Ruby's Concurrency Management: Now and Future

Koichi Sasada

ko1@cookpad.com

RUBYCONF
JAKARTA 2017

cookpad

Today’s talk

- Supported features
  - Process
  - Thread
  - Fiber
- Features under consideration
  - Guild
  - Auto-Fiber

Today’s talk

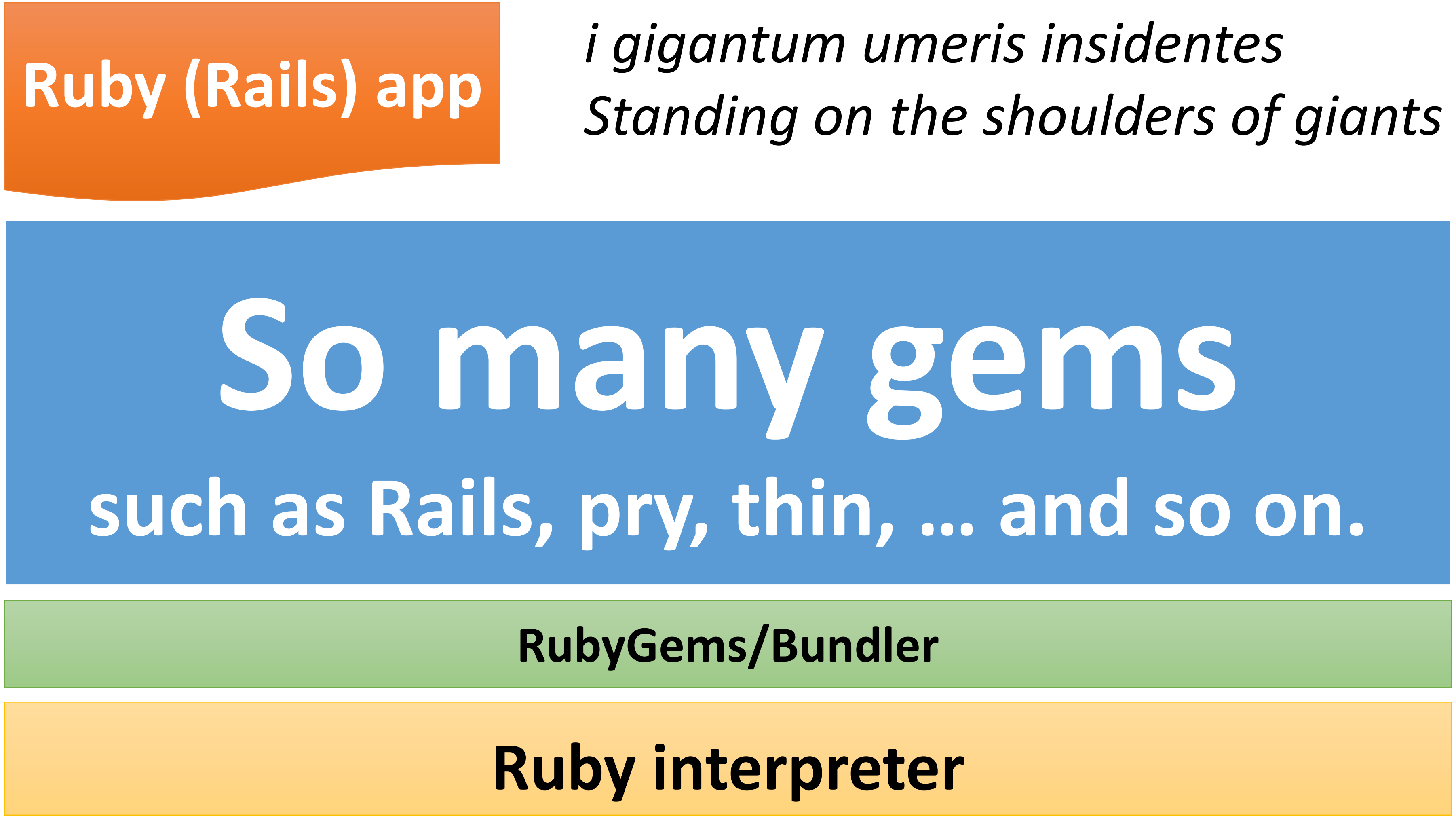
	Process	**Guild**	**Thread**	**Auto-Fiber**	**Fiber**
Available	Yes	**No**	Yes	**No**	Yes
Switch on time	Yes	Yes	Yes	**No**	**No**
Switch on I/O	Auto	Auto	Auto	**Auto**	No
Next target	Auto	Auto	Auto	Auto	**Specify**
Parallel run	**Yes**	**Yes**	No (on MRI)	No	No
Shared data	**N/A**	**(mostly) N/A**	Everything	Everything	Everything
Comm.	Hard	**Maybe Easy**	**Easy**	**Easy**	**Easy**
Programming difficulty	Hard	**Easy**	Difficult	**Easy**	**Easy**
Debugging difficulty	Easy?	**Maybe Easy**	Hard	Maybe hard	**Easy**

Koichi Sasada

http://atdot.net/~ko1/

- A programmer
  - 2006-2012 Faculty
  - 2012-2017 Heroku, Inc.
  - 2017- Cookpad Inc.
- Job: MRI development
  - MRI: Matz Ruby Interpreter
  - Core parts
    - VM, Threads, GC, etc

Normal Ruby developer’s view



Ruby (Rails) app

i gigantum umeris insidentes
Standing on the shoulders of giants

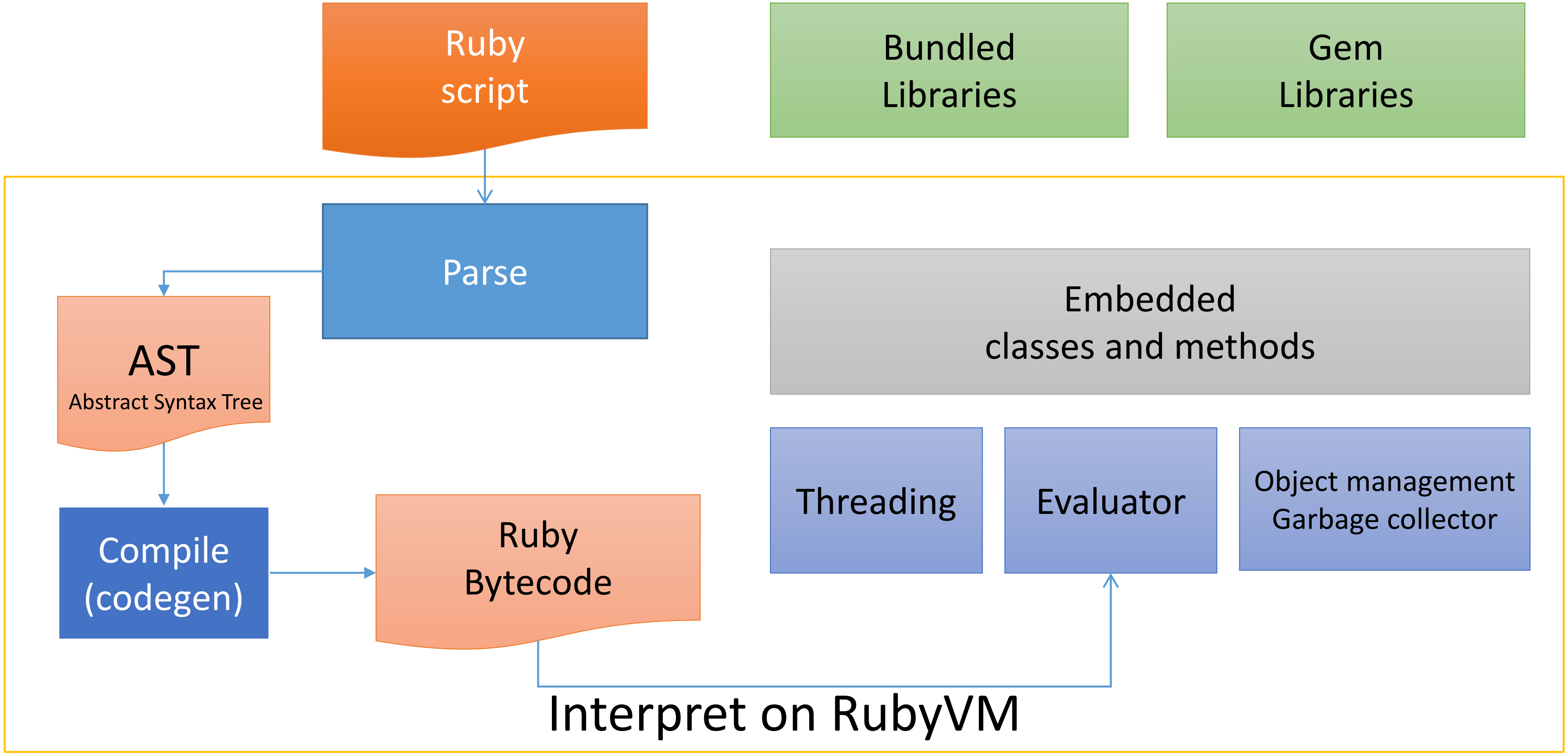
So many gems

such as Rails, pry, thin, … and so on.

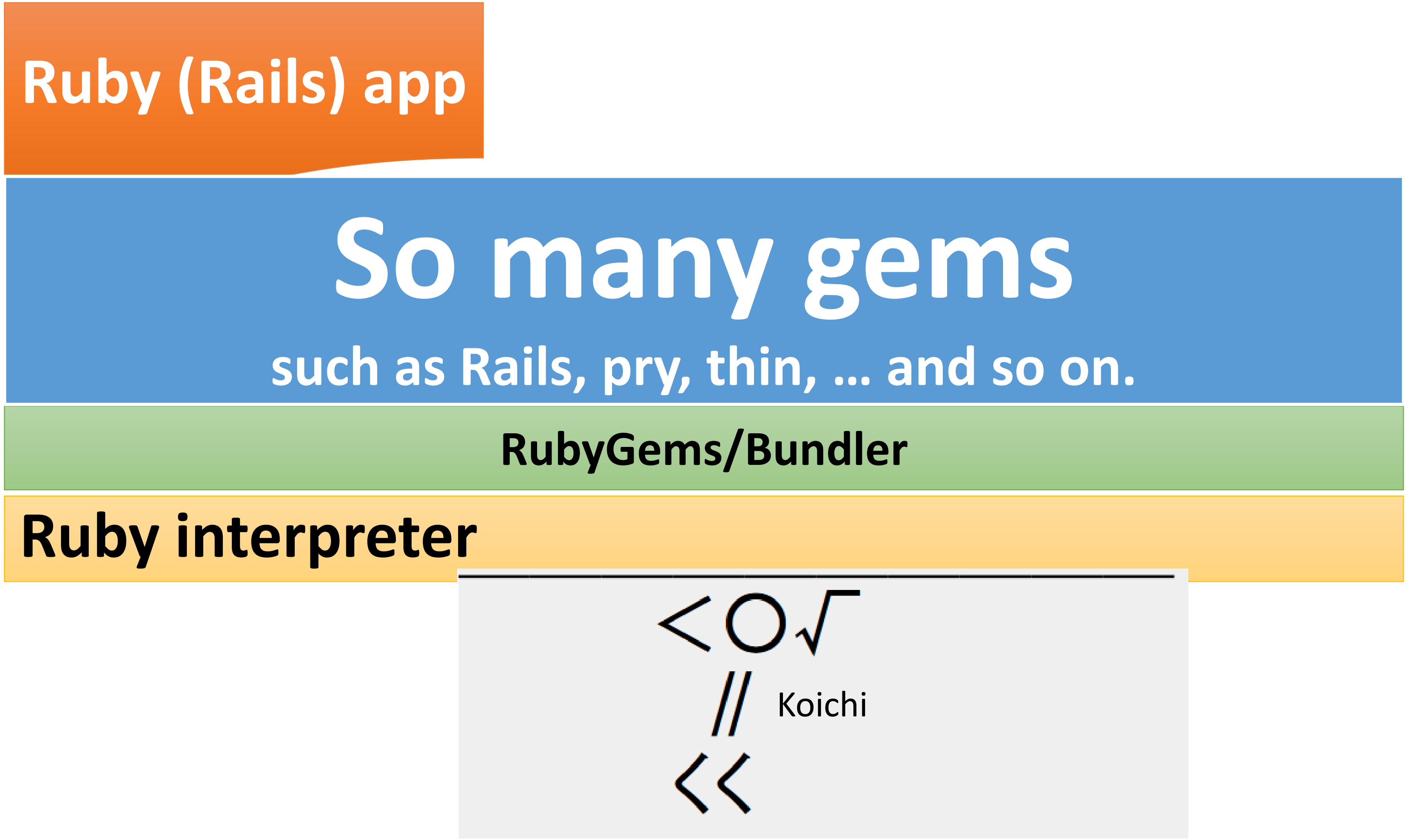
RubyGems/Bundler

Ruby interpreter

Normal MRI developer’s view



Koichi’s job



Ruby (Rails) app

So many gems

such as Rails, pry, thin, … and so on.

