RUBYCONF 2020

Ractor Demonstration

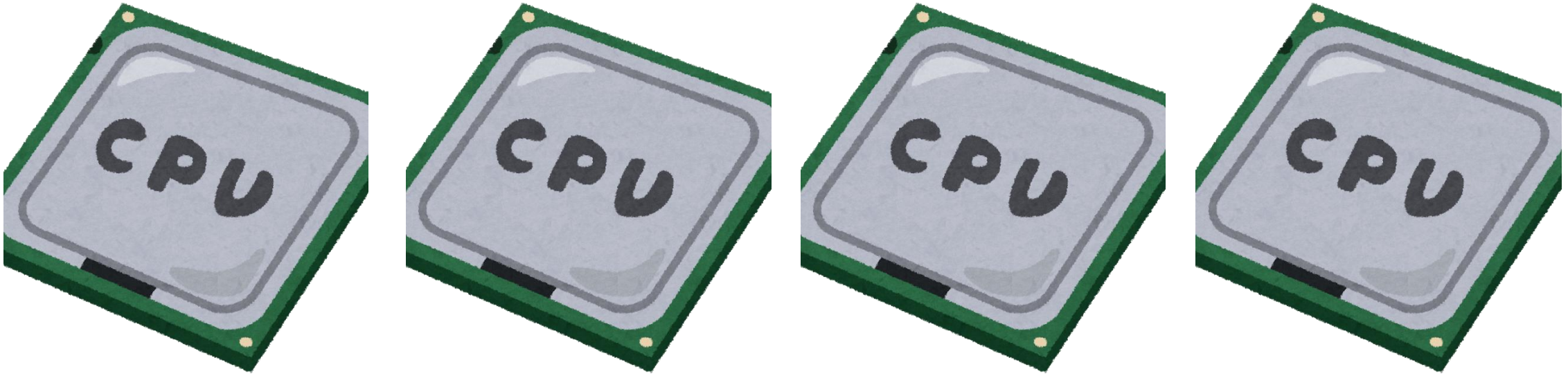
Koichi Sasada

(from Japan)

Cookpad Inc.

Ask Koichi

- I will check tweets with “#ractor” hashtag on twitter so if you have a question, please tell me.
- You can access this presentation slies on: https://www.atdot.net/~ko1/activities/



Background

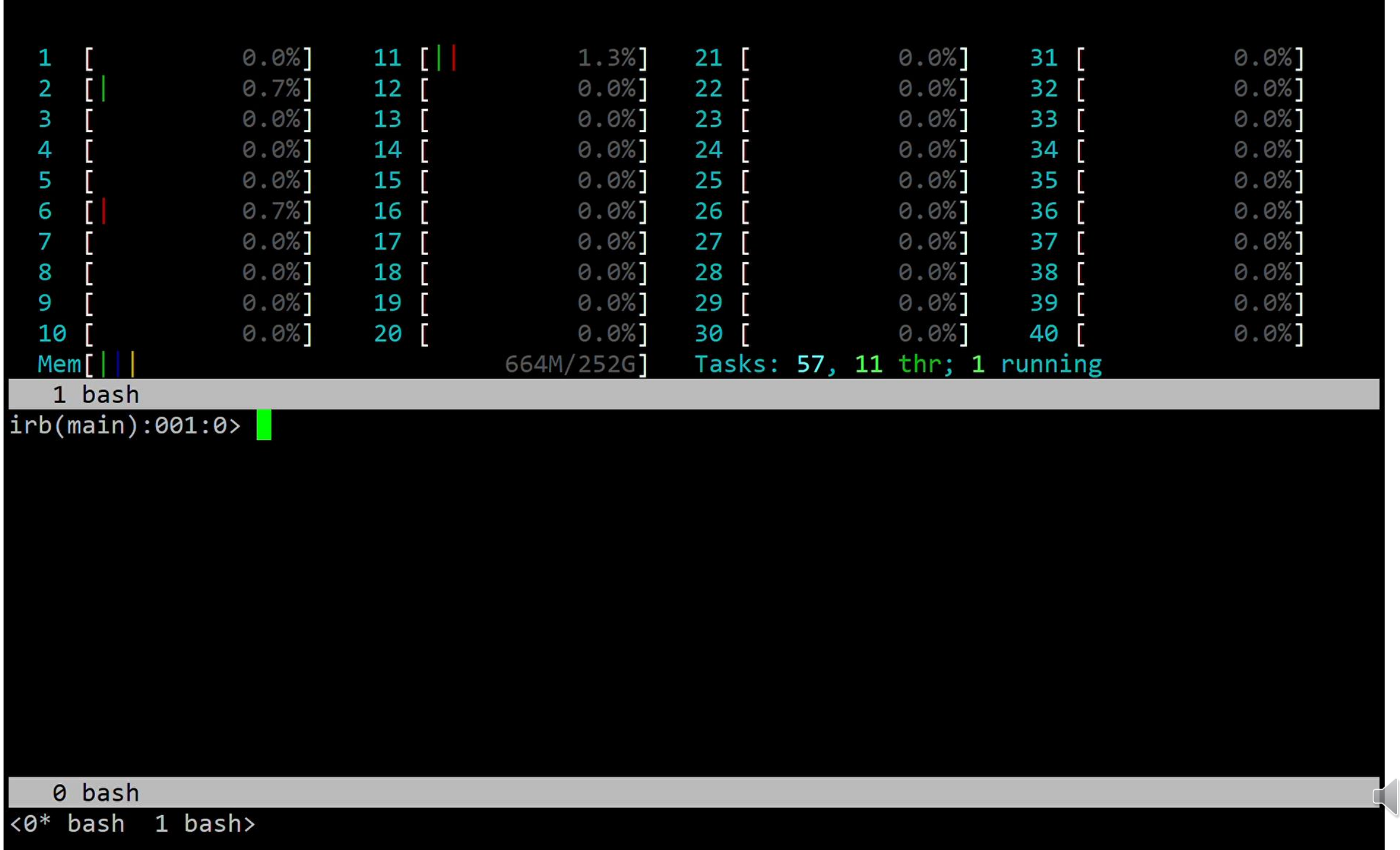
Parallel programming

- Parallel execution on Multi-core CPUs is important
- Multi-process programming is not easy
  - Hard to communicate
  - Hard to control resource consumption
- **Multi-thread doesn’t support parallel execution on MRI**

Demonstration
MRI can not utilize CPUs with threads

```
# make 20 threads running busy loop
(1..20).each do
  # each threads run busy loop
  Thread.new{ loop{} }
end
```

Intel(R) Xeon(R) CPU E5-2630 v4
10 cores x 2 HT x2 CPUs = 40 logical CPUs


1 [0.0%] 11 [|| 1.3%] 21 [0.0%] 31 [0.0%]
2 [| 0.7%] 12 [0.0%] 22 [0.0%] 32 [0.0%]
3 [0.0%] 13 [0.0%] 23 [0.0%] 33 [0.0%]
4 [0.0%] 14 [0.0%] 24 [0.0%] 34 [0.0%]
5 [0.0%] 15 [0.0%] 25 [0.0%] 35 [0.0%]
6 [| 0.7%] 16 [0.0%] 26 [0.0%] 36 [0.0%]
7 [0.0%] 17 [0.0%] 27 [0.0%] 37 [0.0%]
8 [0.0%] 18 [0.0%] 28 [0.0%] 38 [0.0%]
9 [0.0%] 19 [0.0%] 29 [0.0%] 39 [0.0%]
10 [0.0%] 20 [0.0%] 30 [0.0%] 40 [0.0%]
Mem[|| 664M/252G] Tasks: 57, 11 thr; 1 running
1 bash
irb(main):001:0>
0 bash
<0* bash 1 bash>

