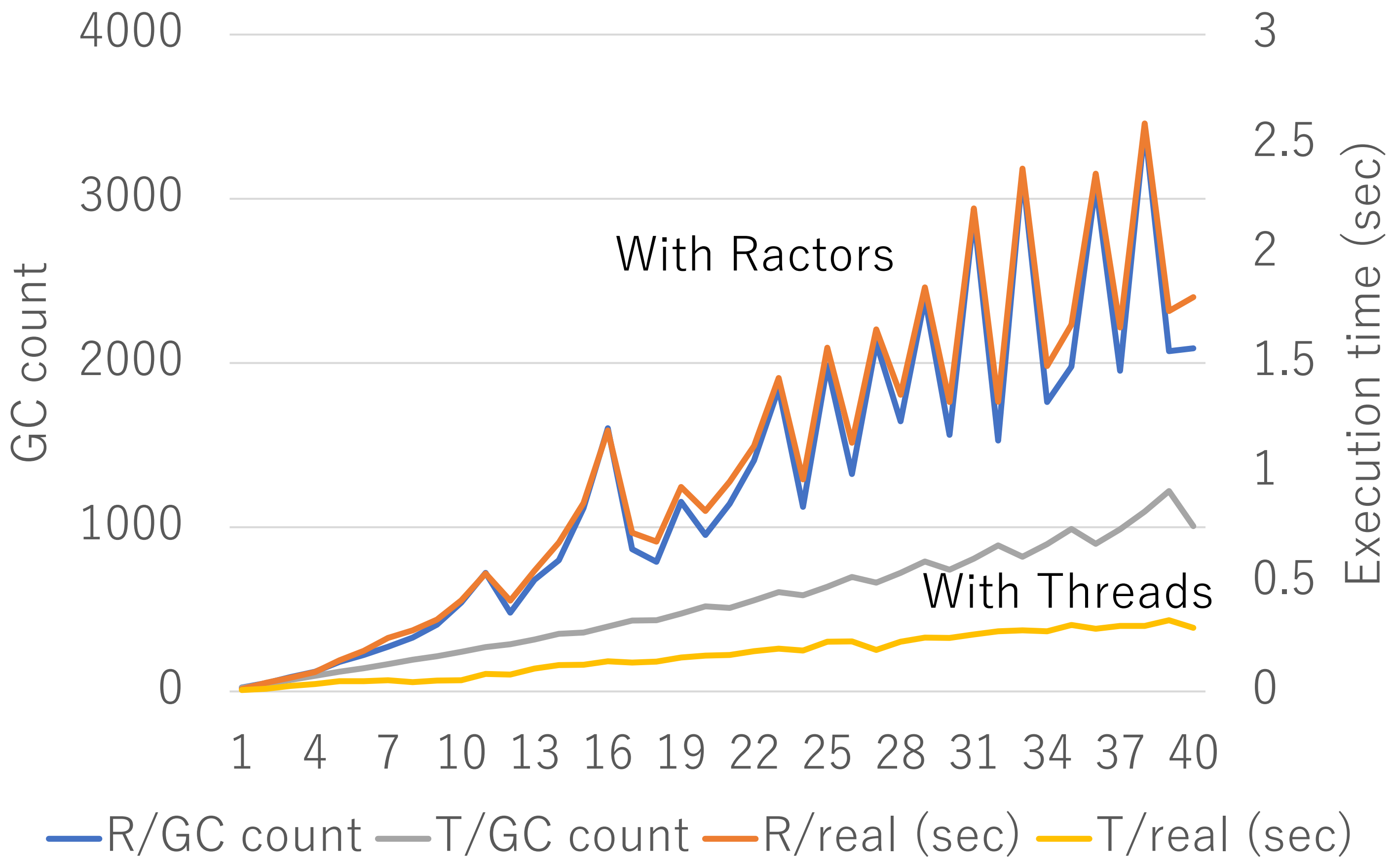
(1) To many GCs because of not enough pages
(2) Stopping active Ractors

… and more issues?

GC performance issue (1) To many GCs because of not enough pages

- Benchmark
  - Make N ractors or threads and they run the task which create 1M arrays
  - `N.times.map{ Ractor.new{ task } }`
  - GC count and execution time can be expected to be proportional to N **at worst**
    - We can expect better speed because of parallel execution