RubyGems/Bundler

Ruby interpreter

Ruby3: Ruby3 has 3 goals

- Static type checking
- Just-in-Time (JIT) compilation
- Parallel execution w/ highly abstract concurrent model

Ruby3: Ruby3 has 3 goals

- For productivity
  - Static checking
- For performance
  - Just-in-Time (JIT) compilation
  - **Parallel execution w/ highly abstract concurrent model**

Concurrency

“In computer science, **concurrency** is the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units.[1] This means that even if the concurrent units of the program, algorithm, or problem are executed out-of-order or in partial order, the final outcome will remain the same. This allows for parallel execution of the concurrent units, which can significantly improve overall speed of the execution in multi-processor and multi-core systems.”

https://en.wikipedia.org/wiki/Concurrency_(computer_science)

Concurrent and Parallel execution

Concurrent execution

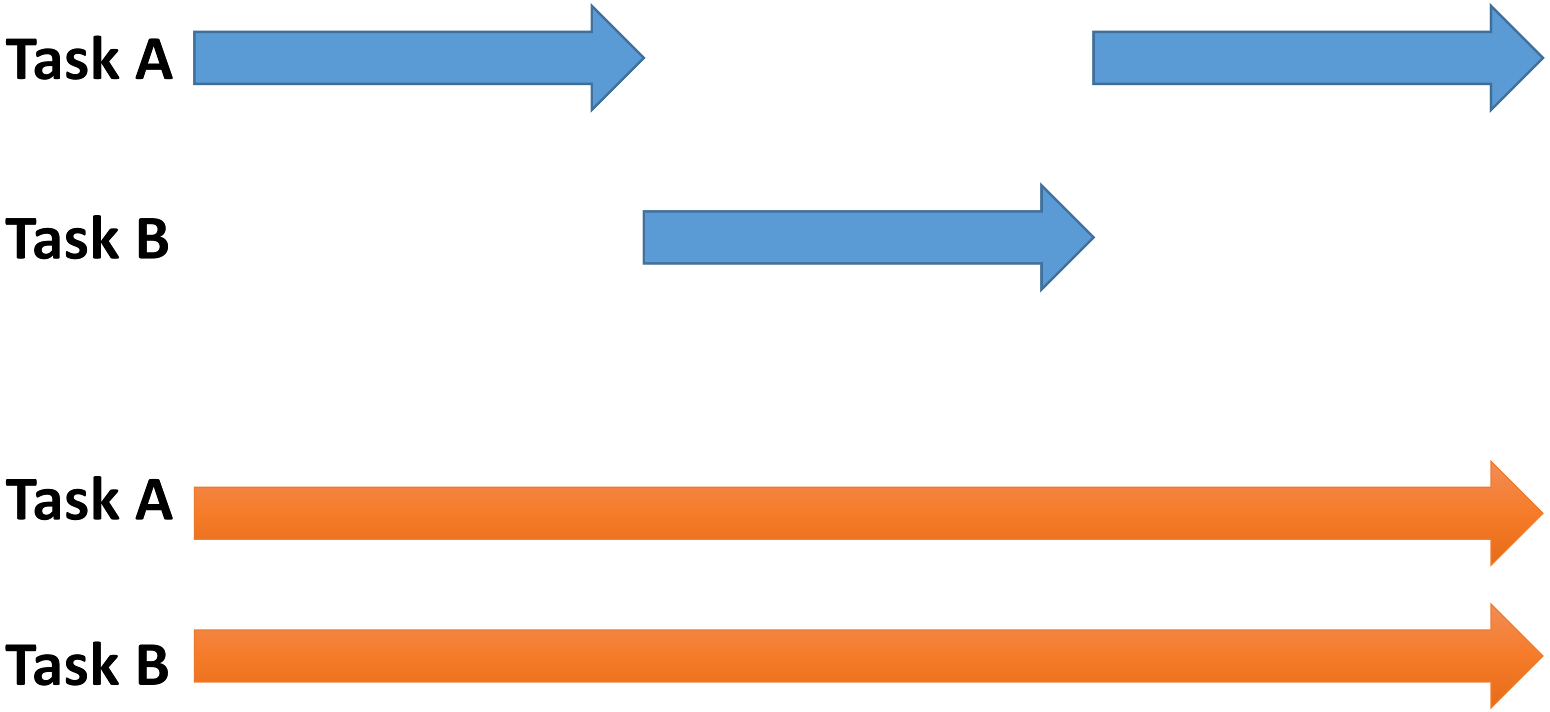
Logical concept

Parallel

(and concurrent)

execution

Physical concept



Task A

Task B

Task A

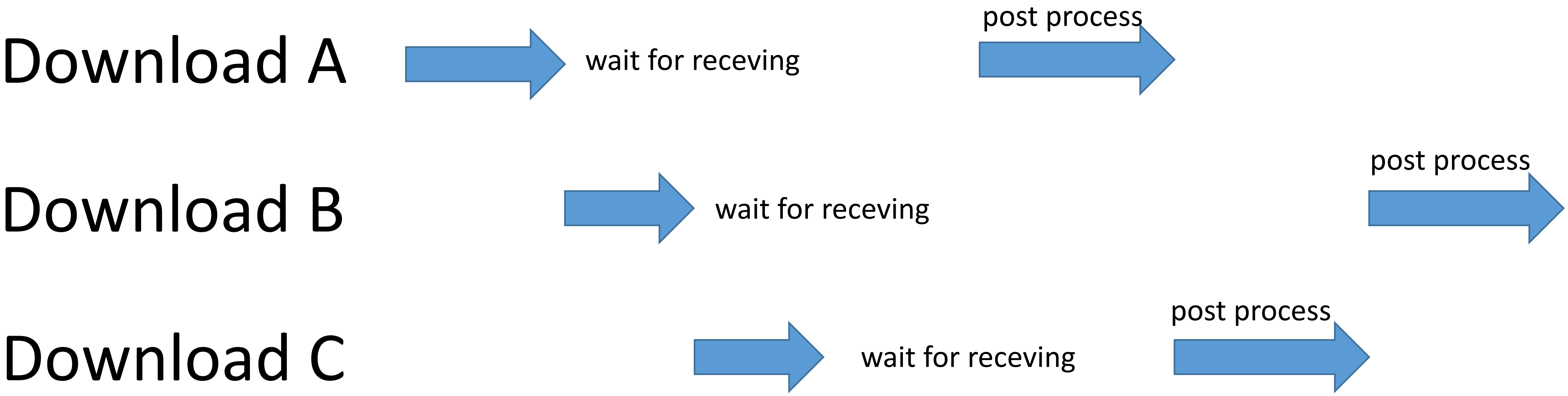
Task B

Ruby (MRI) support only concurrency

Concurrency
Why needed?

- Easy to write some kind of programs
  - Download files **simultaneously**
  - Process web requests **simultaneously**
  - Agent simulation (assume computer games)
    - Each agent has its own logics
    - Run agents **simultaneously**

Concurrency Example: Downloader



Download A

wait for receving

post process

Download B

wait for receving

post process

Download C

wait for receving

post process

We can write this kind of program **w/o concurrency support**, but **not simple, not easy**

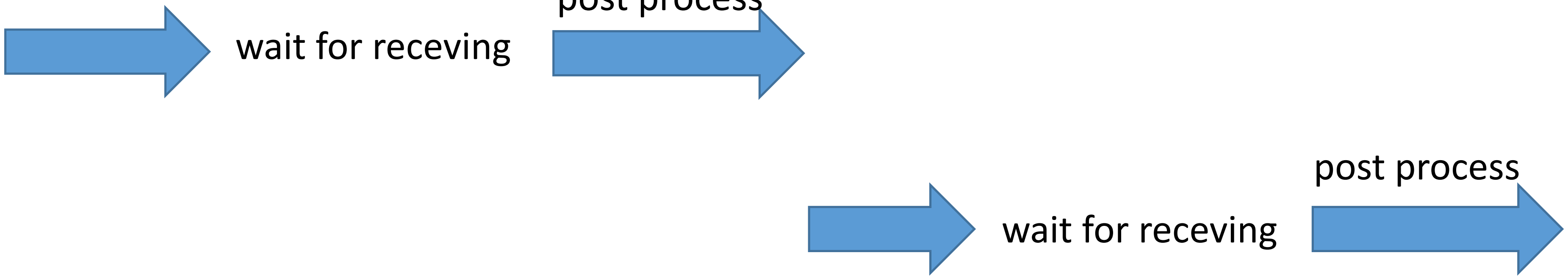
Downloader example
With concurrency support (Thread)

```
ts = URLs.map do |url|
  Thread.new(url) do |u|
    data = download(u)
    File.write(u.to_fname, data)
  end
end.each{|th| th.join} # wait
```

Downloader example
Without concurrency support

```
# Serial execution
URLs.each do |u|
  data = download(u)
  File.write(u.to_fname, data)
end
```

Concurrency
Not concurrent case



Download A

wait for receving

post process

Download B

wait for receving

post process

Download C

… and download C after that

Downloader example
Without concurrency support

```
# Use select. Not so SIMPLE!!
fds = URLs.map do |u|
  download_fd(u)
end

while ready_fds = select(fds)
  ready_fds.each{|fd|
    File.write(…, read(fd))}
end
```

Existing concurrency supports on Ruby (MRI)

Supported features by Ruby/MRI

- Process
- Thread
- Fiber

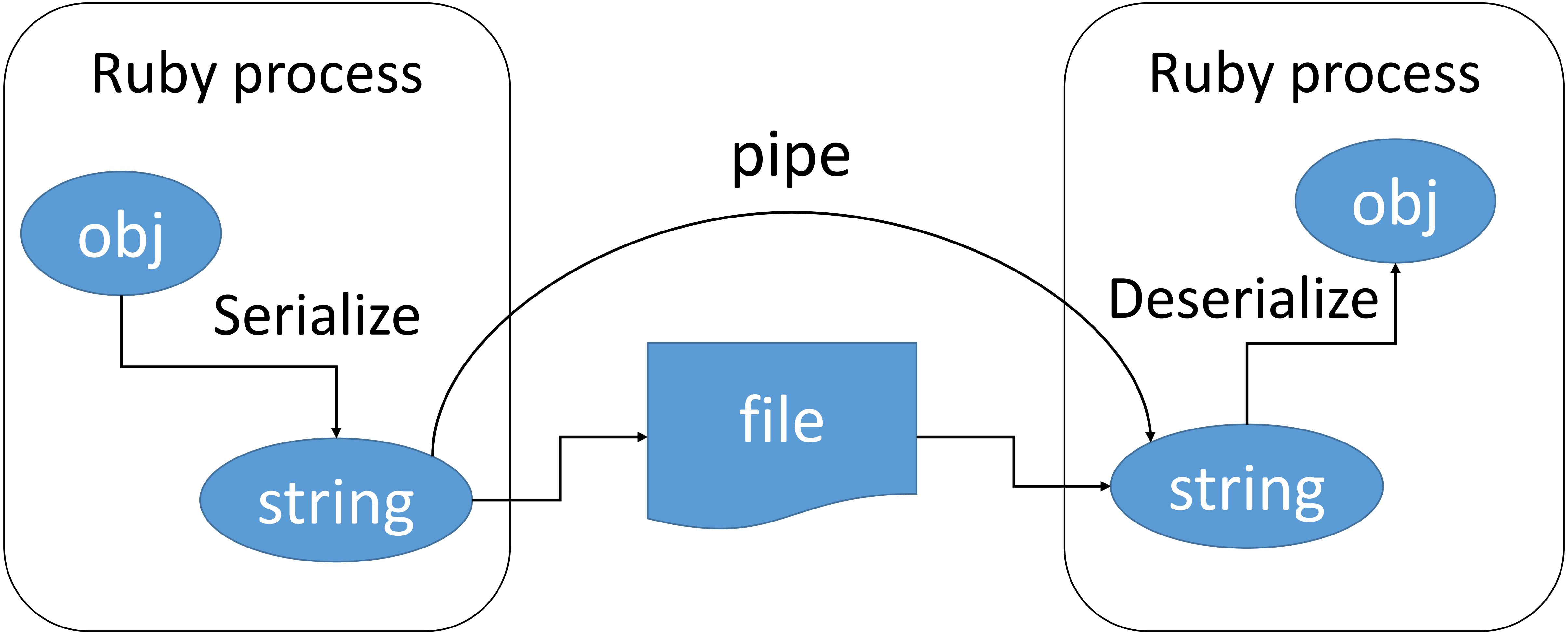
Process

Traditional concurrency support

Process

- Use OS multi-process
    - Use fork on Unix-like systems
- Shared-nothing
    - Communicate with IPC (pipe, etc) such as `IO.pipe`
- Programming
    - Difficult to manage processes and IPC
- Debugging
    - Easy because a few synchronization bugs

Inter-process communication



Inter-process communication
Example code

```
# Traditional multi-process example

r, w = IO.pipe
fork do
  result_str = work_something.to_s
  w.write result_str
  w.close
end
puts r.read # wait for a result
```

Sophisticated libraries/frameworks for process programming

- dRuby: Distributed object for Ruby
- parallel gem: Parallel programming with processes
- unicorn: Process based web application server (master – worker model w/ processes)