Background
Concurrent **Thread** programming is hard

- Appropriate synchronization is needed
  - Threads can share everything
  - Critical bugs w/o synchronization
    - Data race / race condition
    - Dead/live locking
- Difficult debugging on **non-deterministic** nature
- Difficult to tune the performance on fine-grained synchronizations

Demonstration
Threads require synchronization

```
# two threads increment numbers
counter = 0
get = proc{ counter }
(1..2).map do |i|
  Thread.new do
    1_000_000.times{ counter = get.call + 1 }
  end
end.each{|t| t.join}
p counter #=> 2_000_000 is expected,
          # but 1709078, 1712839, …
```

Demonstration
Threads require synchronization

```
# two threads increment numbers
counter = 0; m = Mutex.new
get = proc{ counter }
(1..2).map do |i|
  Thread.new do
    1_000_000.times{ m.synchronize{ counter = get.call + 1 }}
  end
end.each{|t| t.join}
p counter #=> 2_000_000
```

Goal:
Easy and **Parallel** concurrent programming on Ruby

Our proposal:

Ractor

an Actor-like concurrent abstraction

Limited object sharing between ractors

with inter-ractor communication

“Guild” → “Ractor”

- Basic concept was proposed with “Guild” code name at RubyKaigi 2016 and 2018
  - http://rubykaigi.org/2016/presentations/ko1.html
  - https://rubykaigi.org/2018/presentations/ko1.html
- With Matz, we discussed the name of Guild and decided to change the class name from **Guild** to **Ractor** (Ruby’s Actor-like).

Ractor Concepts

- Run multiple ractors in parallel
- Limited object sharing
- Building ractors network with push/pull types communication
- Sending objects with copy/move

- Ractor's specificaiton: https://github.com/ko1/ruby/blob/ractor_parallel/doc/ractor.md

Ractors concept
Run multiple ractors in parallel

- **Multi-Ractors in one process**
- **`Ractor.new{ expr }`** makes a new Ractor
- Ractor and thread
  - A process has at least 1 ractor
  - A ractor has at least 1 thread
  - Threads in a Ractor can not run in parallel (~2.7 compatible)

Demonstration
Making a ractor

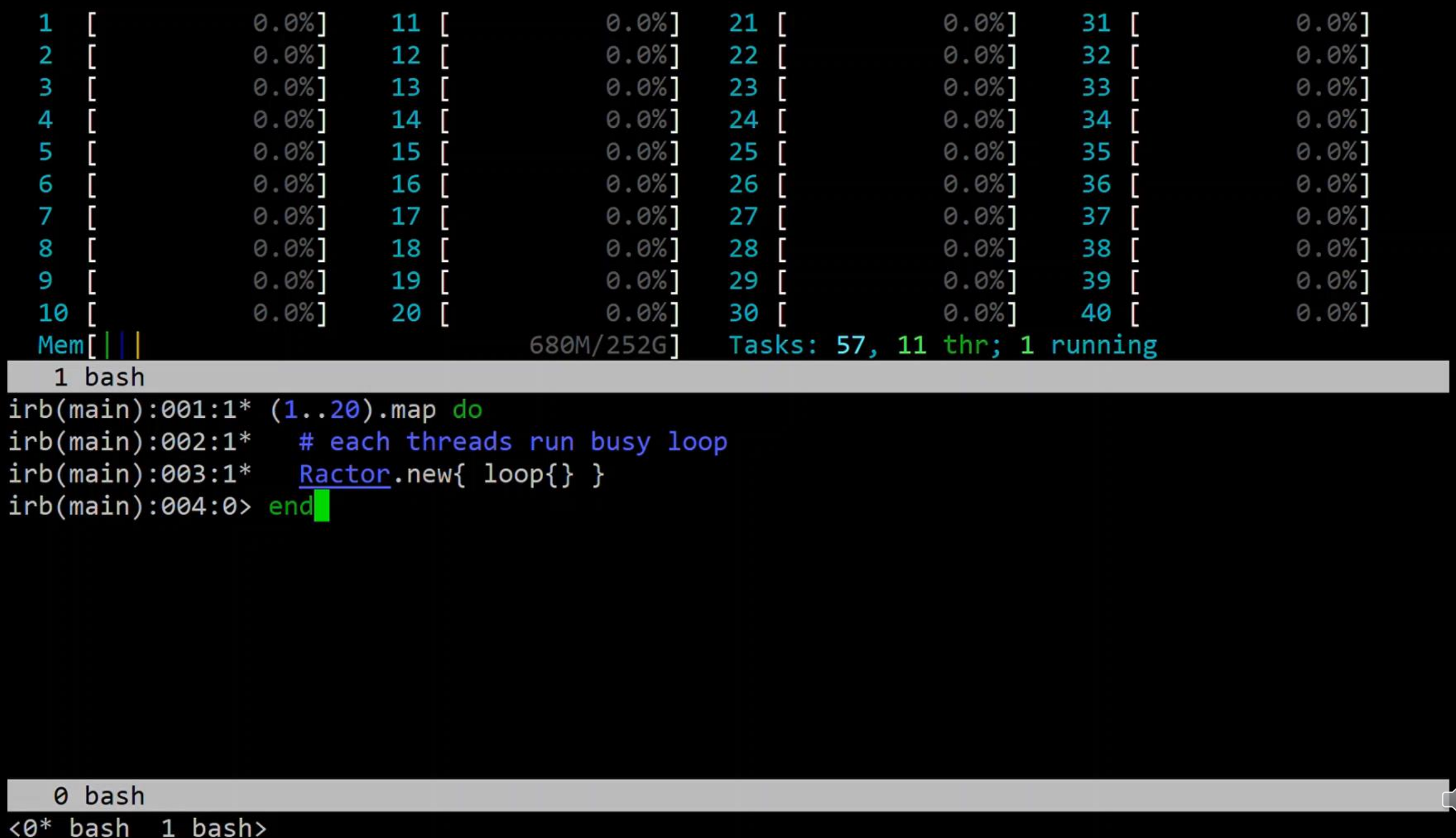
```
r = Ractor.new do
  p self #=> #<Ractor:#2 t.rb:1 running>
end
```

```
#=> warning: Ractor is experimental, and the behavior may change in future
versions of Ruby! Also there are many implementation issues.
```

#=> **Ractor is an experimental feature on Ruby 3.0**
Specifications can be changed with your voice!

