“Ractor” reconsidered

or 2nd progress report of MaNy projects

Koichi Sasada
<ko1@cookpad.com>

RubyKaigi 2023

About this talk

📌 Ractor is introduced from Ruby 3.0

- “Ractor” is not used maybe because …
    - Programming model
        - Memory model (object sharing model)
        - Actor like API
    - Eco-system
    - Implementation
        - Code quality
        - Performance
- Performance improvements
    - New “Selector” API
    - Ractors on M:N Scheduler (MaNy project)
    - Ractor local GC

About Koichi Sasada

- Ruby interpreter developer employed by Cookpad Inc. (2017~) with @mame
  - YARV (Ruby 1.9~)
  - Generational/Incremental GC (Ruby 2.1~)
  - Ractor (Ruby 3.0~)
  - debug.gem (Ruby 3.1~)
  - …
- Ruby Association Director (2012~)

“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

The current status of “Ractor”

- Not used yet widely 😢
- maybe because of several **difficulties/issues** to use

Difficulties and Issues of “Ractor”

- Programming model (API)
  - Memory model (object sharing model)
  - Actor like API
- Eco-system
- Implementation issues
  - Low code quality
  - Low performance

Difficulty – Programming model
Memory model (object sharing model)

- Isolated object spaces … for **most** of objects
  - **Most of** objects: Unshareable objects are isolated
  - **A few** special objects: **Shareable objects**
    - A few special objects
      - Classes/Modules
      - Immutable objects (frozen objects which only refer to immutable objects)
      - Other special objects
- To keep this isolations, there are limitations in Ruby
  - For example, constants couldn't keep unshareable objects.
- **NOT** completely isolated (separated) object spaces like multiple processes

Difficulty – Programming model
Actor like message passing API

- Hybrid object passing API
  - Traditional **Actor style** with send/**receive** methods
  - **Rendezvous style** with **yield**/**take** methods
- Wait for multiple events by **Ractor.select**
- **Copy**/**Move** semantics to keep object isolation
  - send by reference for shareable objects
  - send by copy
  - send by move (source ractor can’t touch it)

Issue – Eco-system

- To keep object space isolation, Ractors introduces strict limitations
  - Constants can refer unshareable objects, no global variables are allowed, …
- Many existing libraries doesn’t work without modifications ≒ lack of eco-system
- Some of programs should be redesign for Ractors

Issue – Implementation
Low code quality

- CI fails every few days (about 1/10,000 trials)
  - https://dev.to/ko1/personal-efforts-to-improve-the-quality-of-ruby-interpreter-2lcl
- Difficult to implementation
  - 😃 Send/receive style is easy because we only need to lock a receiver.
  - 😥 Rendezvous style is difficult because we need to lock sender and receiver ractors = need to manage 2 locks = easy to introduce deadlock
  - 😭 Making an event mediator “Ractor.select” is difficult because we need to synchronize multiple ractors

Issue – Implementation
Low performance

- **Poor performance** because of implementation
  - It can be **even slower** than without Ractor because of additional overhead

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
```

x 3.7 faster!! 😆

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

Repeating object allocations on 4 Ractors

```
N = 10_000_000
def make = N.times{ ["", {}, []] }
require 'benchmark’
Benchmark.bm do |x|
  # sequential version
  x.report('seq'){ 4.times { make } }

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

x 2.0 slower!!! 😫

```
          user     system      total        real
seq   3.824015   0.020009   3.844024 (  3.844017)
par  17.296987   0.733804  18.030791 (  7.850200)
```