Thread

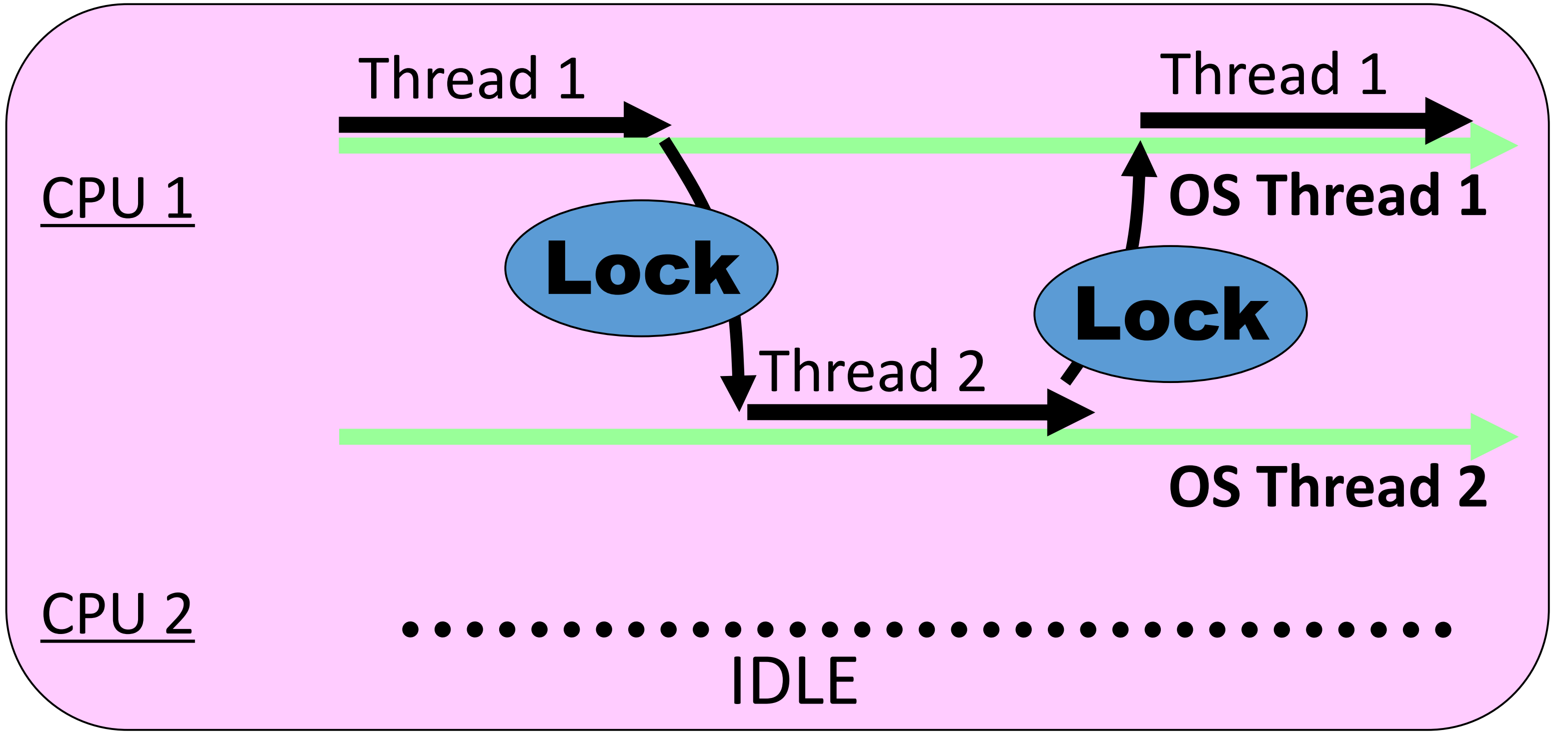
Ruby’s native concurrency support

Thread

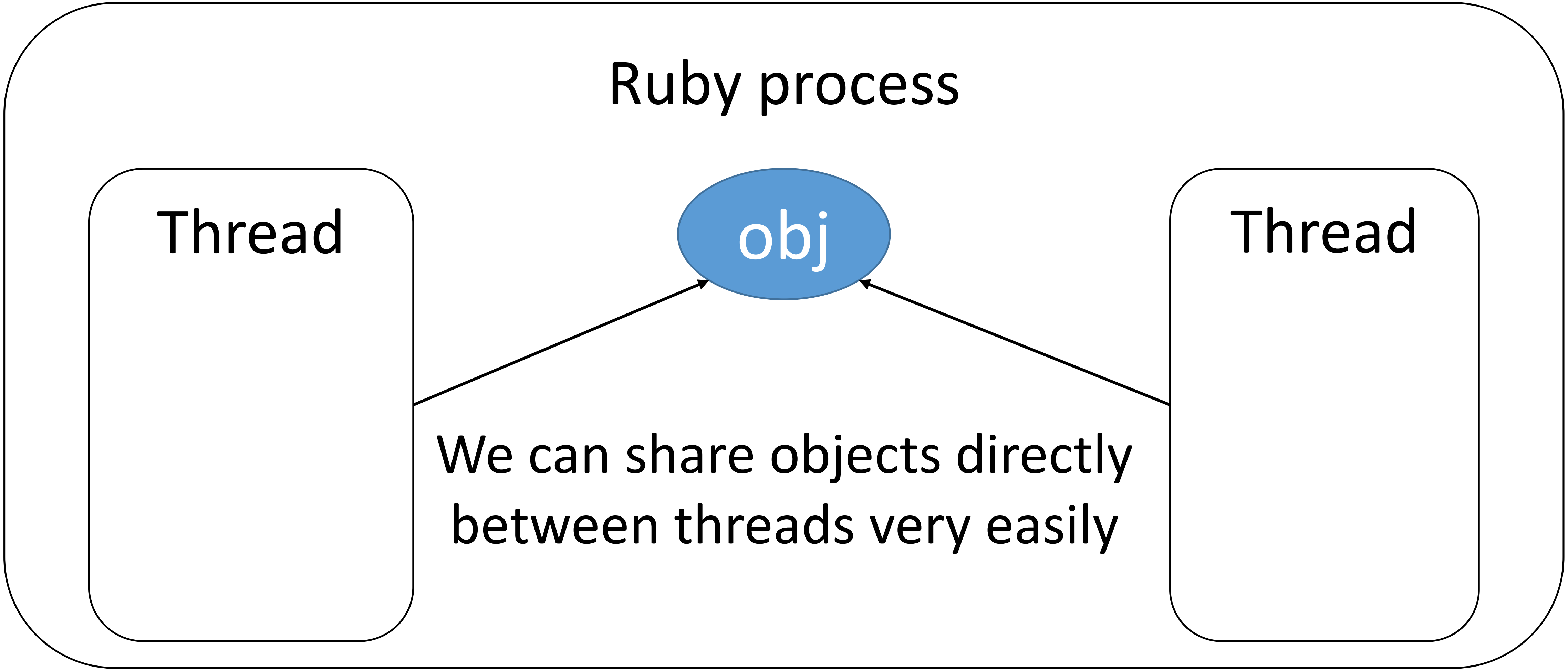
- Use Ruby managed threads
  - `Thread.new do … end`
- Shared-everything
  - Communication is very easy
- Programming
  - Easy to make, easy to communicate (at a glance)
  - Difficult to make completely safe program
- Debugging
  - Hard because of synchronization

MRI: Thread with Giant Lock (GIL)

- Only a thread keeping the GIL can run (can’t run in parallel)



Inter-thread communication
Easy to share objects



We can share objects directly between threads very easily

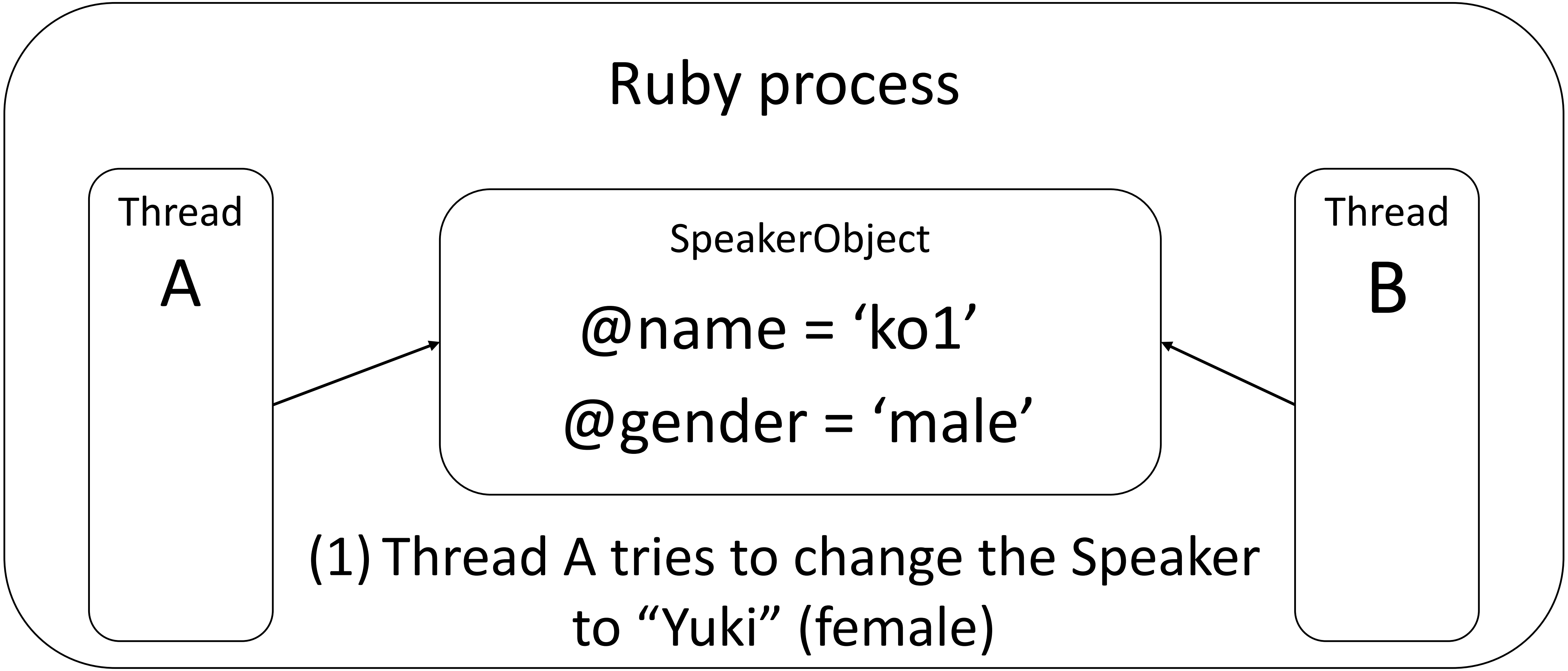
Inter-thread communication

```ruby
v = Object.new
$g = Object.new
Thread.new do
  p [v, $g]
end
p [v, $g]
```

Thread programming
Synchronization is required

- Reading/writing data simultaneously w/o synchronization will cause serious problem
    - Race condition
    - Data race

Mutate shared objects
Lucky case



Ruby process

Thread A

SpeakerObject

@name = ‘ko1’

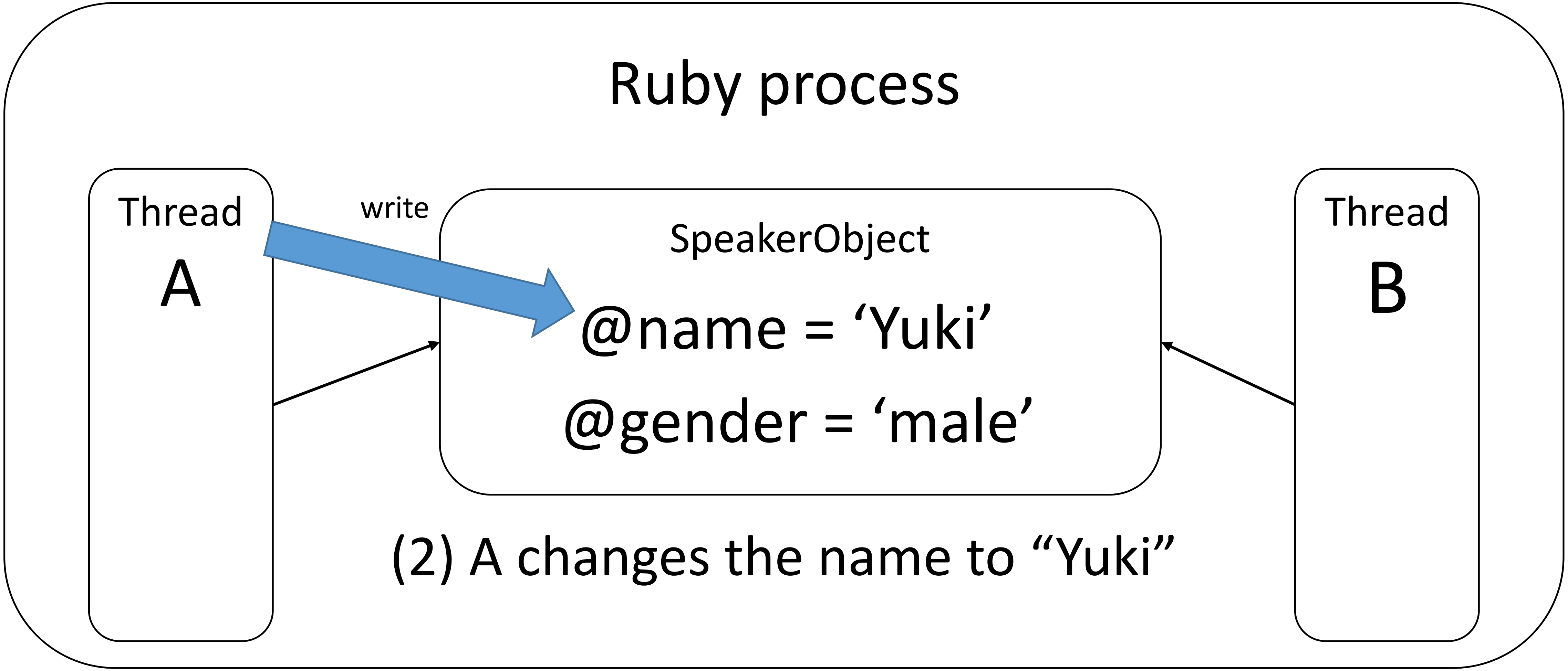
@gender = ‘male’

Thread B

(1) Thread A tries to change the Speaker to “Yuki” (female)