Demonstration
Multiple Ractors run simultaneously

```
# make 20 ractors running busy loop
(1..20).map do
  # each threads run busy loop
  Ractor.new{ loop{} }
end
```


1 [0.0%] 11 [0.0%] 21 [0.0%] 31 [0.0%]
2 [0.0%] 12 [0.0%] 22 [0.0%] 32 [0.0%]
3 [0.0%] 13 [0.0%] 23 [0.0%] 33 [0.0%]
4 [0.0%] 14 [0.0%] 24 [0.0%] 34 [0.0%]
5 [0.0%] 15 [0.0%] 25 [0.0%] 35 [0.0%]
6 [0.0%] 16 [0.0%] 26 [0.0%] 36 [0.0%]
7 [0.0%] 17 [0.0%] 27 [0.0%] 37 [0.0%]
8 [0.0%] 18 [0.0%] 28 [0.0%] 38 [0.0%]
9 [0.0%] 19 [0.0%] 29 [0.0%] 39 [0.0%]
10 [0.0%] 20 [0.0%] 30 [0.0%] 40 [0.0%]
Mem[||| 680M/252G] Tasks: 57, 11 thr; 1 running
1 bash
irb(main):001:1* (1..20).map do
irb(main):002:1* # each threads run busy loop
irb(main):003:1* Ractor.new{ loop{} }
irb(main):004:0> end
0 bash
<0* bash 1 bash>

CPU

Intel(R) Core(TM) i7-10810U CPU @ 1.10GHz

60 秒間の使用率 (%)

100%

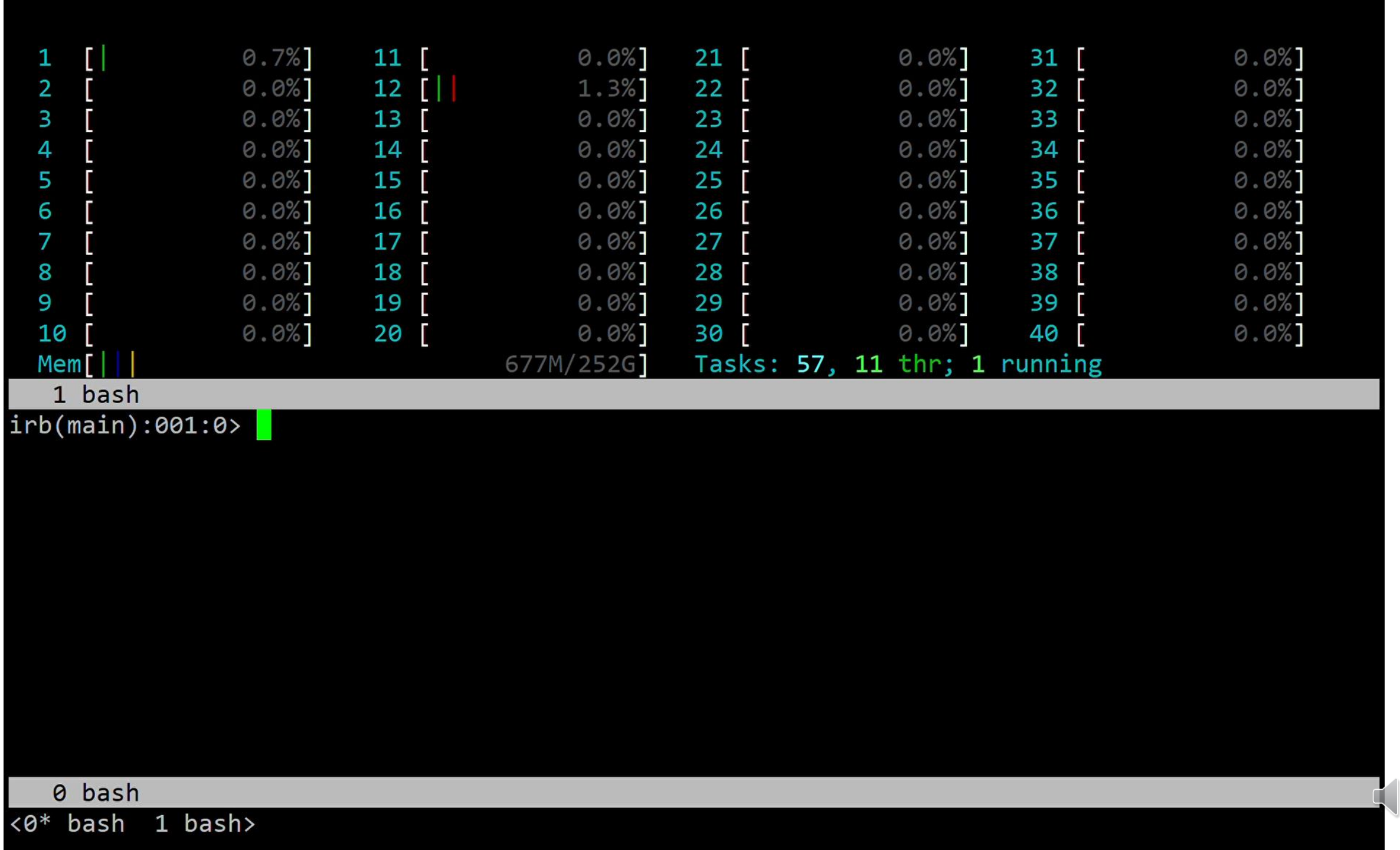
Demonstration
Ractor creation and waiting for the result

```
require "prime"

r = Ractor.new( 2**61 - 1 ) do |i|
  i.prime?
end

p r.take # You can get the result
#=> true
```

Demonstration
Heavy numeric calculation

```
# Sequential thread
(t1 = Time.now
 n1 = 2**61 - 1
 n2 = 2**61 + 15
 [n1.prime?, n2.prime?])
p Time.now - t1
```

```
# multi-ractor
(t1 = Time.now
 n1 = 2**61 - 1
 n2 = 2**61 + 15
 r1 = Ractor.new(n1){|p1| p1.prime? }
 r2 = Ractor.new(n2){|p2| p2.prime? }
 [r1.take, r2.take])
p Time.now - t1
```


1 [| 0.7%] 11 [0.0%] 21 [0.0%] 31 [0.0%]
2 [0.0%] 12 [|| 1.3%] 22 [0.0%] 32 [0.0%]
3 [0.0%] 13 [0.0%] 23 [0.0%] 33 [0.0%]
4 [0.0%] 14 [0.0%] 24 [0.0%] 34 [0.0%]
5 [0.0%] 15 [0.0%] 25 [0.0%] 35 [0.0%]
6 [0.0%] 16 [0.0%] 26 [0.0%] 36 [0.0%]
7 [0.0%] 17 [0.0%] 27 [0.0%] 37 [0.0%]
8 [0.0%] 18 [0.0%] 28 [0.0%] 38 [0.0%]
9 [0.0%] 19 [0.0%] 29 [0.0%] 39 [0.0%]
10 [0.0%] 20 [0.0%] 30 [0.0%] 40 [0.0%]
Mem[|| 677M/252G] Tasks: 57, 11 thr; 1 running
1 bash
irb(main):001:0>
0 bash
<0* bash 1 bash>

Demonstration
Object creation on ractors is slower yet

```
def task =
  1_000_000.times{''}

(t1 = Time.now
 2.times{task}
 Time.now - t1)
```

0.2 seconds

```
(t1 = Time.now
 r1 = Ractor.new{ task }
 r2 = Ractor.new{ task }
 r1.take; r2.take
 Time.now - t1)
```

0.7 seconds

Ractor's concept

Limited object sharing

- The biggest difficulties of thread programming is shared everything
- Most of objects are **not shared** with multiple ractors
  - String, Array, Hash, User defined objects…
  - You can not introduce synchronization bugs because they are not needed on Ractors!