Issue – Implementation
Low performance

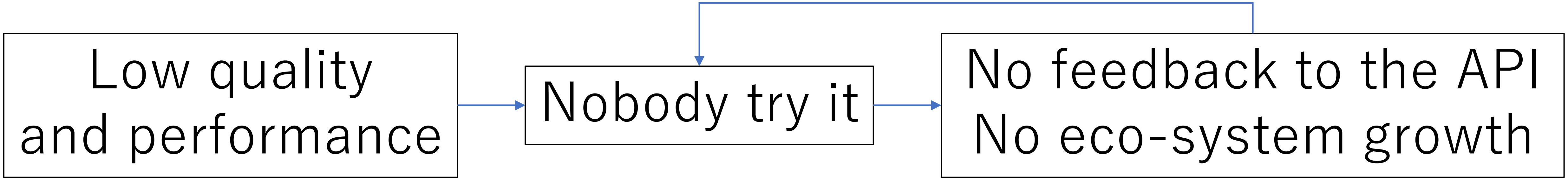
- Overhead is because …
    - Stop all ractors (barrier synchronization) on GC
        - Stop “**all**” ractors (not only “**running**” ractors) and GC for whole heap on each GC events
            - Ractors are almost isolated semantically but share same object space
        - We couldn't utilize “isolated” nature
    - Using native threads (pthreads, …) per Ractor
        - increases system calls (and consumes system resources)
        - can not make flexible ractor scheduling
    - Ractor.select(*rs) needs O(n) like “select()”
    - …

Issue – Implementation Performance

- The purpose of using Ractors is to improve application’s performance
- However, the current implementation does not meet this expectation 😥

Current situation

😥

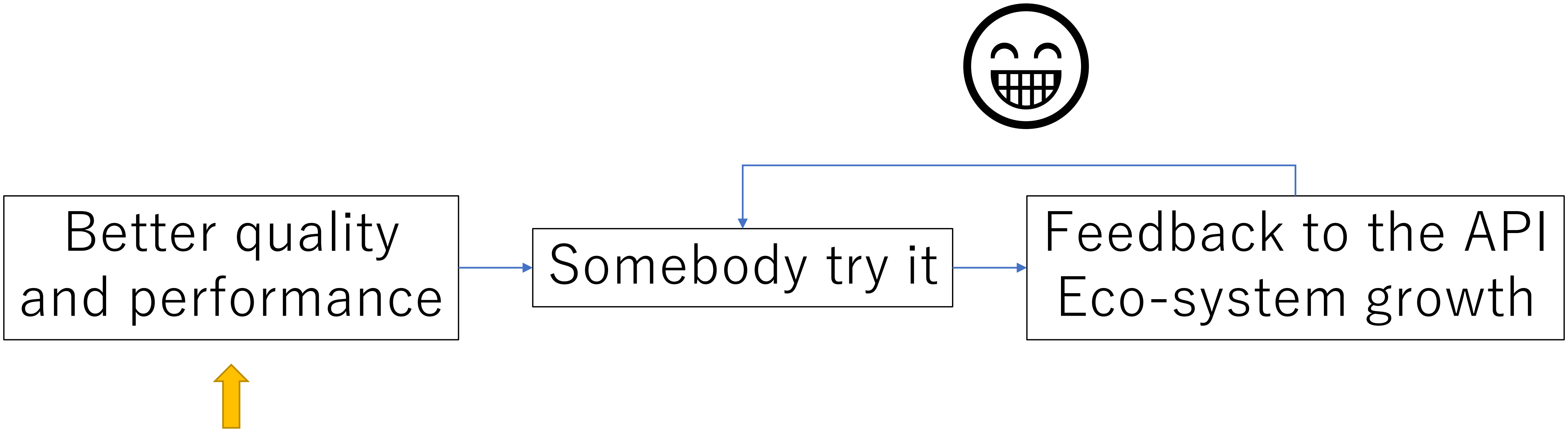


Low quality
and performance

Nobody try it

No feedback to the API
No eco-system growth

Future expected situation



Better quality and performance

Somebody try it

Feedback to the API Eco-system growth

The first area to be improved

Recent improvements

Improve code quality

- Difficulties
  - 😥 Rendezvous style is difficult
    - Needs two locks for yielding and taking ractors
  - 😭 Making an event mediator “Ractor.select” is more difficult
- We've rewritten all Ractor's synchronization code
  - Rewrite Ractor synchronization mechanism #7371
  - Redesign rendezvous protocol and mediation protocol
  - 😊 And (if I didn’t miss) we don’t have any CI failures!!