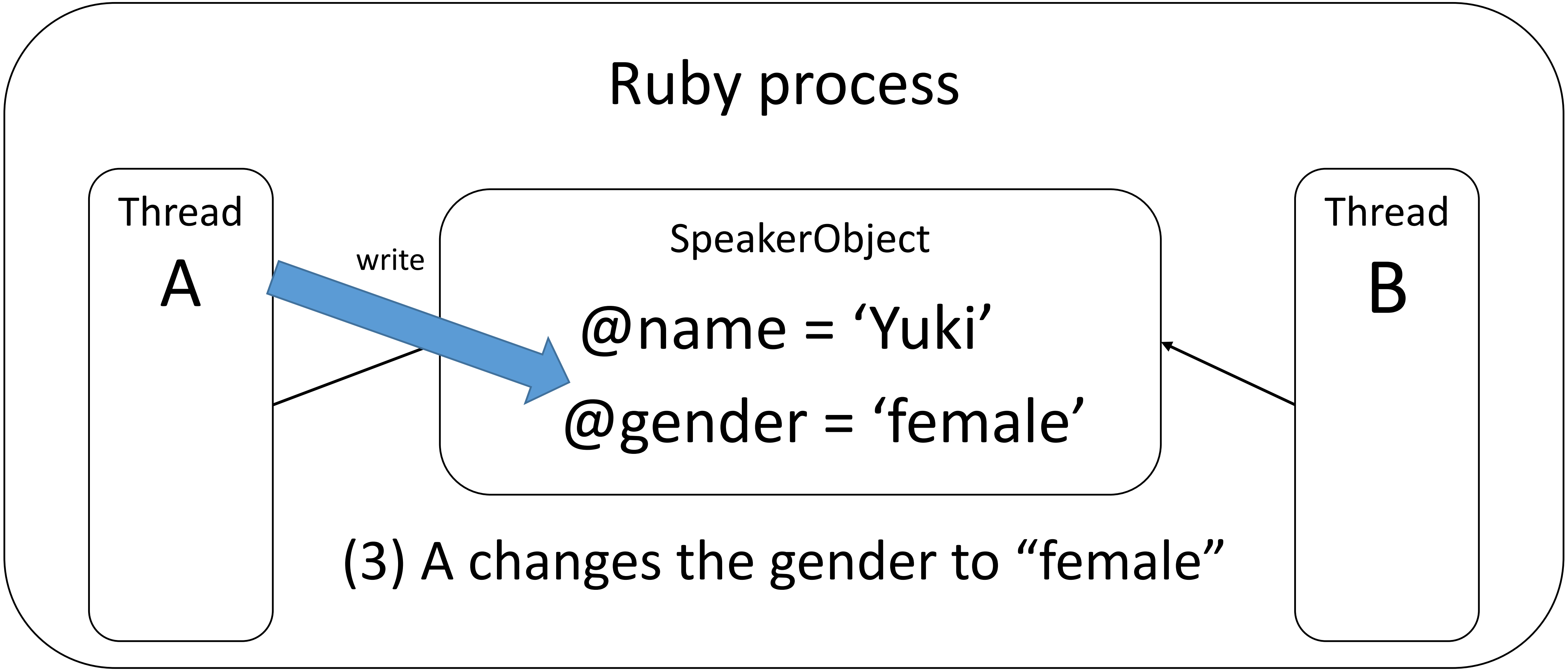
Note: Yuki is my wife.

Mutate shared objects
Lucky case



(2) A changes the name to “Yuki”

Mutate shared objects
Lucky case



Ruby process

Thread A

write

SpeakerObject

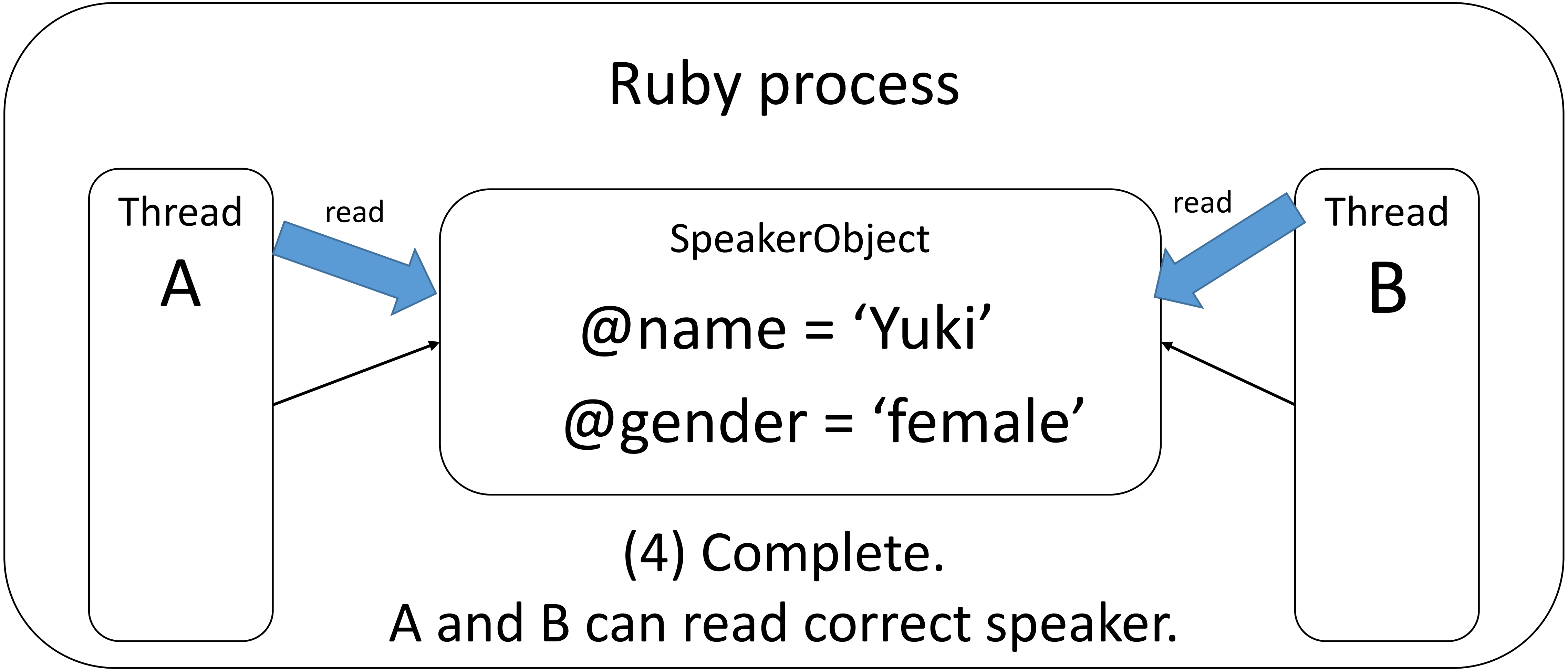
@name = ‘Yuki’

@gender = ‘female’

Thread B

(3) A changes the gender to “female”

Mutate shared objects
Lucky case


Ruby process

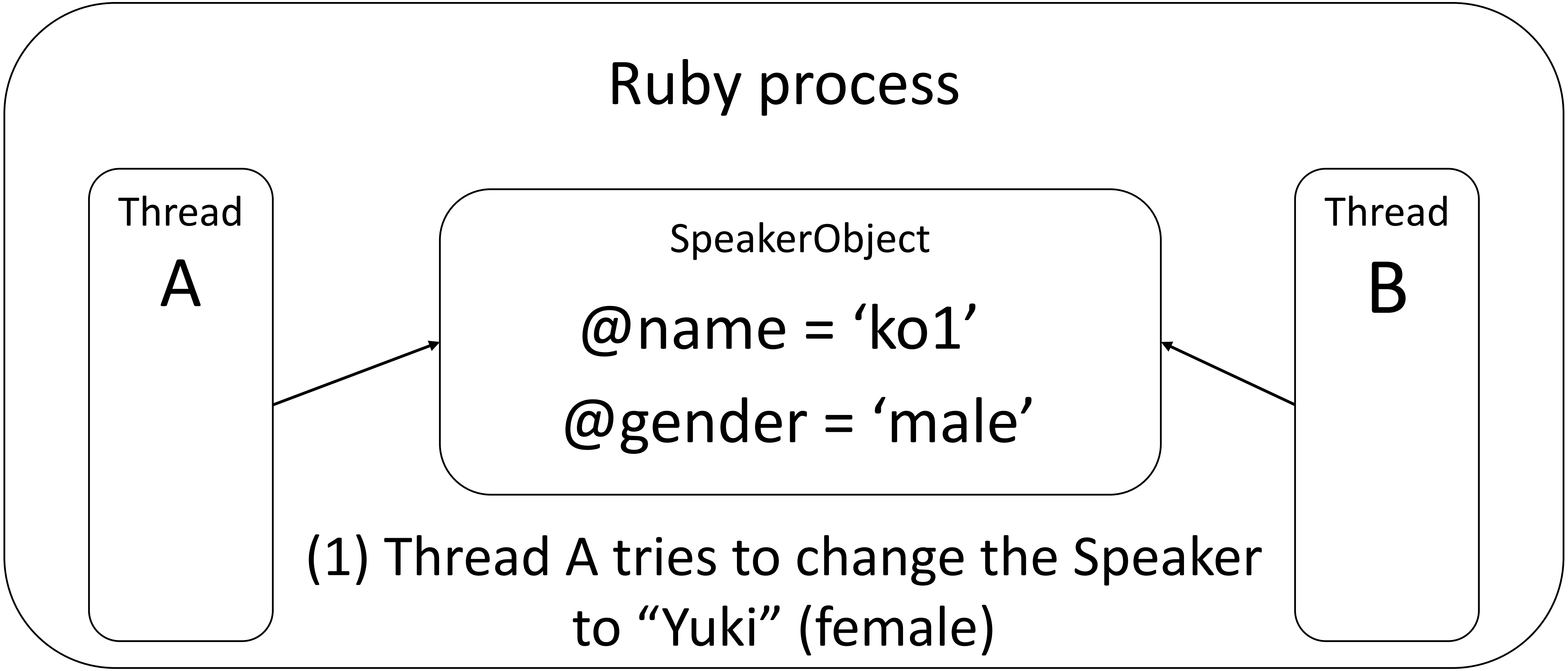
@name = ‘Yuki’

@gender = ‘female’

(4) Complete.

A and B can read correct speaker.

Mutate shared objects
Problematic case



Ruby process

Thread A

SpeakerObject

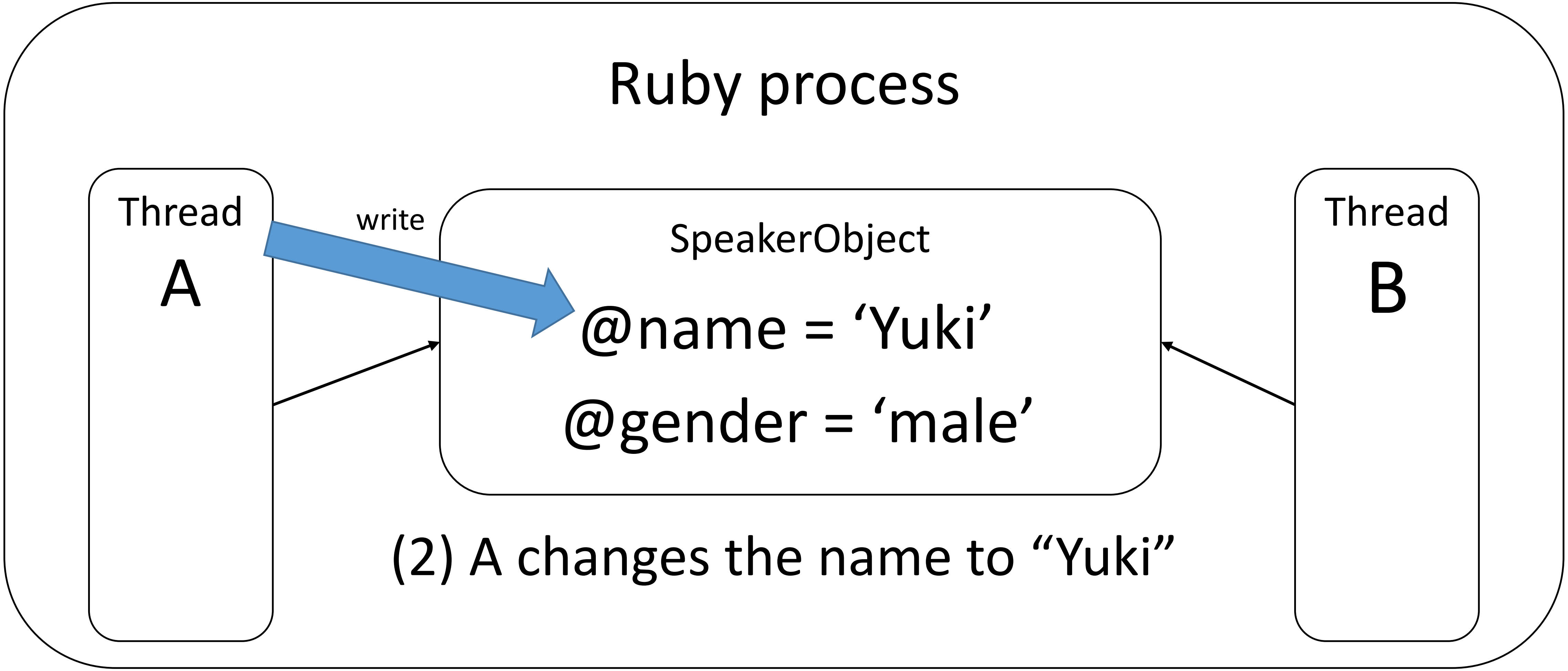
@name = ‘ko1’

@gender = ‘male’

Thread B

(1) Thread A tries to change the Speaker to “Yuki” (female)