Ractor's concept
Shareable objects

- Classes/modules
- Immutable objects (deeply frozen objects)
  - `Ractor.make_sharable(obj)` makes `obj` recursively frozen
- Special shared objects
  - Ractor objects
  - Transactional variables (not introduced yet)
  - Sharable Proc
  - …

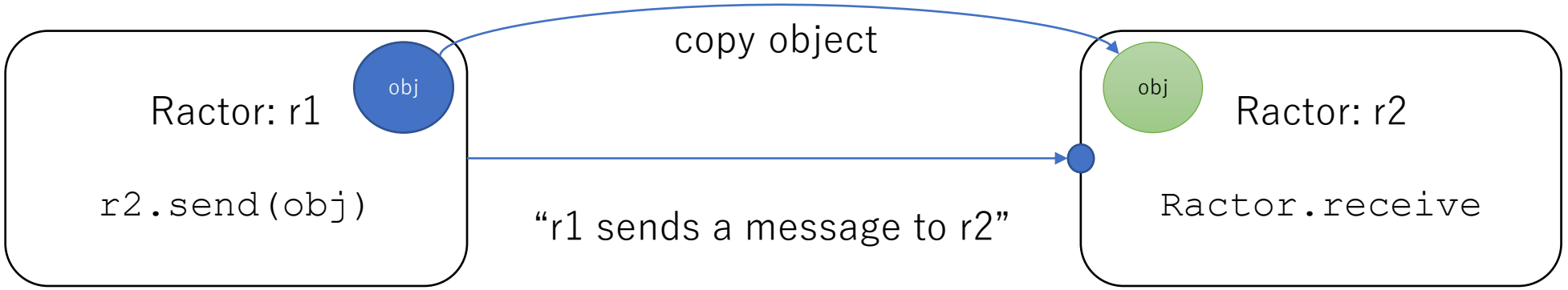
Ractor's concept

Building ractors network with push/pull types communication

- Make a program with multi-ractors network
- Ractors can wait for the message arrival

  → We can manage the control flow
- Two types communication APIs
  - Push type (Ractor#send / Ractor.receive)
  - Pull type (Ractor.yield / Ractor#take)

Ractor's idea
Push type communication



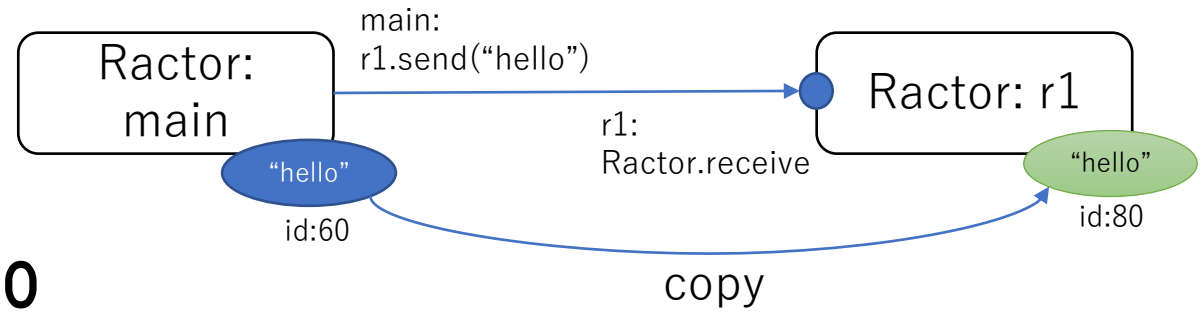
Send a message to r2,
and return "send" immediately

Wait the message,
and return with a
receipt (copied) object

Message passing / Actor style communication

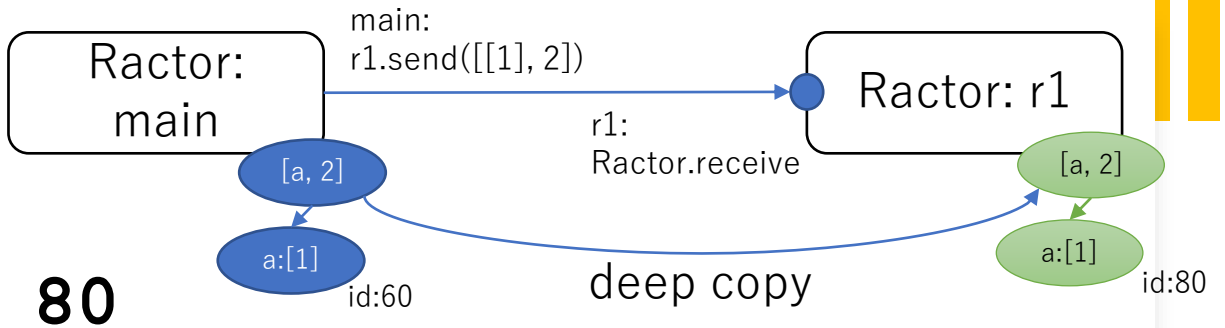
Demonstration
Sending an object

```
r1 = Ractor.new do
  obj = Ractor.receive
  p obj.object_id #=> 80
end
obj = "hello"
p obj.object_id   #=> 60
r1.send(obj)
r1.take # wait for the execution
```

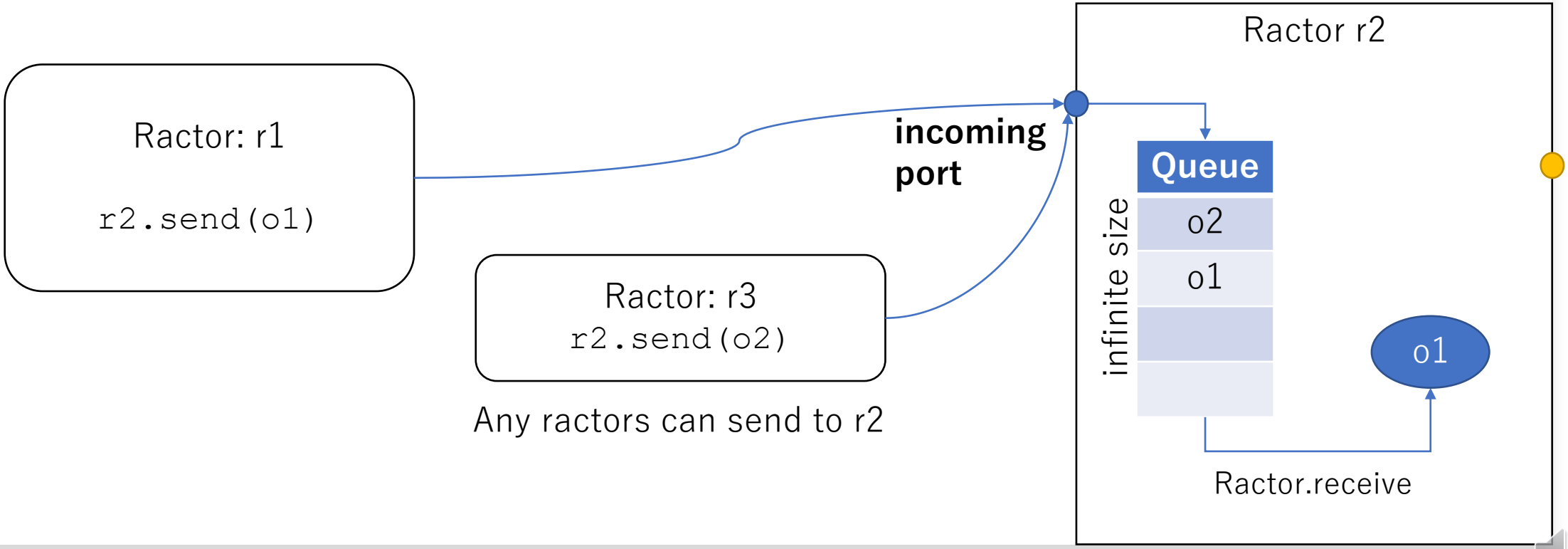


Demonstration
Sending a nested object

```
r1 = Ractor.new do
  obj = Ractor.receive
  p obj[0].object_id #=> 80
end
obj = [[1], 2] # nested array
p obj[0].object_id   #=> 60
r1.send(obj)
r1.take # wait for the execution
```