Improve performance
Ractor.select() functionality

- 😥 Ractor.select needs O(n)
- Introduce “Ractor::Selector” API
  - Rewrite Ractor synchronization mechanism #7371
  - Pre-registration API (register at first)
  - 🙂 The waiting cost can become **O(1)**
    - but O(n) on current implementation 😅
  - (not accepted by Matz though😅)

Ractor::Selector

(not accepted by Matz though😅)

```
n.times do
  # wait and it takes
  # O(n) each time
  Ractor.select(
    r1,
    r2,
    r3, …)
end
```

```
# prepare
s = Ractor::Selector.new(
      r1, r2)
s.add(r3)
…
# wait
n.times do
  # O(1) (in theory)
  s.wait
end
```

Order is important to wait for massive number of ractors

Performance improvement
MaNy project

- 😥 Poor performance because of depending on native threads
- 😄 Introduce own M:N scheduler

  → Ractor on MaNy project

  - MaNy project: Making *MaNy* threads on Ruby (RubyKaigi 2022)
    - Last year I only introduced about M:N scheduler with Ruby’s threads, and now Ractor is also supported

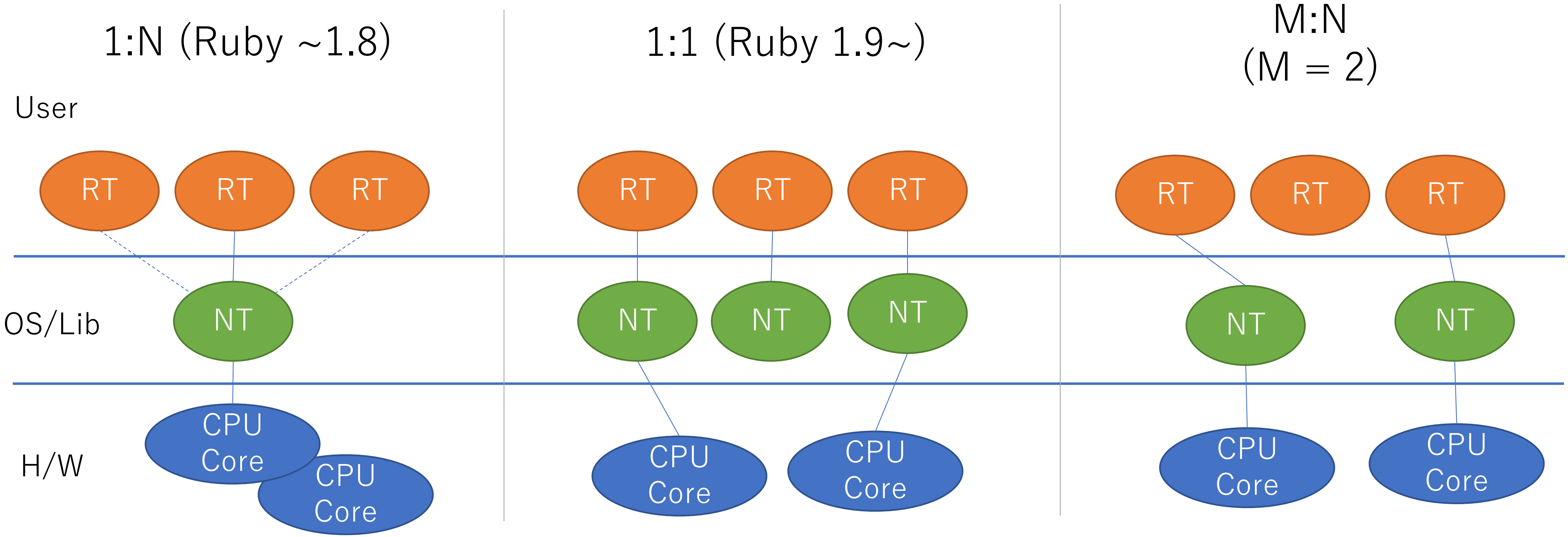
Quoted from RubyKaigi 2022 slides (modified)

MaNy project
Thread system implementation techniques

Study in computer science/OS area

NT: Native thread or kernel thread

How to handle N=3 Ruby threads/ractors (RTs) on 2 CPU cores?