Mutate shared objects
Problematic case



Ruby process

Thread A

write

SpeakerObject

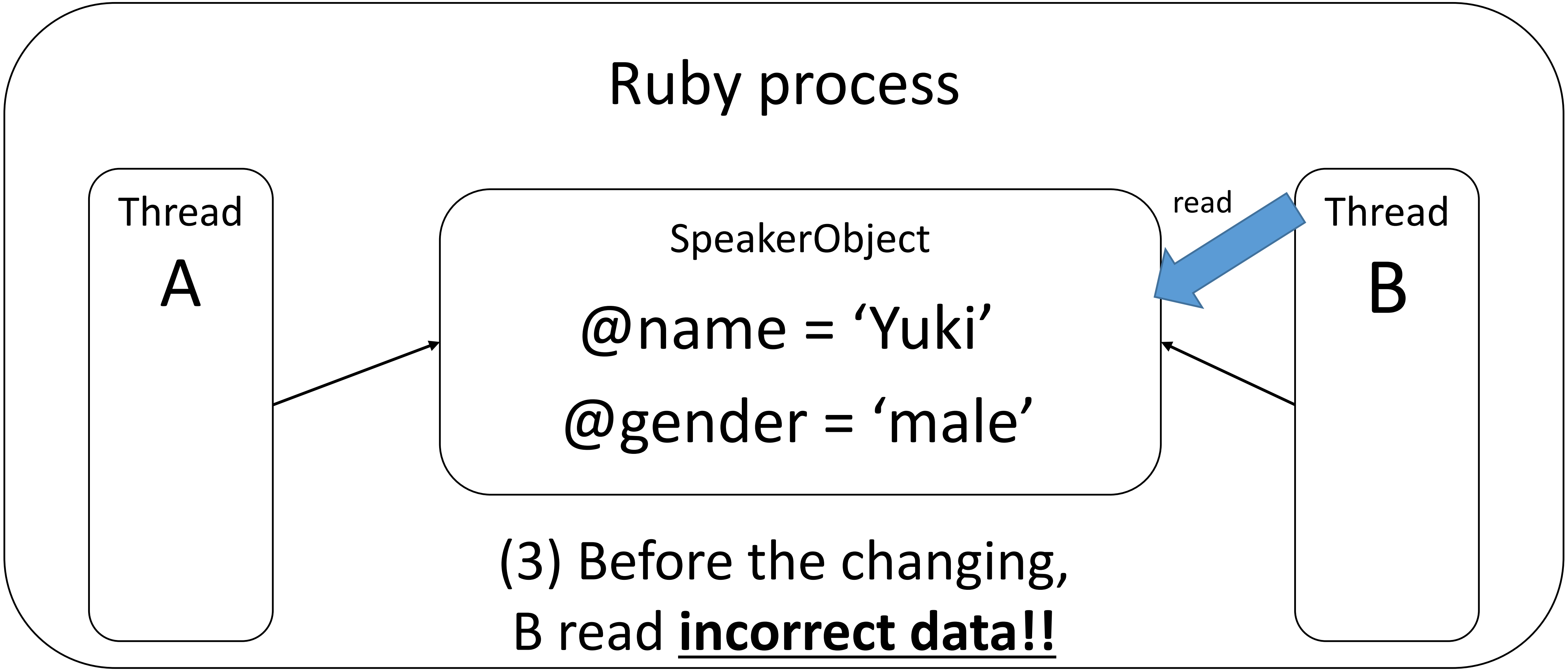
@name = ‘Yuki’

@gender = ‘male’

Thread B

(2) A changes the name to “Yuki”

Mutate shared objects
Problematic case



Ruby process

Thread

A

SpeakerObject

@name = ‘Yuki’

@gender = ‘male’

read

Thread

B

(3) Before the changing,

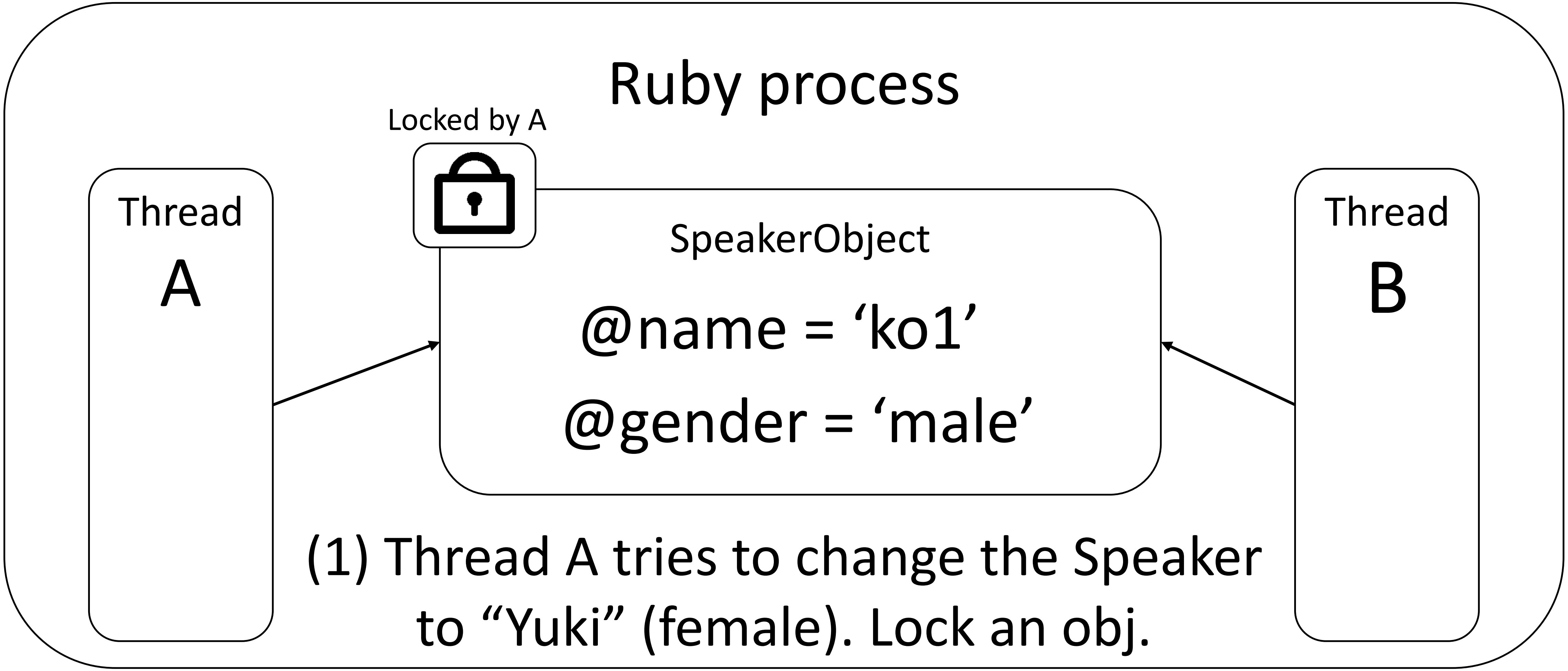
B read **incorrect data!!**

Note: Yuki should be female.

Inter-thread communication
Synchronization

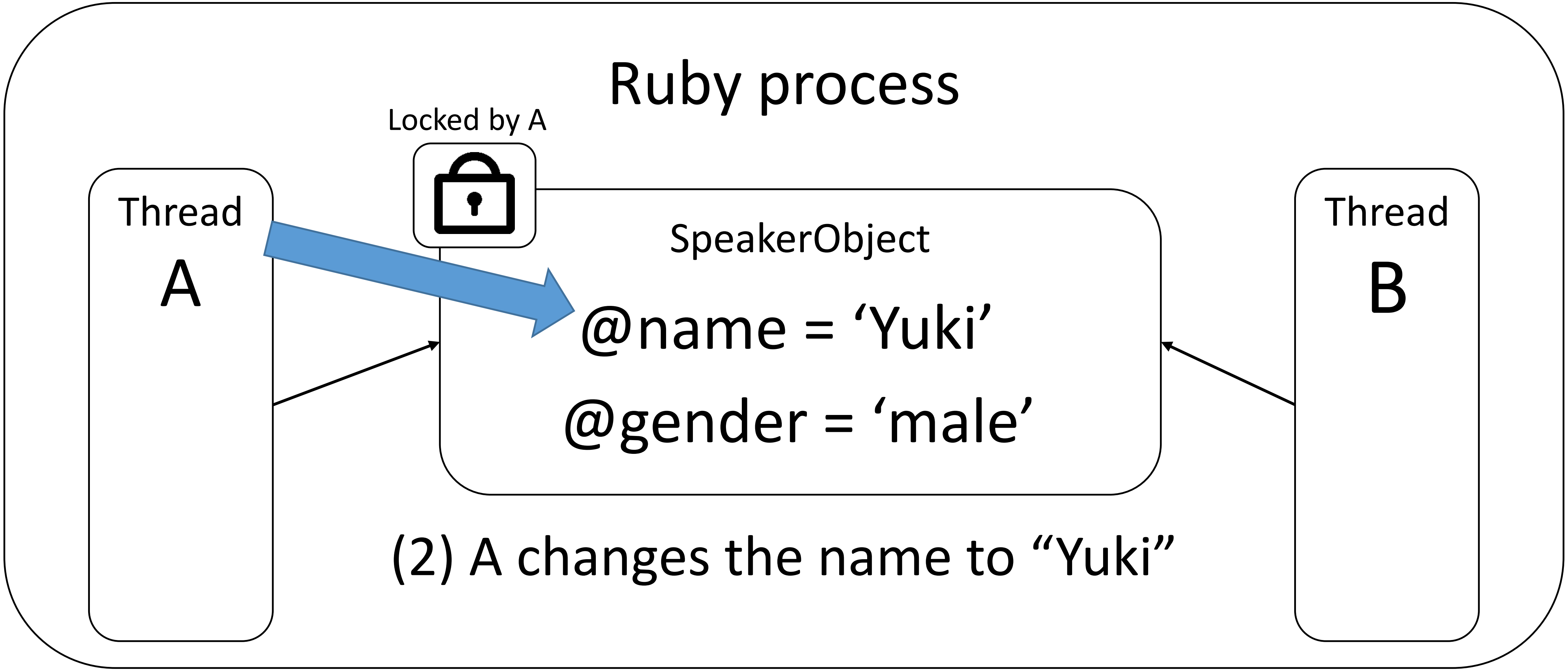
- Require synchronization for shared data
  - `Mutex`, `Queue` and so on
    - Usually `Queue` is enough
  - To prohibit simultaneous mutation
  - We need to keep consistency for each objects

Mutate shared objects
With lock



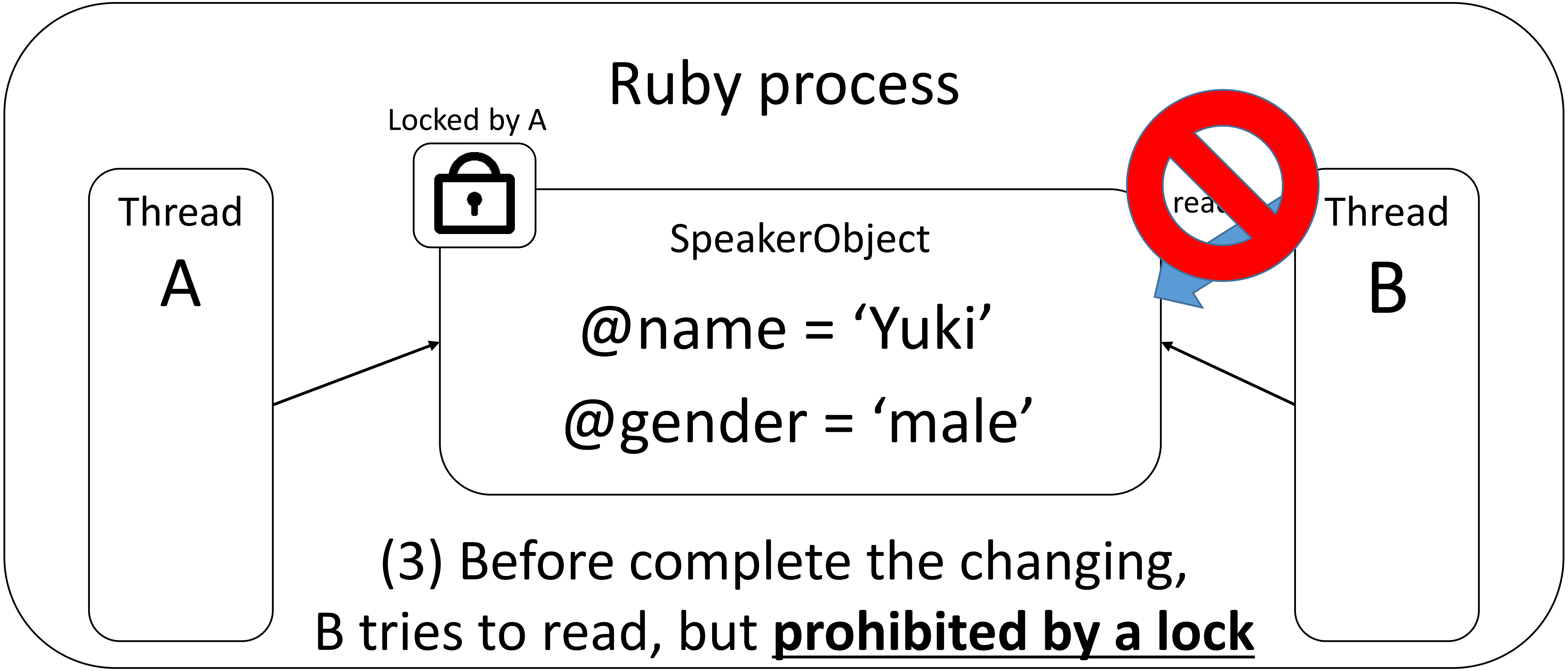
(1) Thread A tries to change the Speaker to “Yuki” (female). Lock an obj.