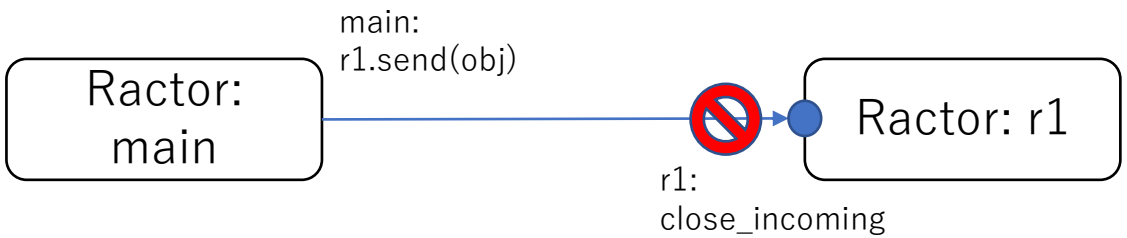


Ractor's idea
Incoming queue and incoming port



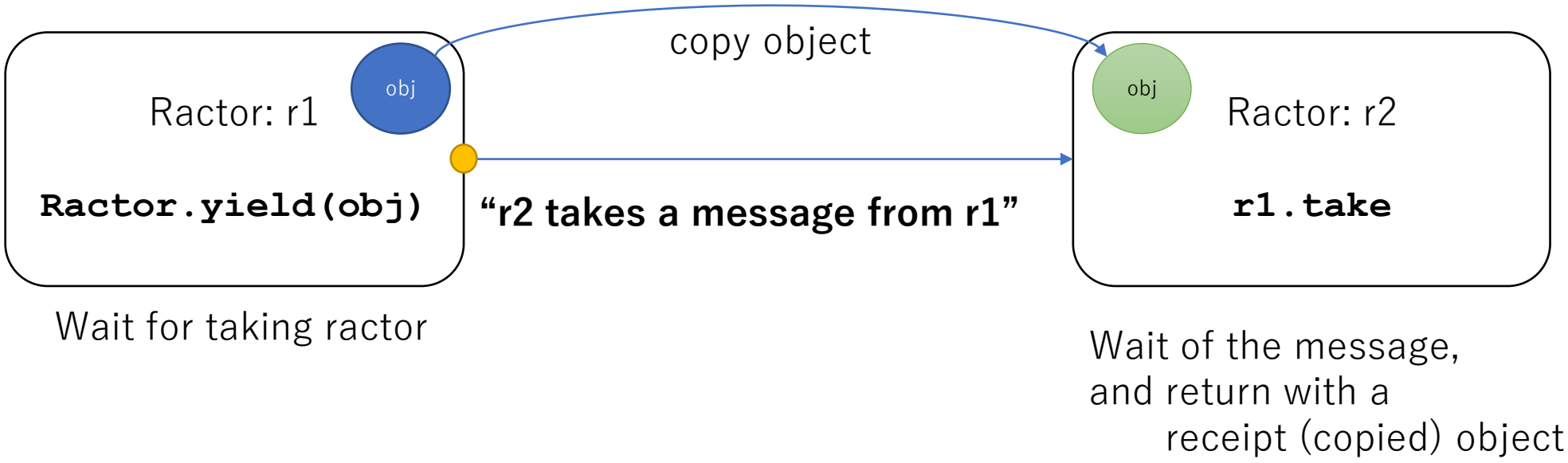
Demonstration
Send to closed port

```
r1 = Ractor.new do
  close_incoming
end
sleep 0.1 # wait for close
r1.send(1)
#=> The incoming-port is already closed (Ractor::ClosedError)
```



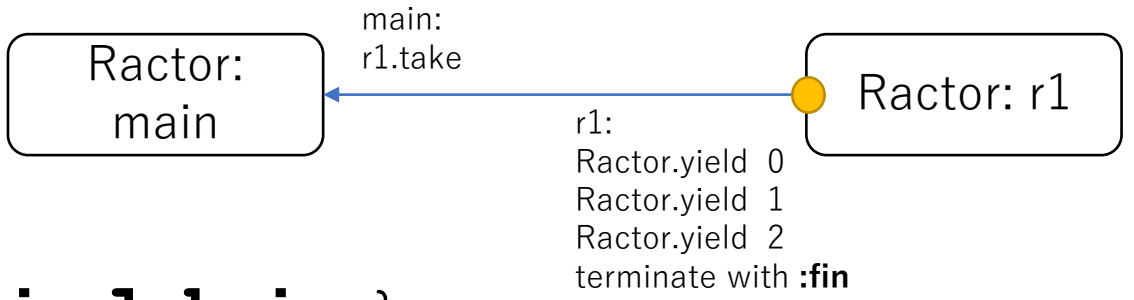
NOTE:
Incoming port will be closed
when the ractor is finished.
→ You can not send to a dead ractor

Ractor's idea
Pull type communication



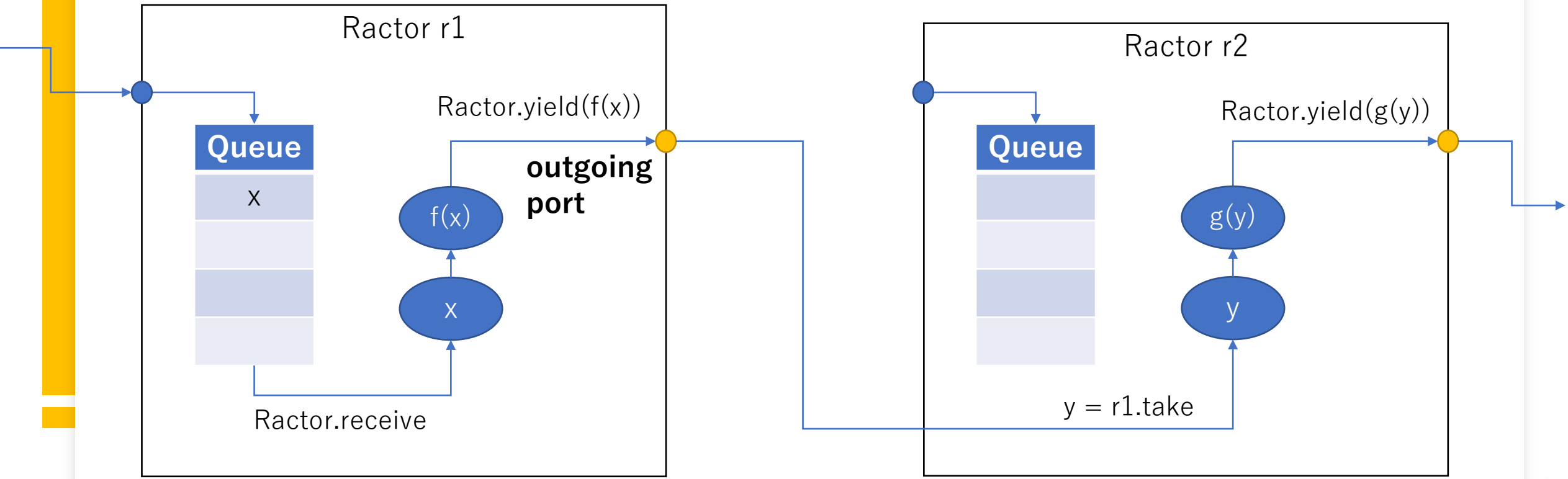
Rendezvous style communication

Demonstration
Pull from a ractor

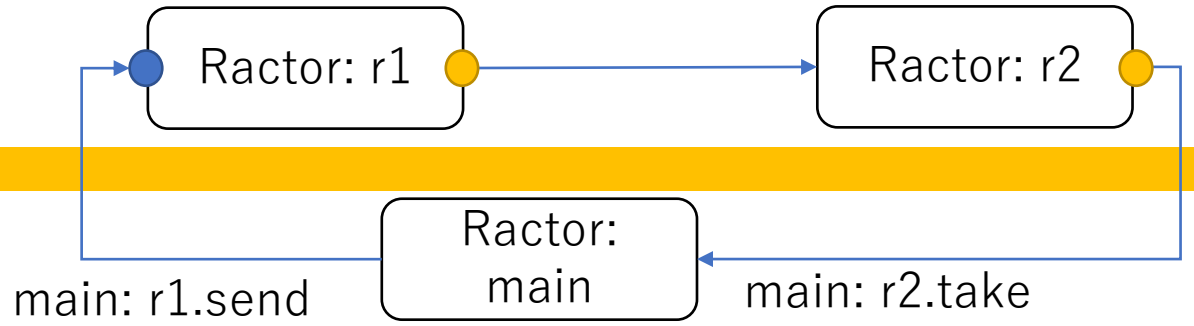



```
r1 = Ractor.new do
  3.times{|i| Ractor.yield i }
  :fin # the result of block will be yielded
end
4.times{ p r1.take } #=> 0 1 2 :fin
```