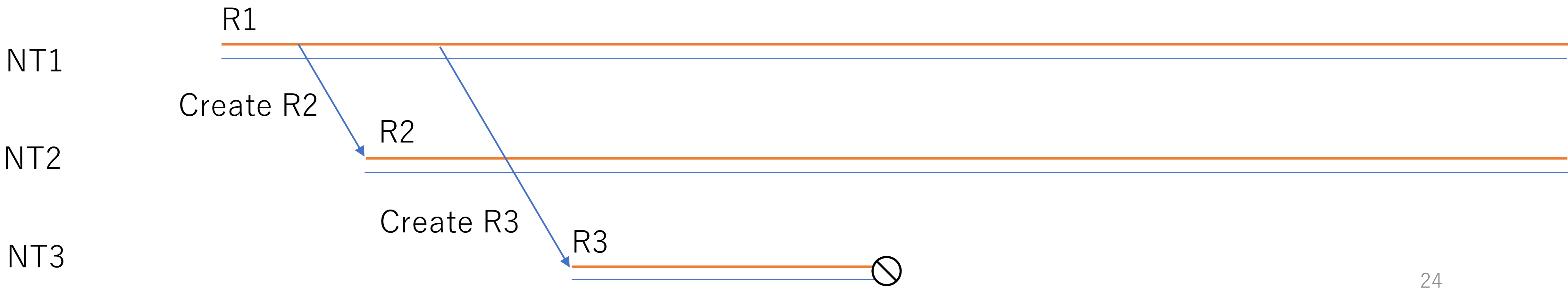


Quoted from RubyKaigi 2022 slides (modified)