Mutate shared objects
With lock



(2) A changes the name to “Yuki”

Mutate shared objects With lock



@name = ‘Yuki’

@gender = ‘male’

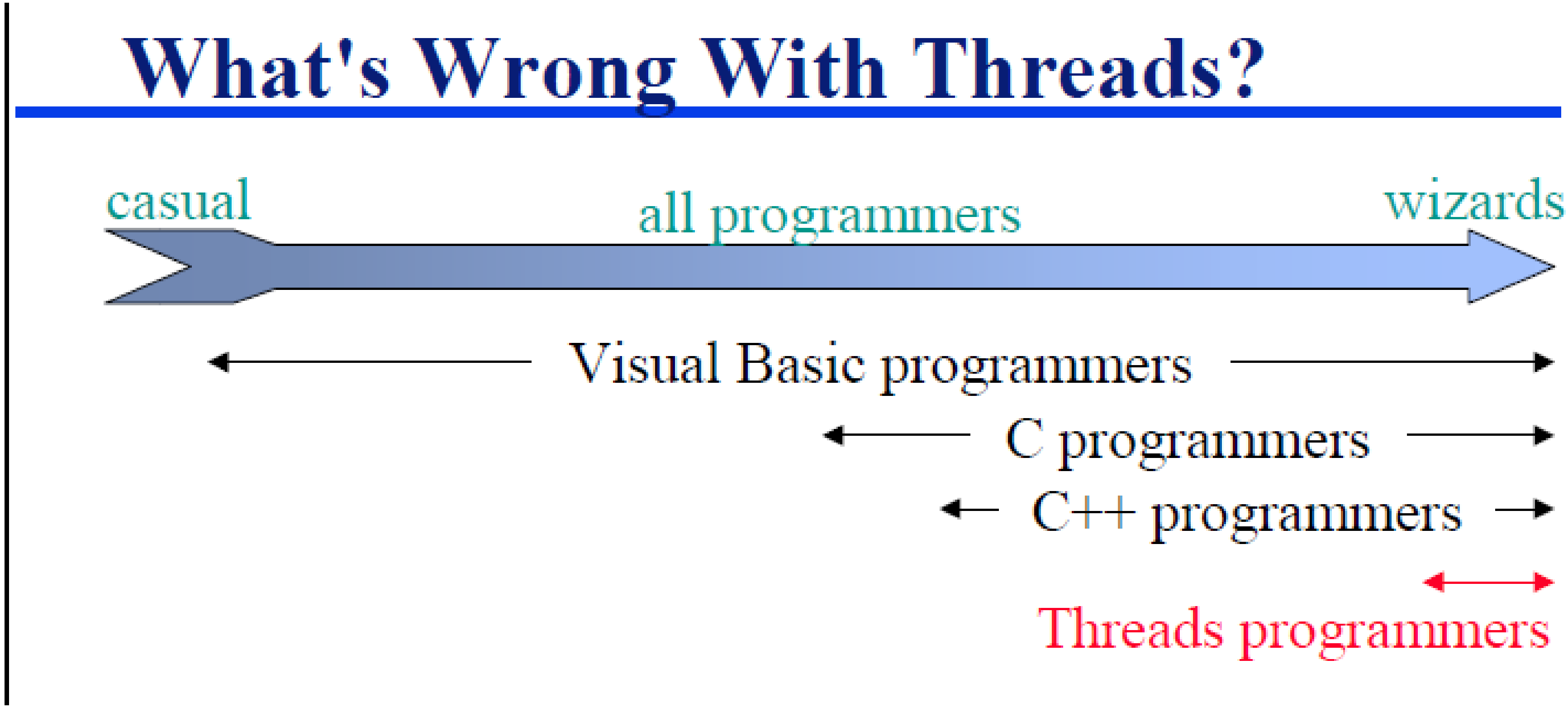
(3) Before complete the changing,
B tries to read, but **prohibited by a lock**

Thread programming
Easy to share data: Good and Bad

- Good: Easy to communicate with threads
- Bad: Too easy. Difficult to manage all of them
  - Mutation for shared data requires correct synchronization
  - Sometimes objects are shared implicitly
  - **Otherwise, it causes serious problems**

“Why Threads Are A Bad Idea (for most purposes)”

- Quoted from John Ousterhout, 1995 (about 20 years ago ☺)



What's Wrong With Threads?

casual

all programmers

wizards

Visual Basic programmers

C programmers

C++ programmers

Threads programmers

Compare Process with Thread

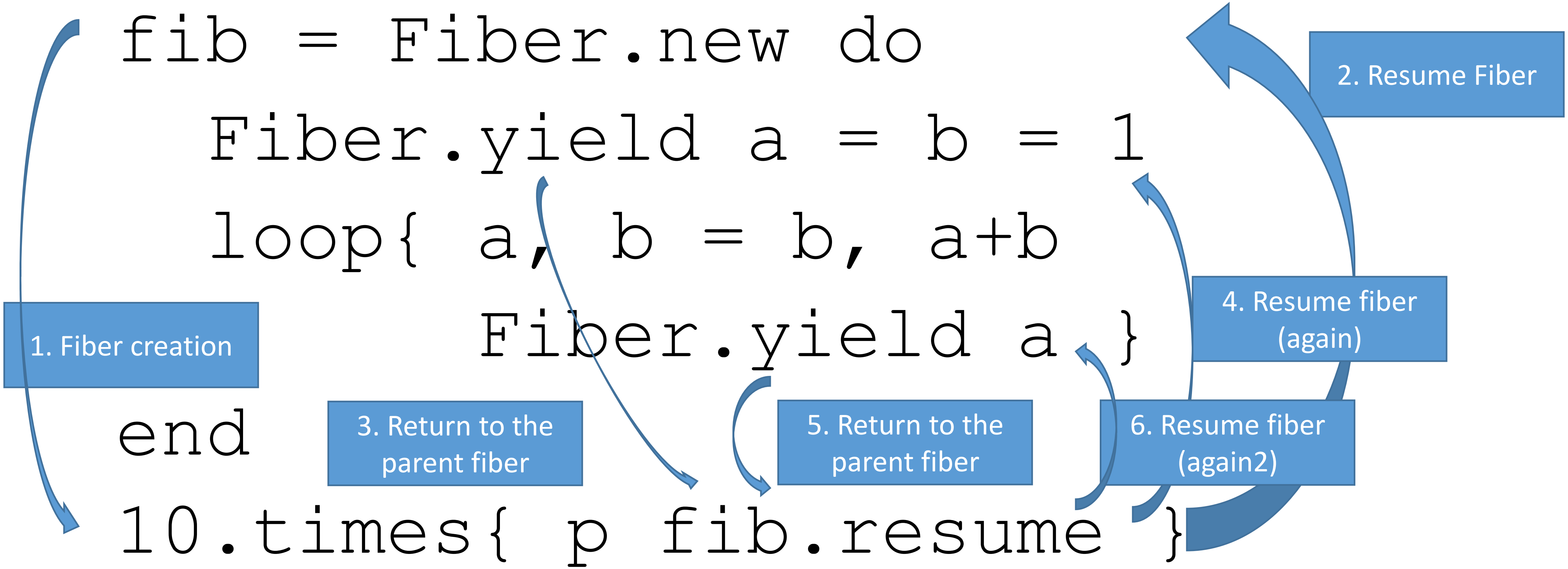
	Process	**Thread**
Available	Yes	Yes
Switch on time	Yes	Yes
Switch on I/O	Auto	Auto
Next target	Auto	Auto
Parallel run	**Yes**	No (on MRI)
Shared data	**N/A**	Everything
Communication	Hard (high-overhead)	**Easy (lightweight)**
Programming difficulty	Hard	**Difficult**
Debugging difficulty	Easy?	**Hard**

Fiber
User-defined context switching

Fiber example
Infinite generator

```
fib = Fiber.new do
  Fiber.yield a = b = 1
  loop{ a, b = b, a+b
        Fiber.yield a }
end
10.times{ p fib.resume }
```

Fiber example
Infinite generator



```
fib = Fiber.new do
  Fiber.yield a = b = 1
  loop{ a, b = b, a+b
        Fiber.yield a }
end
10.times{ p fib.resume }
```

1. Fiber creation

2. Resume Fiber

3. Return to the parent fiber

4. Resume fiber (again)

5. Return to the parent fiber

6. Resume fiber (again2)

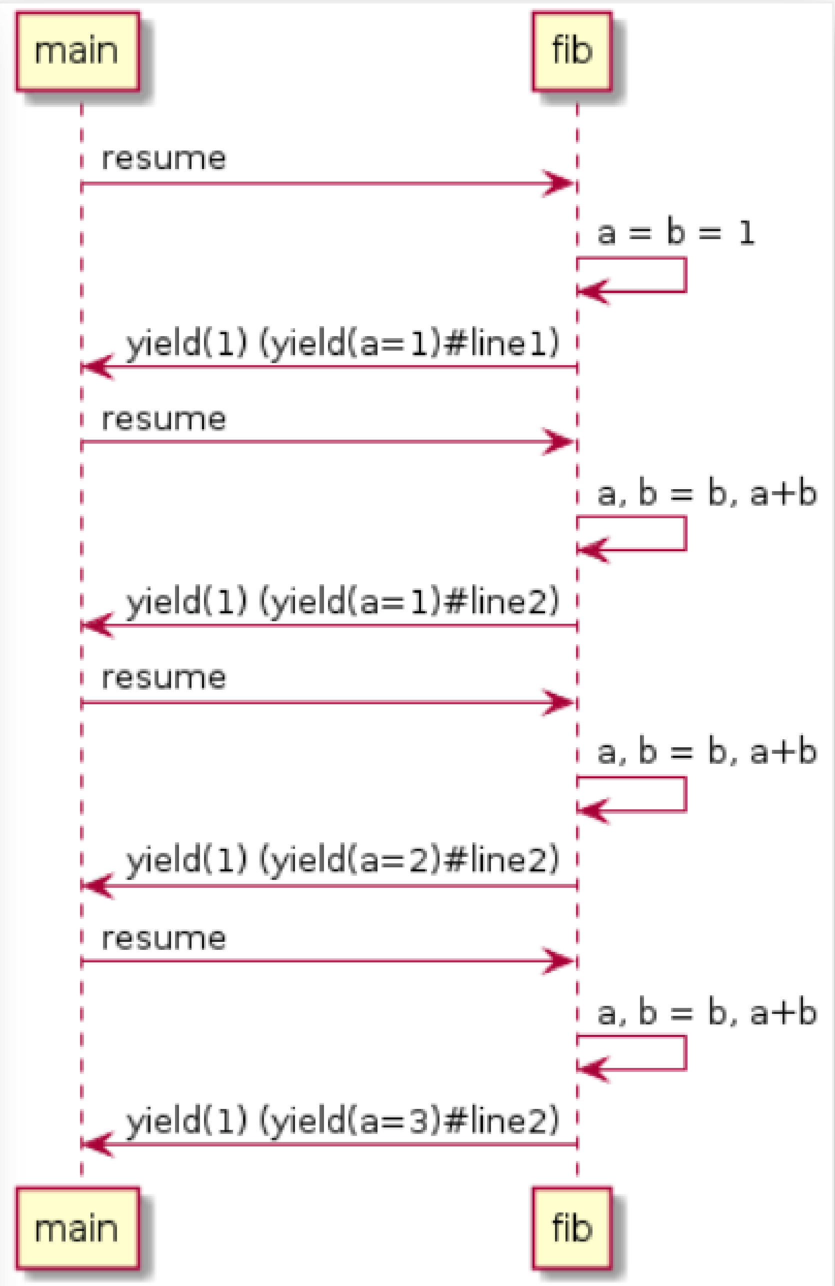
Fiber example
Infinite generator

```
fib = Fiber.new do
   Fiber.yield a = b =
   loop{ a, b = b, a+b
         Fiber.yield a
end
10.times{ p fib.resum
```

1. Fiber creation

3. Return to the parent fiber

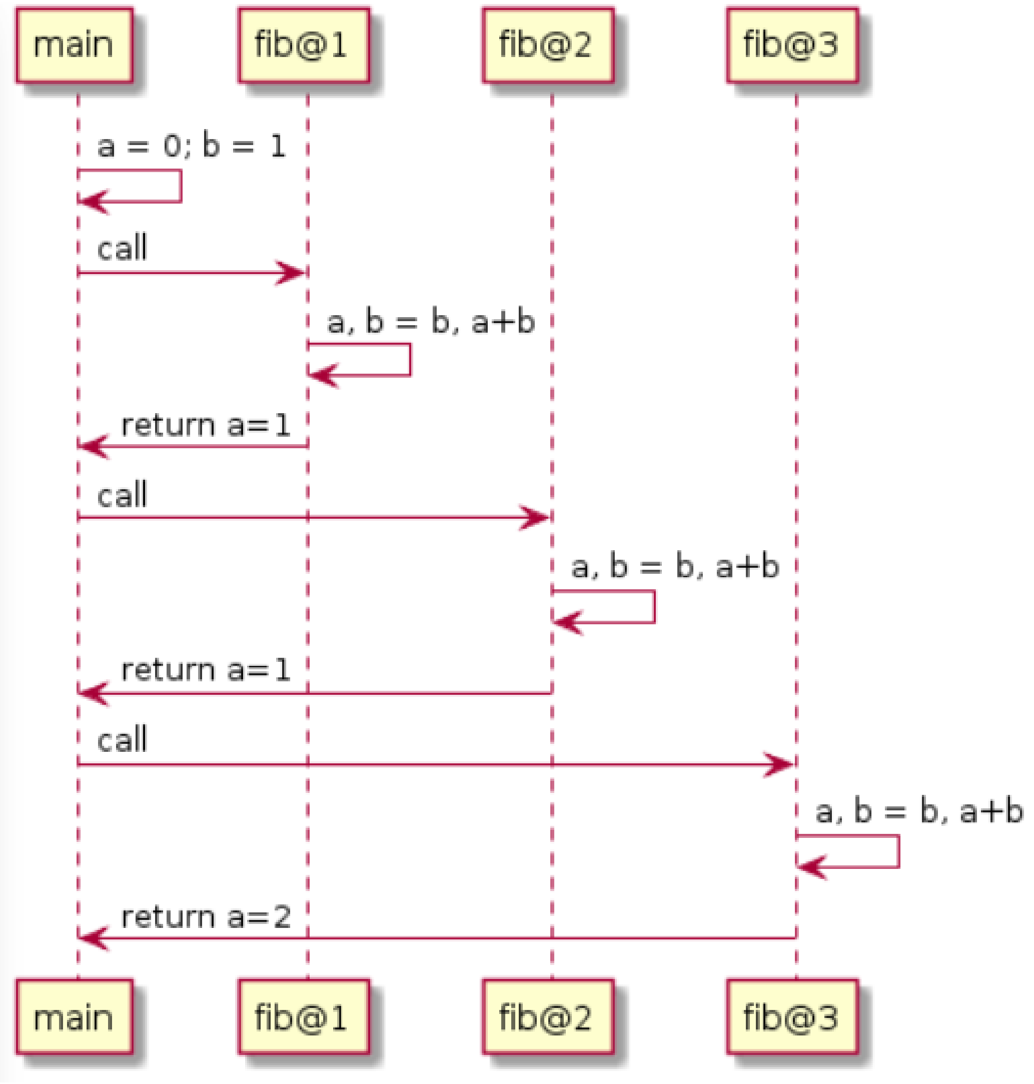
5. Return to the parent fiber


main
fib
resume
a = b = 1
yield(1) (yield(a=1)#line1)
resume
a, b = b, a+b
yield(1) (yield(a=1)#line2)
resume
a, b = b, a+b
yield(1) (yield(a=2)#line2)
resume
a, b = b, a+b
yield(1) (yield(a=3)#line2)
main
fib

Not a Proc?