Ractor's idea
Yield/take via outgoing port

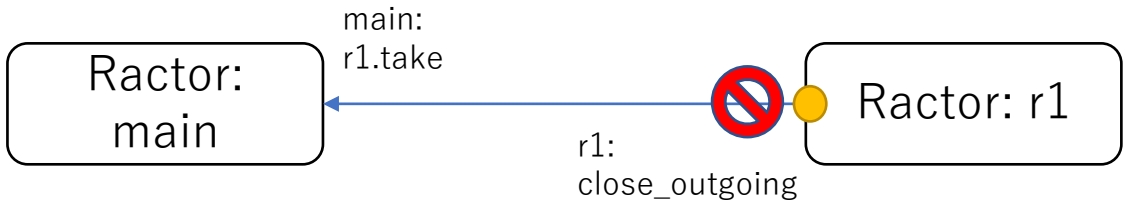


Demonstration
Pipeline parallel




```
def task(n) = n + 1
r1 = Ractor.new do
  loop{ Ractor.yield(task(Ractor.receive)) }
end
r2 = Ractor.new r1 do |r1|
  loop{ Ractor.yield(task(r1.take)) }
end
r1.send(1)
p r2.take #=> 3
```

Demonstration
Close outgoing port

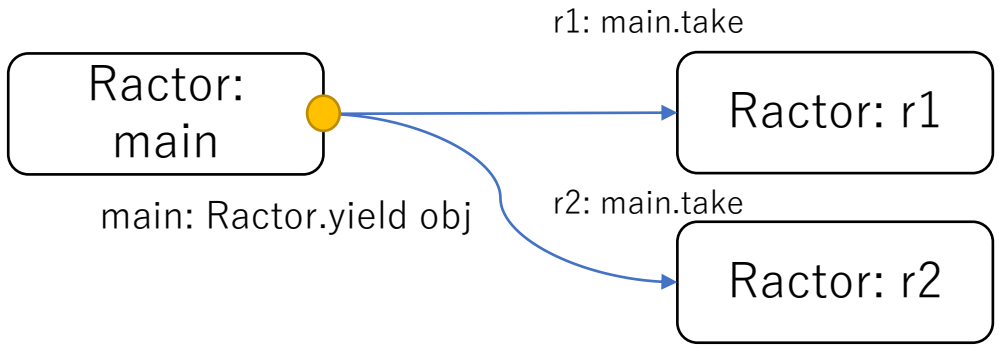



```
r1 = Ractor.new do
  close_outgoing
end
r1.take
#=> `take': The outgoing-port is already closed (Ractor::ClosedError)
```

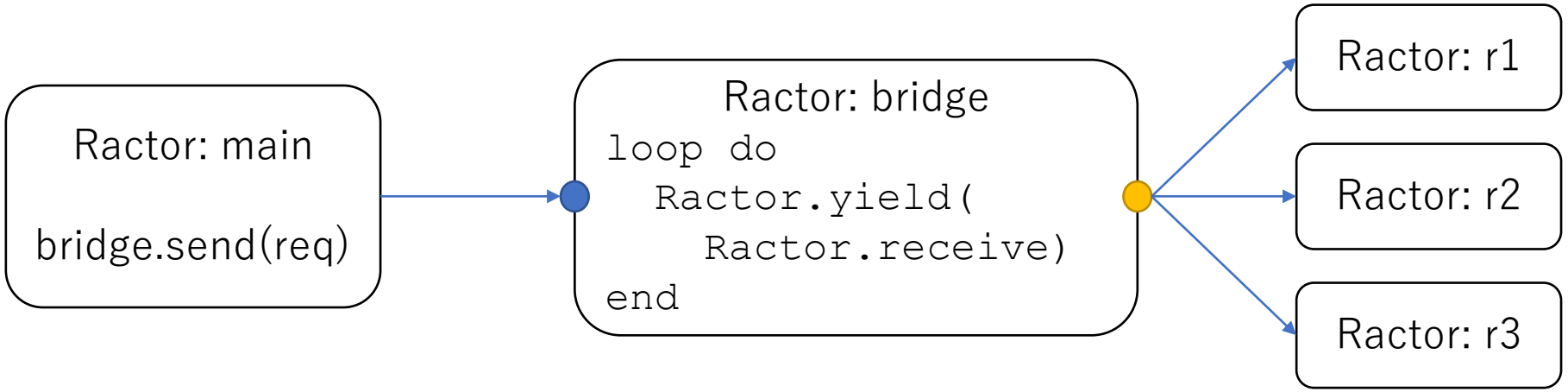
NOTE:
Outgoing port will be closed
when the ractor is finished.
→ You can not take from a dead ractor

Demonstration
Taken by multi-ractors

```
main = Ractor.current
r1 = Ractor.new main do |main|
  p r1: main.take
end
r2 = Ractor.new main do |main|
  p r2: main.take
end

Ractor.yield(:messaeg)
#=> r1 or r2 take a message
```