1:1 model

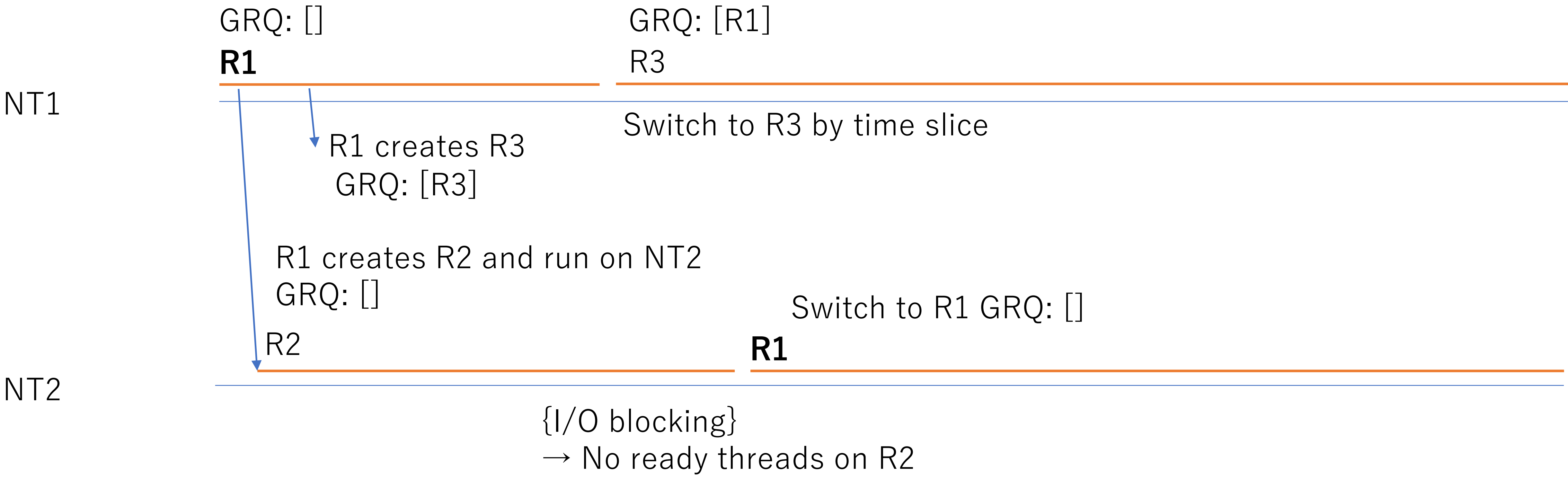
Most simplified technique

- 1 native thread (NT) per Ruby a thread / ractor
  - Ruby 1.9~ (has GVL limitation. This page eliminates it)
  - 😃 Simple, easy to handle blocking operations (system does)
  - 😃 Can run in parallel on multi-core systems
  - 😨 More overhead (compare with 1:N, in theory)
  - 😨 Less controllable (only native thread system schedules)



Quoted from RubyKaigi 2022 slides

M:N Ractor level scheduling (M=2)



GRQ: []

R1

GRQ: [R1]

R3

NT1

Switch to R3 by time slice

R1 creates R3

GRQ: [R3]

R1 creates R2 and run on NT2

GRQ: []

Switch to R1 GRQ: []

R2

R1

NT2

{I/O blocking}

→ No ready threads on R2

📌 Ractor R1, R2, R3 have 1 thread, respectively.

💪 R1 runs on NT1 and NT2 (M:N scheduler)

M:N scheduler
Technical topics

- Design our own scheduler two level scheduler
    - Thread level scheduler and Ractor level scheduler
    - Rebirth timer thread to manage “waiting”
        - Redefine I/O waiting and canceling protocol
        - Redefine sleeping protocol
    - Redefine signal delivering protocol
    - Dynamic native threads numbers
    - Supports dedicated (1:1) native threads for compatibility for C-extensions
    - Robust canceling code on parallel execution
    - Introduce a lazy queuing scheduling technique for performance
    - Rewrite ractor synchronization code with the scheduler
    - Rewrite barrier implementation for ractors with the scheduler
    - Issue from thread-local storage
        - https://twitter.com/_ko1/status/1650385648006873088
- Current code is here: https://github.com/ko1/ruby/tree/many2
- Complete **almost tests** in ruby/ruby

Evaluation
Ractor creation/joining on M:N scheduler

	Time (sec) on GC.enable	Time (sec) on GC.disable
Threads (master)	0.22	0.21
Threads (MaNy)	0.08	0.06
Ractors (master)	4.88	0.76
Ractors (MaNy)	2.35	0.55
Ractors (MaNy, MAX_PROC=1)	1.09	0.41

x 2.6

x 2.1

x 4.5 😀

x 13.6 ?

- Creating 10,000 threads or ractors and wait all of terminations
- MAX_PROC: Maximum native thread number (default: 8)
- Machine and VM stack is limited to minimum size
- https://gist.github.com/ko1/b9222243ed246d782ab259252da15ad1