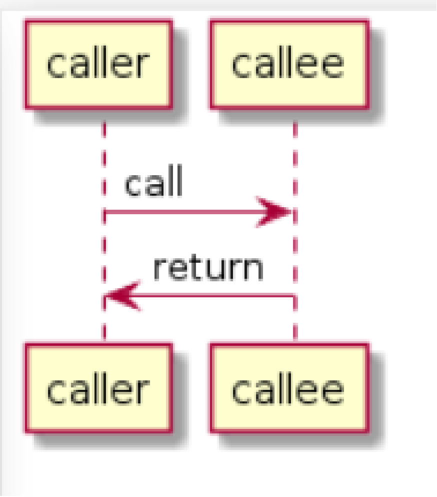
```
a = 0; b = 1
fib = Proc.new{
  a, b = b, a+b
  a
}
p fib.call #=> 1
p fib.call #=> 1
p fib.call #=> 2
p fib.call #=> 3
p fib.call #=> 5
```

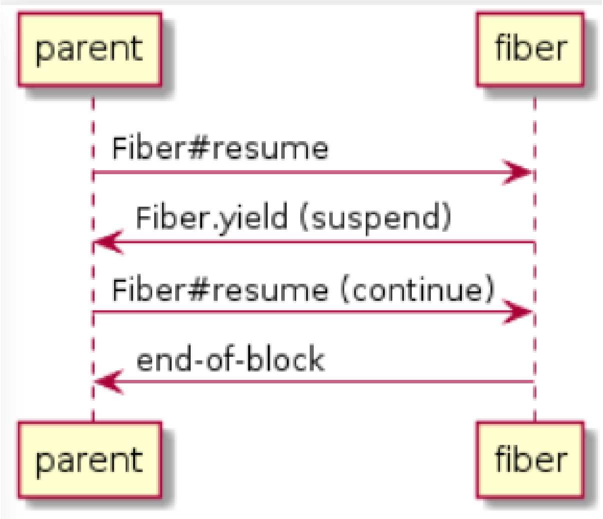
Proc can’t restart from the middle of block



Proc (method) v.s. Fiber

	Proc (method)	Fiber
Start	OK: call	OK: Fiber#resume
Parameters	OK: block (method) parameters	OK: block parameters
Return	OK: exit Proc/method	OK: exit Proc/method
Suspend	NG: N/A	**OK: Fiber.yield**
Continue	NG: N/A	**OK: Fiber#resume**





Fiber example
Inner iterator to external iterator

```
f1 = Fiber.new do
  2.times{|i| Fiber.yield i}
end

p f1.resume #=> 0
p f1.resume #=> 1
p f1.resume #=> 2 # return value of #times
p f1.resume #=> dead fiber called
                (FiberError)
```

Fiber example
Inner iterator to external iterator

```
etc_passwd_ex_iter = Fiber.new do
  open('/etc/passwd').each_line{|line|
    Fiber.yield line
  }
end
p etc_passwd_ex_iter.resume #=> 1st line
p etc_passwd_ex_iter.resume #=> 2nd line
…
```

Fiber example
Inner iterator to external iterator

```
# make Enumerator
iter = open('/etc/passwd').each_line

# Enumerator#next use Fiber implicitly
p iter.next #=> 1st line
p iter.next #=> 2nd line
```

…

Fiber example
Agent simulation

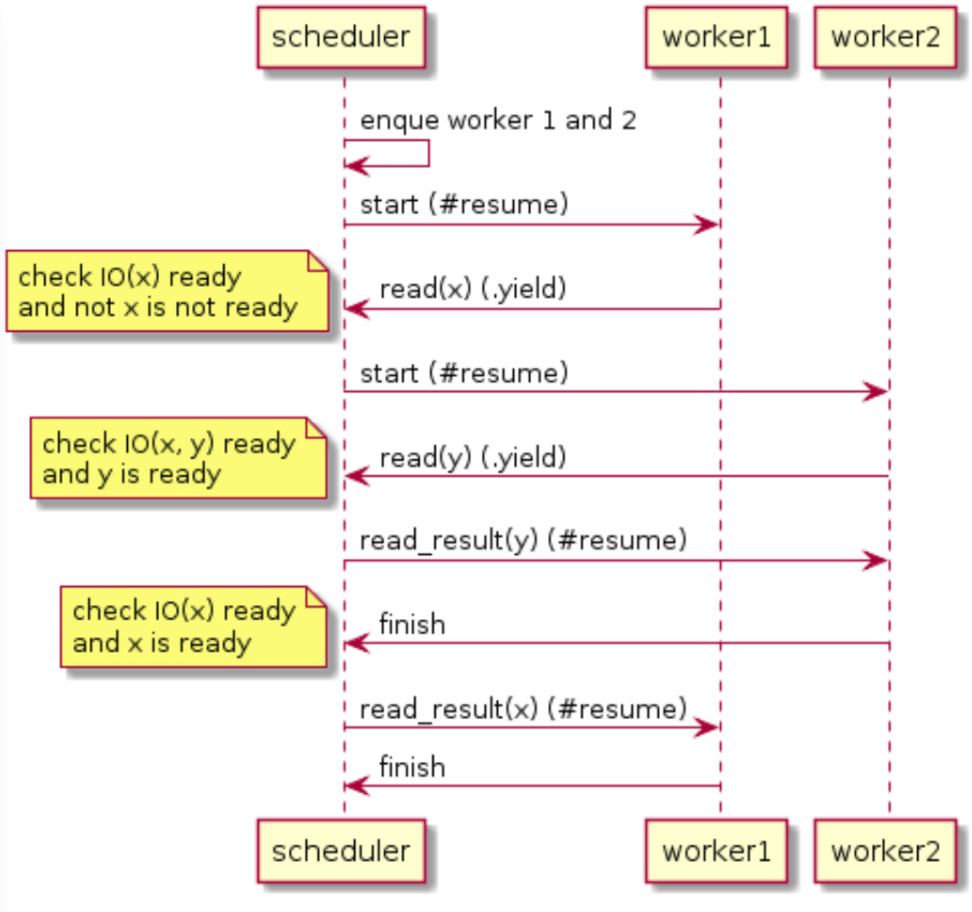
```
characters << Fiber.new{
  loop{cat.move_up; Fiber.yield}}
characters << Fiber.new{
  loop{dog.move_left; Fiber.yield}}
…
loop{cs.each{|e| e.resume}; redraw}
```

Fiber example
Agent simulation

```
characters << Fiber.new{
  # you can specify complex rule for chars
  loop{
    cow.move_up;    Fiber.yield
    cow.move_right; Fiber.yield
    cow.move_down;  Fiber.yield
    cow.move_left;  Fiber.yield
  }
}
```

Fiber example
Non-blocking IO scheduler

Wait multiple IO ops with traditional “select” or modern “poll”, “epoll” interface



Fiber
Programming difficulty

- Good
  - Synchronization for shared data is not required because of **no unexpected switching**
  - **Lightweight** than Processes and Threads
- Bad
  - We need to switch explicitly. For example, “Blocking operations” (I/O blocking, etc) stop all fibers

Comparison of existing supports

	Process	Thread	Fiber
Available	Yes	Yes	Yes
Switch on time	Yes	Yes	**No**
Switch on I/O	Auto	Auto	**No**
Next target	Auto	Auto	**Specify**
Parallel run	**Yes**	No (on MRI)	No
Shared data	**N/A**	Everything	Everything
Comm.	Hard	**Easy**	**Easy**
Programming difficulty	Hard	Difficult	**Easy**
Debugging difficulty	Easy?	Hard	**Easy**

Fiber: Brief history

- 2007/05/23 cont.c (for callcc)
- 2007/05/25 Fiber impl. [ruby-dev:30827]
- 2007/05/28 Fiber introduced into cont.c
- 2007/08/25 Fix Fiber spec
- 2017 is 10th anniversary I introduced ☺

Proposed concurrency features

Guild

Auto-Fiber

Guild

Proposed concurrency support for Ruby 3

Key idea

Problem of multi-thread programming:

Easy to share mutable objects

Idea:

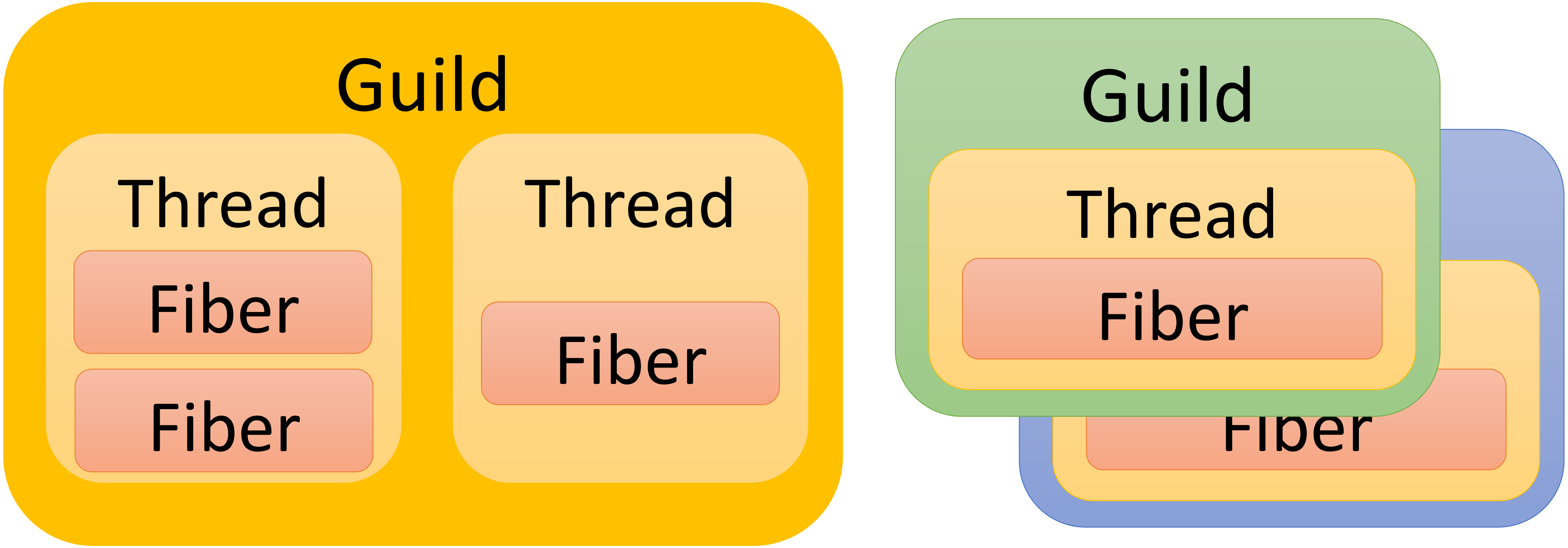
Prohibit sharing mutable objects

Our goal for Ruby 3

- We need to **keep compatibility** with Ruby 2.
- We can make **parallel program**.
- We **shouldn’t consider** locks any more.
- We **can share** objects with **copy**, but **copy operation should be fast.**
- We **should share immutable objects** if we can.
- We can **provide special objects** to share mutable objects like Clojure if we really need speed.