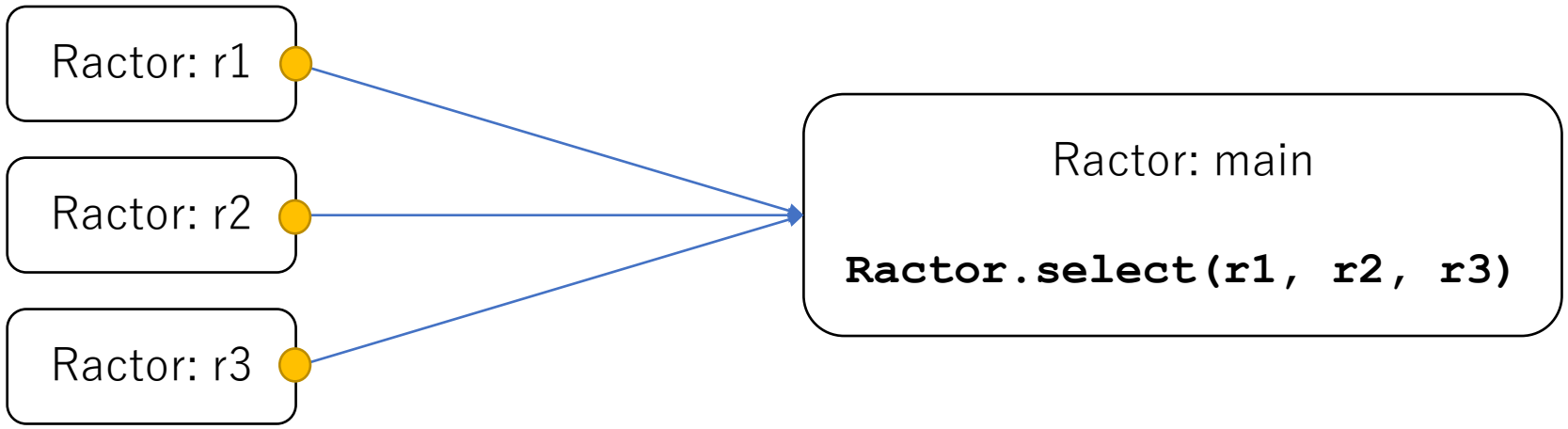


Ractor's idea
Load balancing with a bridge ractor

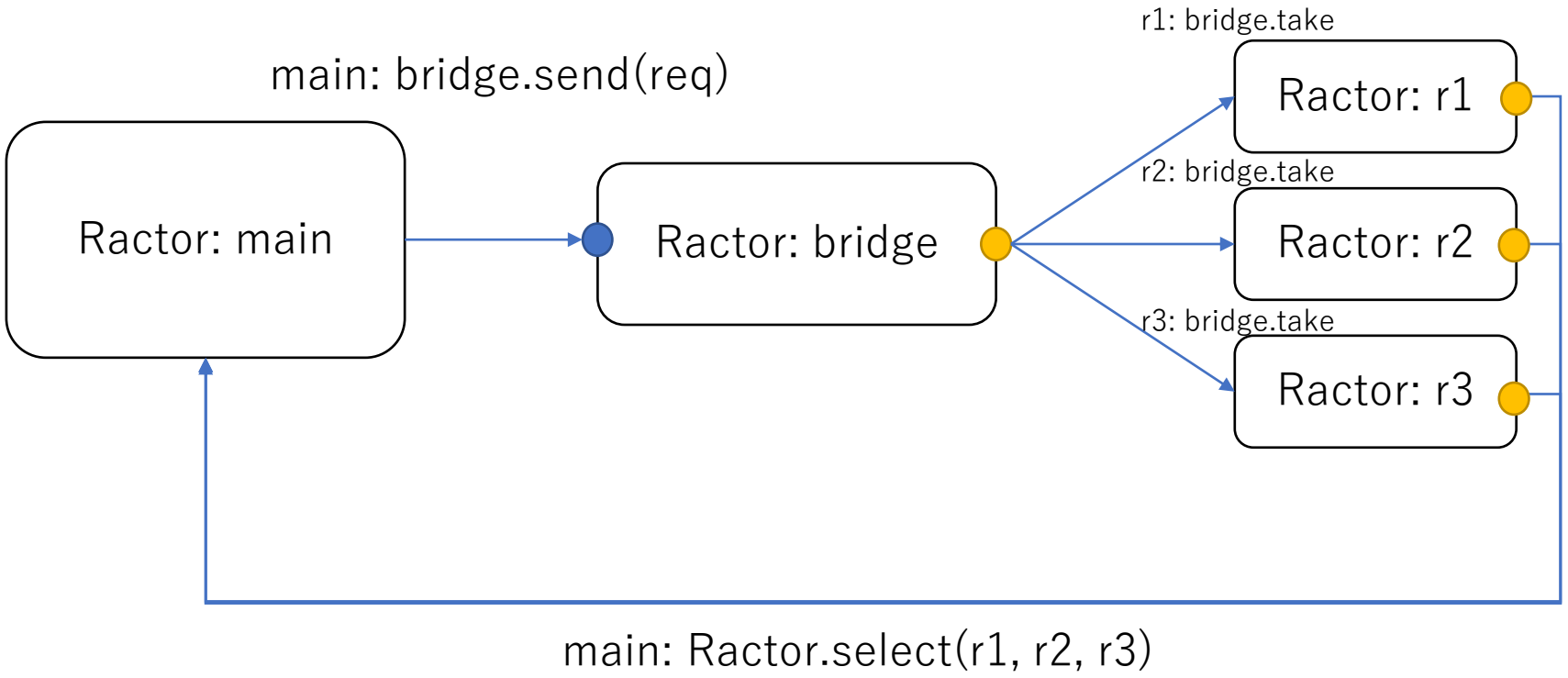


A sent "obj" will be received by **idle ractor** r1, r2 or r3

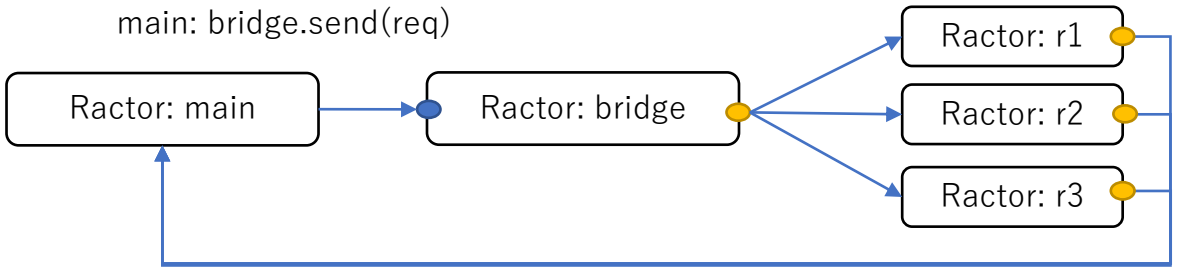
Ractor's idea
Taking from multiple ractors with "select"



Ractor's idea
Get the results from worker pool

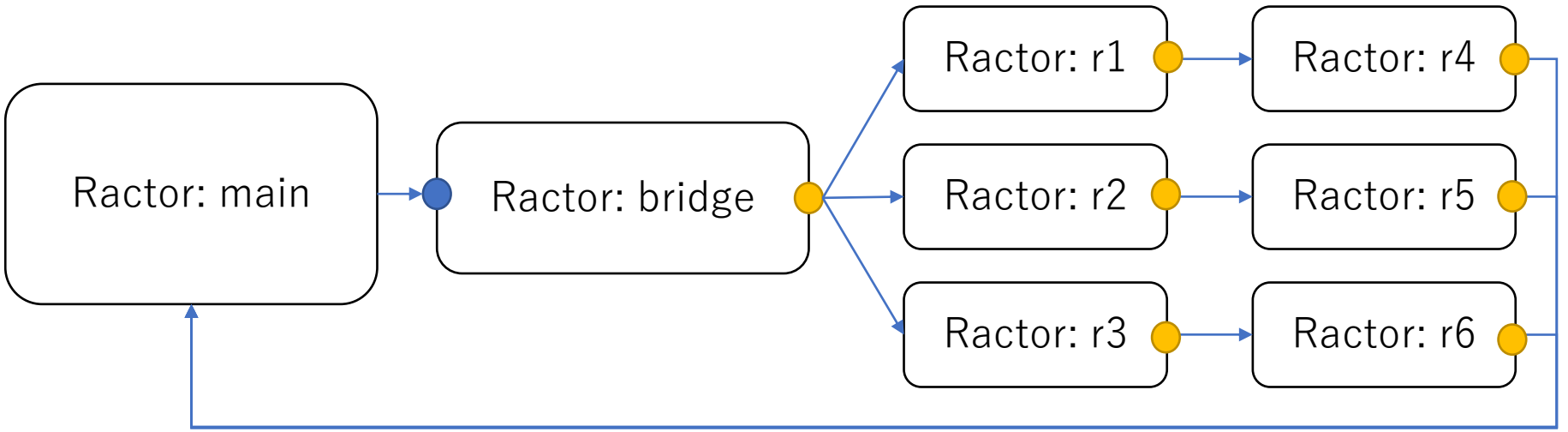


Demonstration
Workers pool




```
require "prime"
def task(n) = [n, n.prime?]
bridge = Ractor.new{loop{Ractor.yield Ractor.receive}}
workers = (1..3).map{|i|
  Ractor.new(bridge, name: "r#{i}"){|b|
    loop{Ractor.yield task(b.take)}}}
3.times{|i| bridge.send 11 + i}          # send 3 requests
3.times{ p Ractor.select(*workers) } # take 3 responses
#=> [#<Ractor:#3 r1 ...>, [11, true]]
#=> [#<Ractor:#3 r1 ...>, [13, true]]
#=> [#<Ractor:#4 r2 ...>, [12, false]]
```