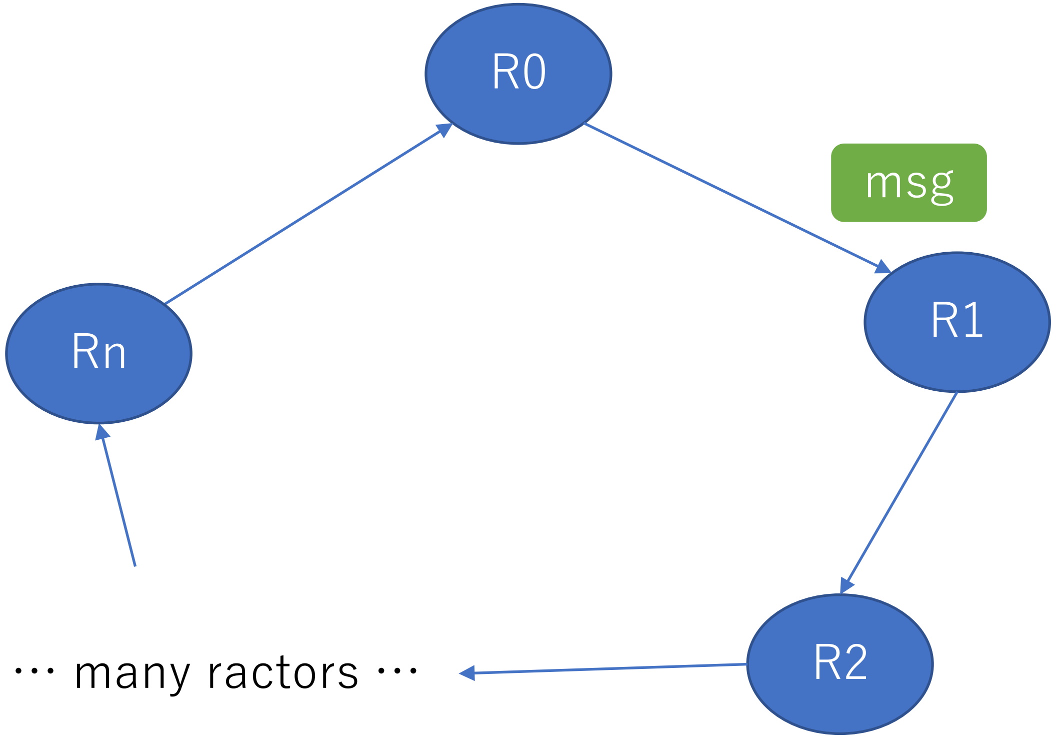
Environment:
AMD Ryzen 9 5900HX (8 cores, 16 H/W threads)
Ubuntu 22.04
gcc version 11.3.0
ruby 3.3.0dev (2023-04-28T11:29:02Z master 7ba37cb7aa)

Should be same in theory

Quoted from RubyKaigi 2022 slides (modified)

Evaluation
Ring example on M:N scheduler

- Prepare **n** Ractors (/threads) ordered sequentially
- Pass a message to the next Ractor (/thread)



Evaluation
Ring example

	Time (sec)
Threads (master)	969.55
Threads (MaNy)	9.20
Ractors (master)	166.52
Ractors (MaNy)	14.22
Ractors (MaNy, MAX_PROC=1)	7.38

x 105.4

x 11.7

x 22.6

- Making 1 ring by **10,000** threads/ractors and **1,000** times message passings = **10M** passings
- Time of making threads/ractors is excluded.
- Benchmark code: https://gist.github.com/ko1/ac325a785ae292540bd99f141ad55383