Benchmark result

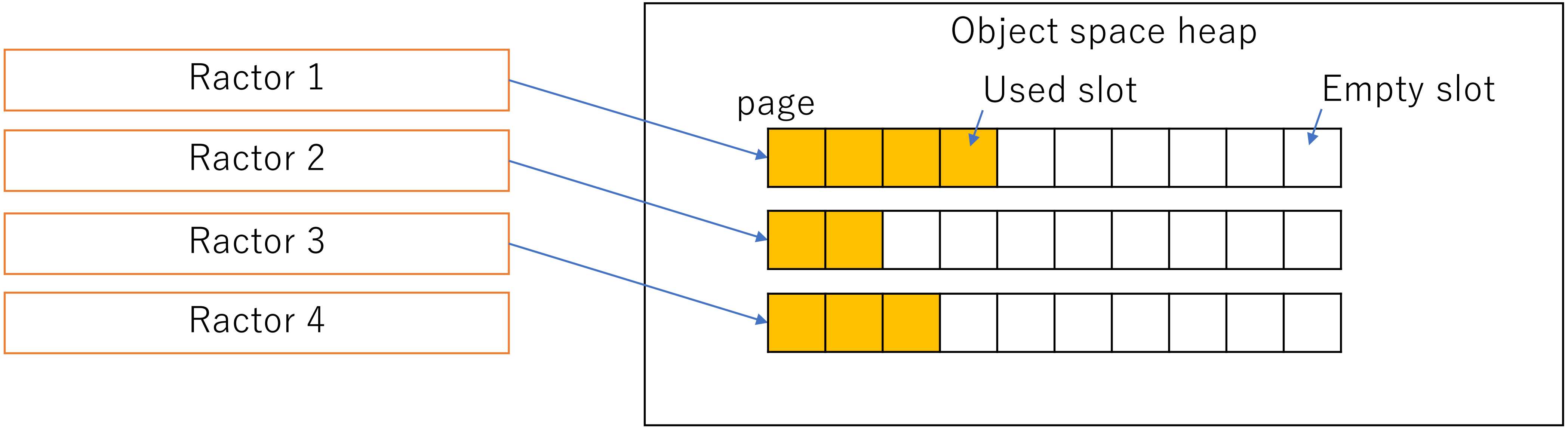


- Execution time is strongly correlated with GC count
- Ractors have clearly a larger GC count than Threads
- GC perf. on Ractors seems slower than on Threads

Object allocation on Ractors

- At the object allocation on a Ractor, the Ractor reserved a heap page
  - To remove additional synchronization per object allocation
- With 3 pages, Ractor 4 tries to reserve a page, but no page

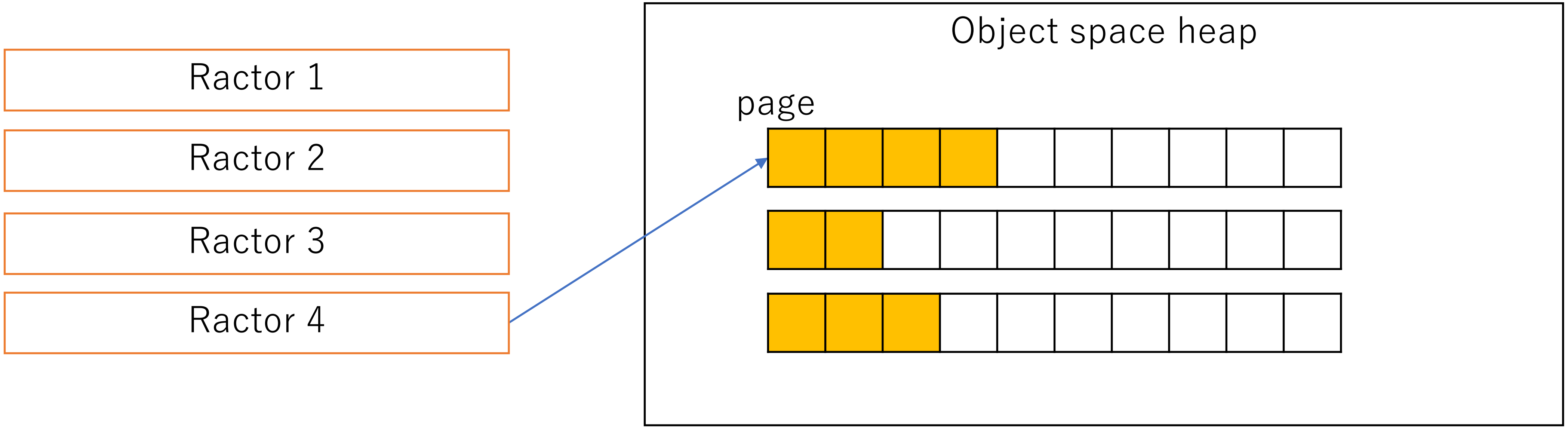
  → **Run GC!!** even if there are many unused slots