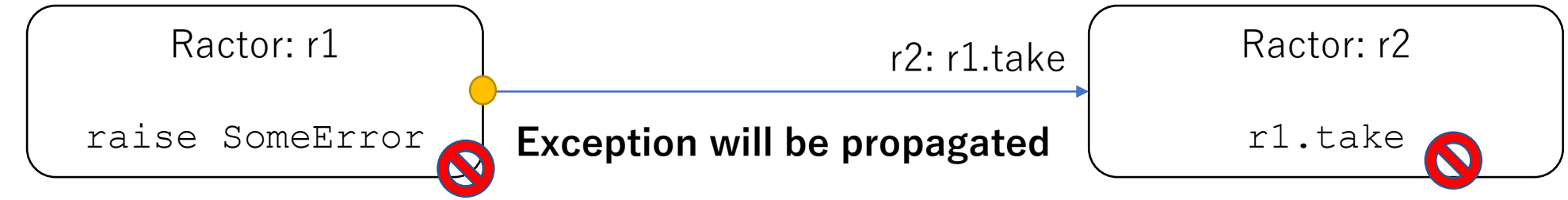
Ractor's idea
More complex ractor network



main: Ractor.select(r4, r5, r6)

r1~r6 run their task in parallel

Ractor's idea
Exception propagation



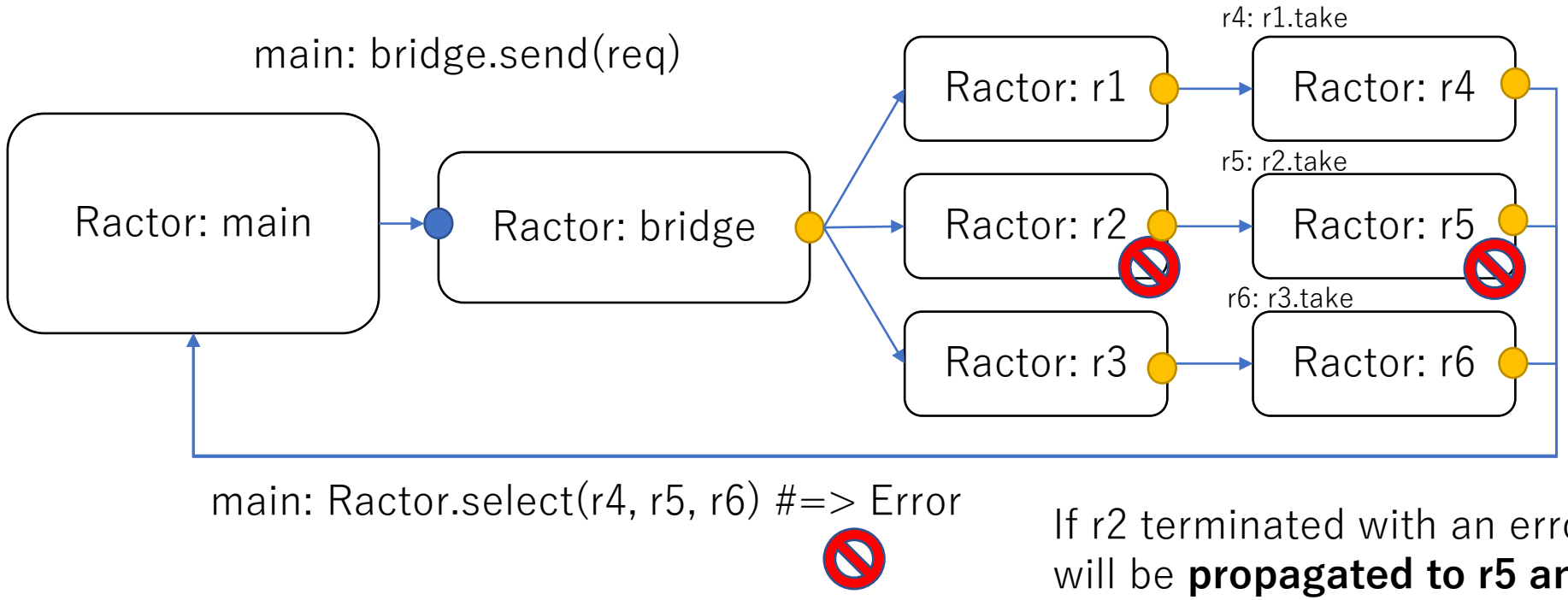
A ractor finished with an error

r1.take raises a RemoteError
(like `Thread#join`)

Demonstration
Exception propagation

```
r1 = Ractor.new do
  1+"2" #=> TypeError
end
begin
  r1.take
rescue Ractor::RemoteError => e
  p e         #=> #<Ractor::RemoteError: thrown by remote Ractor.>
  p e.cause   #=> #<TypeError: String can't be coerced into Integer>
  p e.ractor #=> #<Ractor:#2 … terminated>
end
```

Ractor's idea
Error recovery on the ractor network



If r2 terminated with an error, the error will be **propagated to r5 and main**.
main can restart r2 and r5 if needed.

Important semantic changes

- **Completely compatible with Ruby 2.x** if there is only the main Ractor (first created Ractor)
- Limited to the main Ractor
  - Global variables `$gv`
    - Some (`$stdout`, `$$` …) are Ractor local
  - Class variables `@@cv`
  - **Instance variables of shareable objects**
    - **Ivars of class/module are prohibited**
  - **Constants refer to unshareable objects**
    - `C = [1]` is prohibited
- **For Ractor programming, many modifications are needed**
  - We are discussing how to provide an easy way to make Ractor libraries

Ractor implementation progress

- [x] Basic Ractor APIs
- [ ] Advanced APIs
- [x] Ruby apps without Ractor
- [ ] Complex application with Ractor (not enough synchronizations)
- [ ] Existing Ruby's API considerations
- [ ] C-extension supports
- [ ] Performance tuning

```
$ ./miniruby -e Ractor.new{}
<internal:ractor>:37: warning: Ractor is experimental,
and the behavior may change in future versions of Ruby!
Also there are many implementation issues.
```

More interesting features…

- Sending message with copying/moving semantics
- Shareable “Proc” semantics
- Ractor-safe and efficient internal implementations
- …

Reference

- Ractor specification: https://github.com/ko1/ruby/blob/ractor_parallel/doc/ractor.md
- RubyKaigi 2020 takeout presentation (“Ractor report”)
  - More detailed data and programming models
  - https://www.youtube.com/watch?v=40t8EPpnujg
  - http://www.atdot.net/~ko1/activities/2020_rubykaigi.pdf

Conclusion

- Ruby can run in parallel with Ractor without thread-safety headache
- You can enjoy ractor programming on Ruby 3.0
- Ractor API and implementation is not matured
  - Ruby 3.0 is a Ractor preview release
  - Your comments on your experience are welcome 😍

RUBYCONF 2020

Ractor Demonstration

Koichi Sasada
Cookpad Inc.