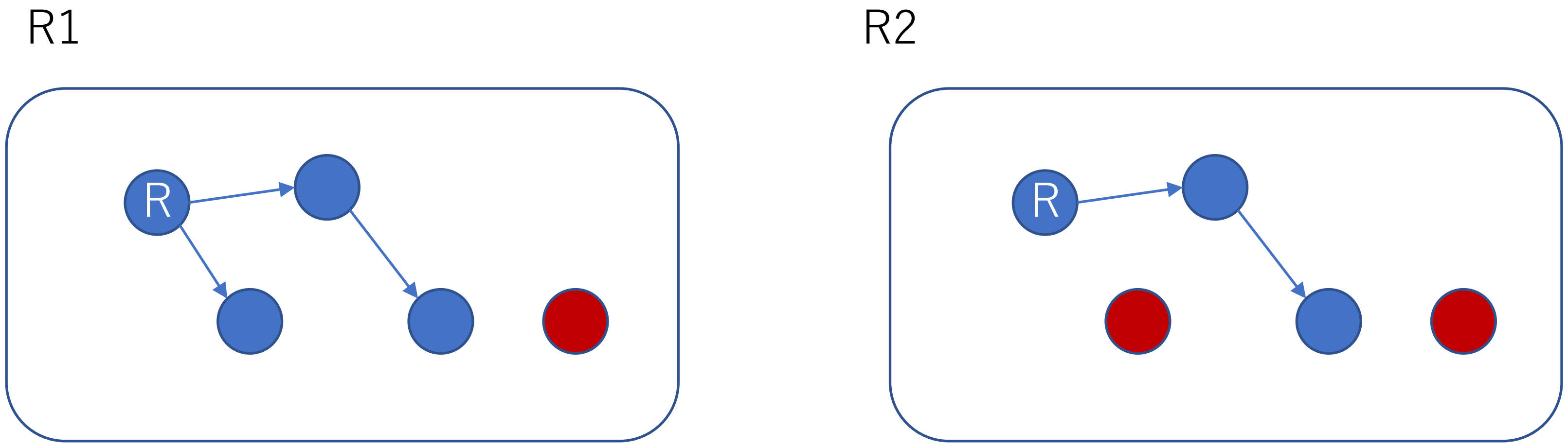
Future work

Further performance improvement
Ractor local GC

- Ractor’s object space is almost separated with other ractors’ object space

→ Run GC separately

  - Do not need to stop all ractors
  - Run GC in parallel

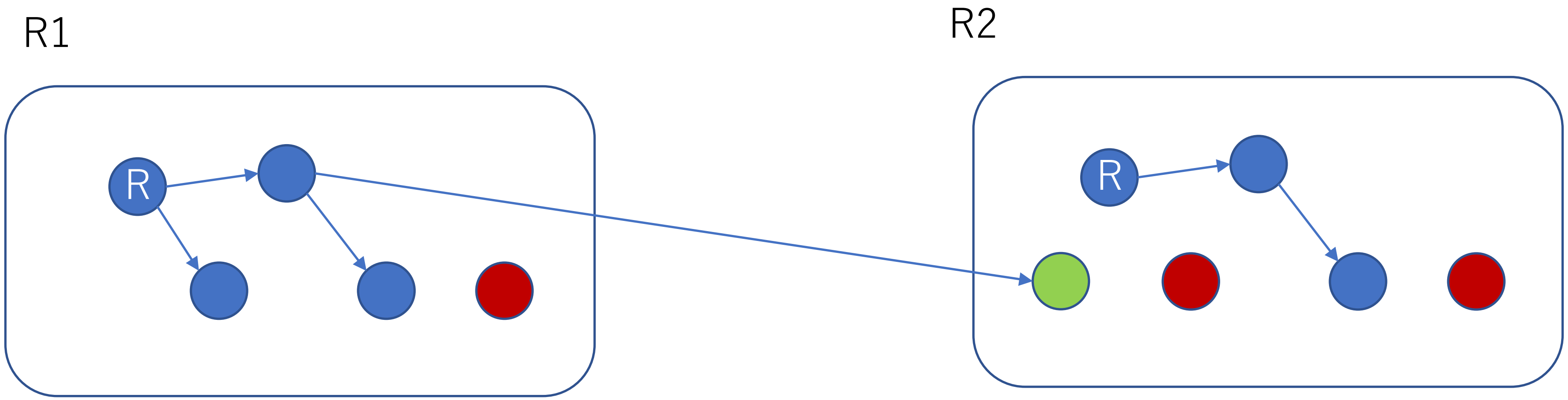


Further performance improvement
Ractor local GC

- Problem is “There are several shared shareable objects” between ractors

→ Distributed GC (with a few whole GC)

Ractor local GC is ongoing project with GSoC 2022 contributor Rohit Menon



About this talk

- “Ractor” is not used maybe because …
    - Programming model
        - Memory model (object sharing model)
        - Actor like API
    - Eco-system
    - Implementation
        - Code quality
        - Performance
- Performance improvements
    - New “Selector” API
    - Ractors on M:N Scheduler (MaNy project)
    - Ractor local GC