Object allocation on Ractors

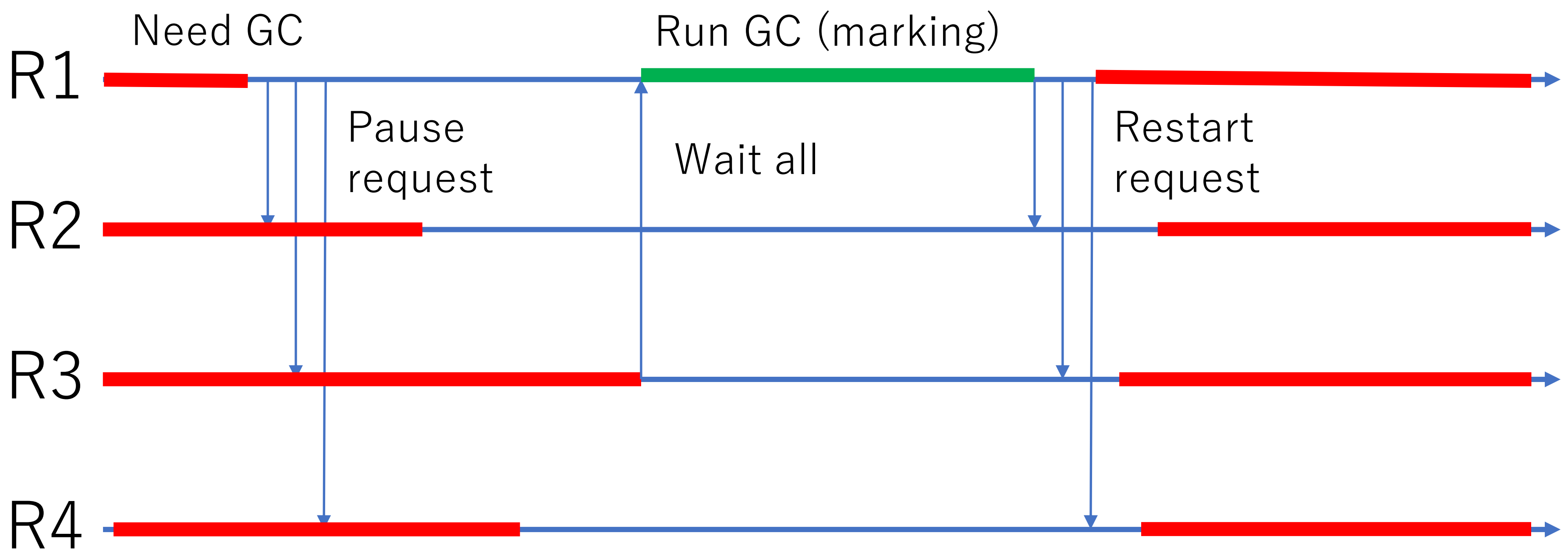
- At the object allocation on a Ractor, the Ractor reserved a heap page
  - To remove additional synchronization per object allocation
- With 3 pages, Ractor 4 tries to reserve a page, but no page

  → **Run GC!!** even if there are many unused slots



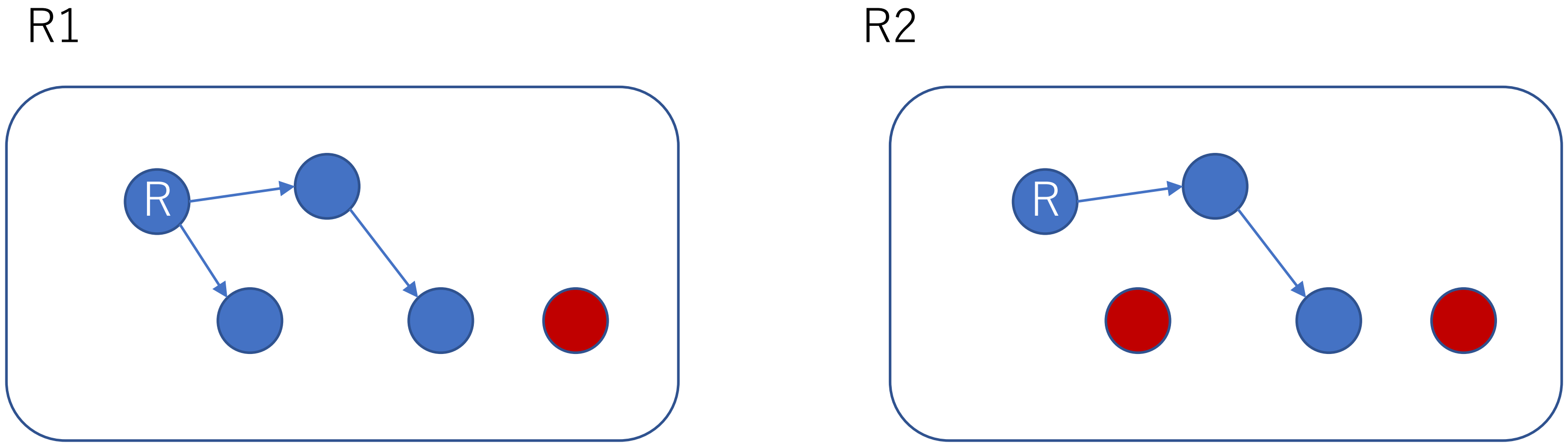
GC performance issue (2) Stopping active Ractors

- Barrier synchronization for each GC (marking) to make sure the there is no mutation while traversing the whole heap



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



Future
Proposed methods in this talk

- `Ractor#interrupt_exec (and Thread#interrupt_exec)`
- `Ractor#main?`
- `Ractor.require(feature)`
- `Ractor::Channel.new`
- `RubyVM.timeout_exec(sec, proc)`

- And more?

Today’s topics

- Support important features on Ractors
  - “require”
  - “timeout”
- Memory management issues on Ractors
- Future enhancement plans
  - GC strategy
  - Proposed APIs

STORES