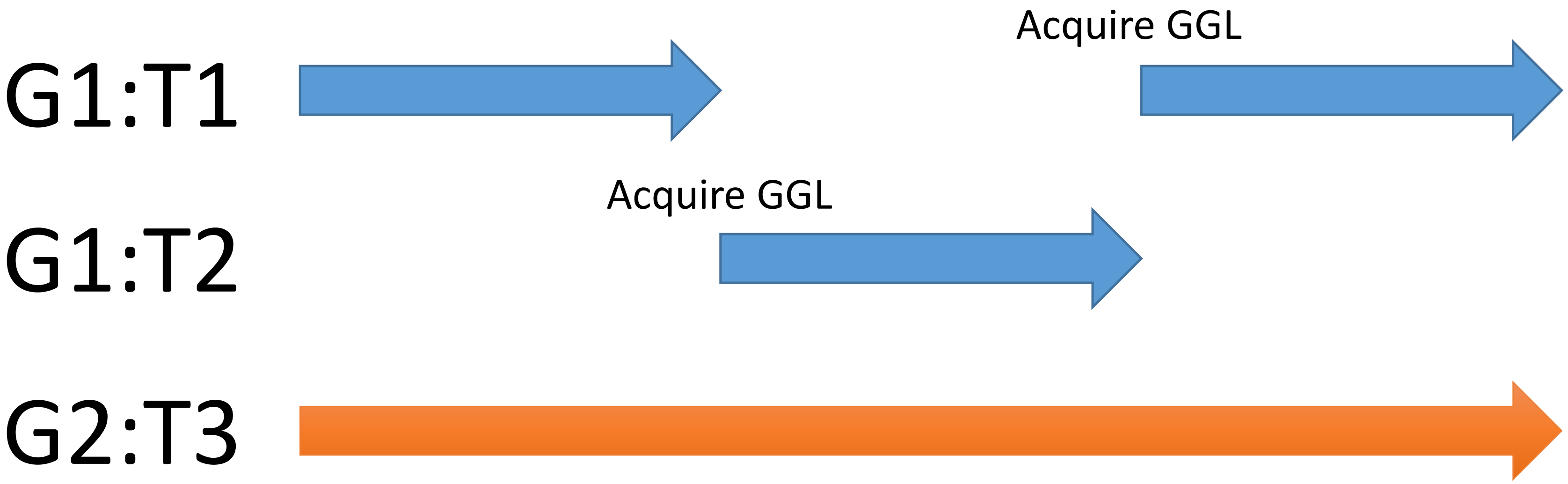
Guild: New concurrency abstraction

- Guild has at least one thread (and a thread has at least one fiber)



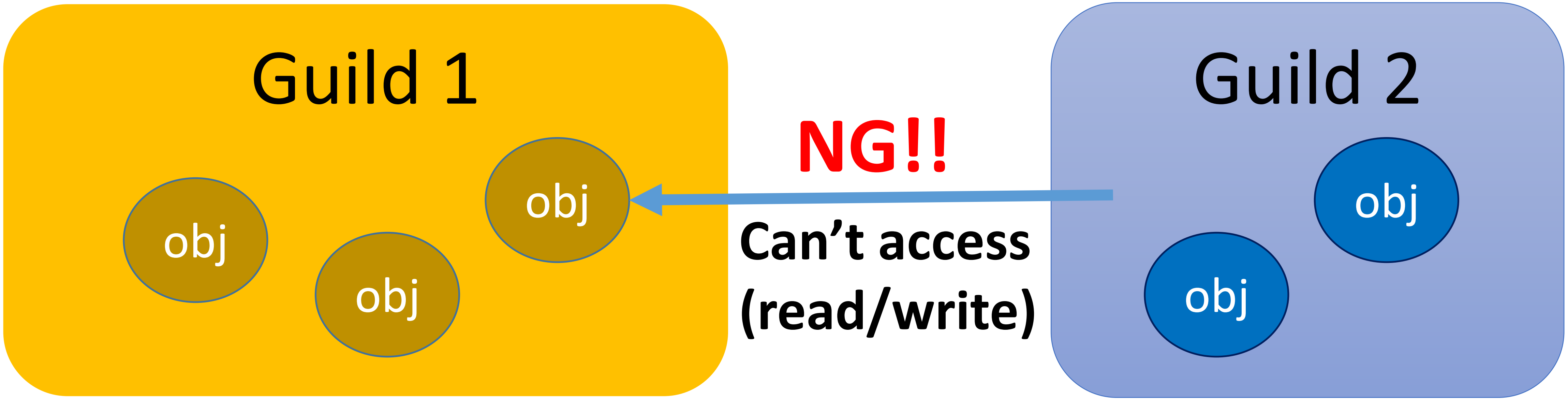
Threads in different guilds can run in Parallel

- Threads in different guilds **can run in parallel**
- Threads in a same guild **can not run in parallel** because of GVL (or GGL: Giant Guild Lock)



Important rule:
Mutable Objects have a membership

- All of mutable objects should belong to **only one Guild** exclusively
- Guild can not touch objects belong to other



Object membership

Only one guild can access mutable object

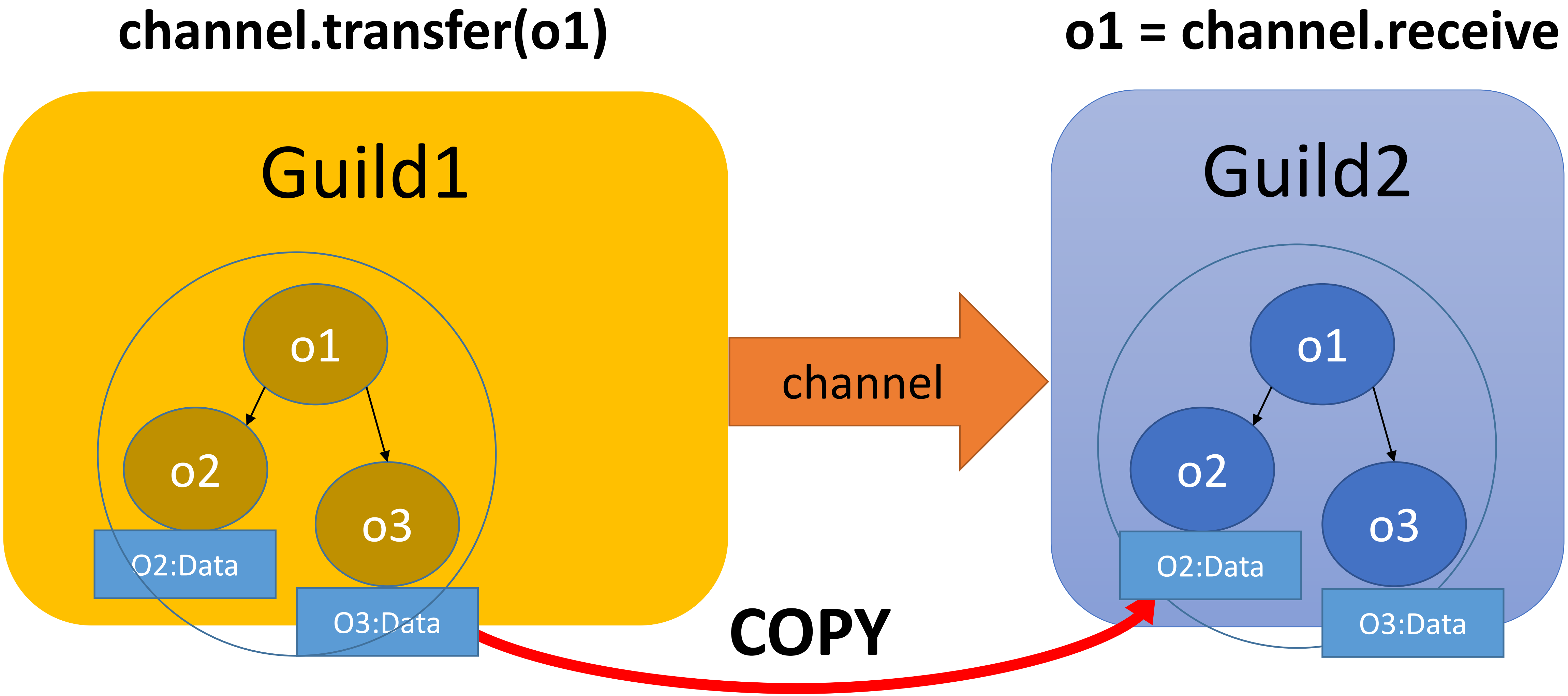
→ **We don’t need to consider locks**

(if Guild has only one thread)

Inter-guild communication

- **"Guild::Channel"** to communicate each guilds
- Two communication methods
  1. **Copy**
  2. **Move (transfer_membership)**

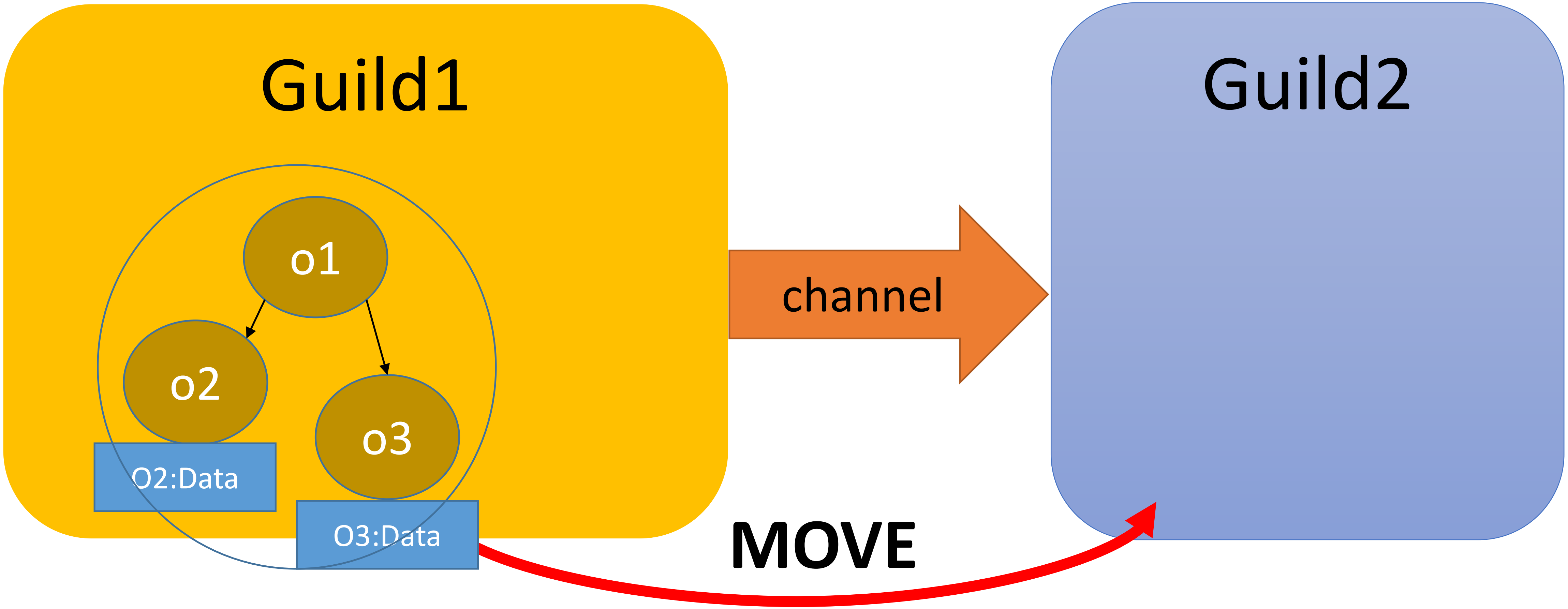
Copy using Channel



Move using Channel

channel.transfer_membership(o1)

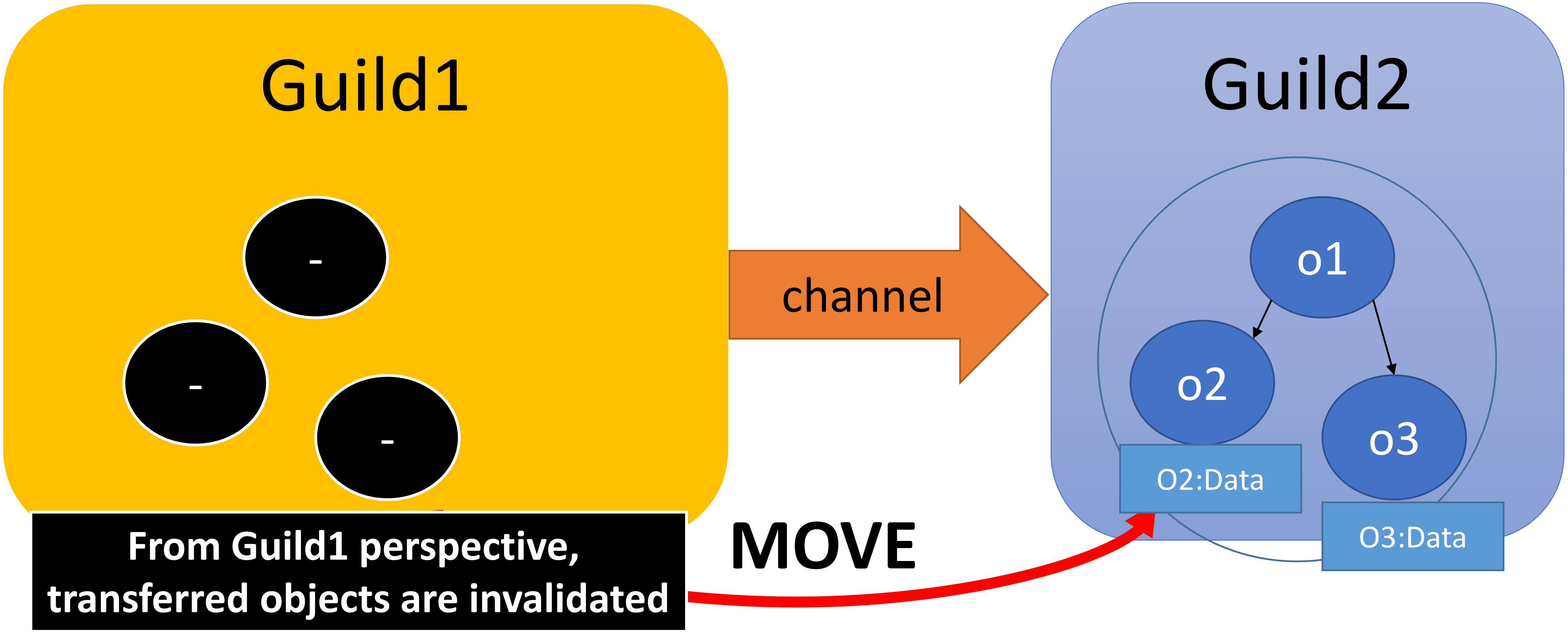
o1 = channel.receive



Move using Channel

channel.transfer_membership(o1)

o1 = channel.receive

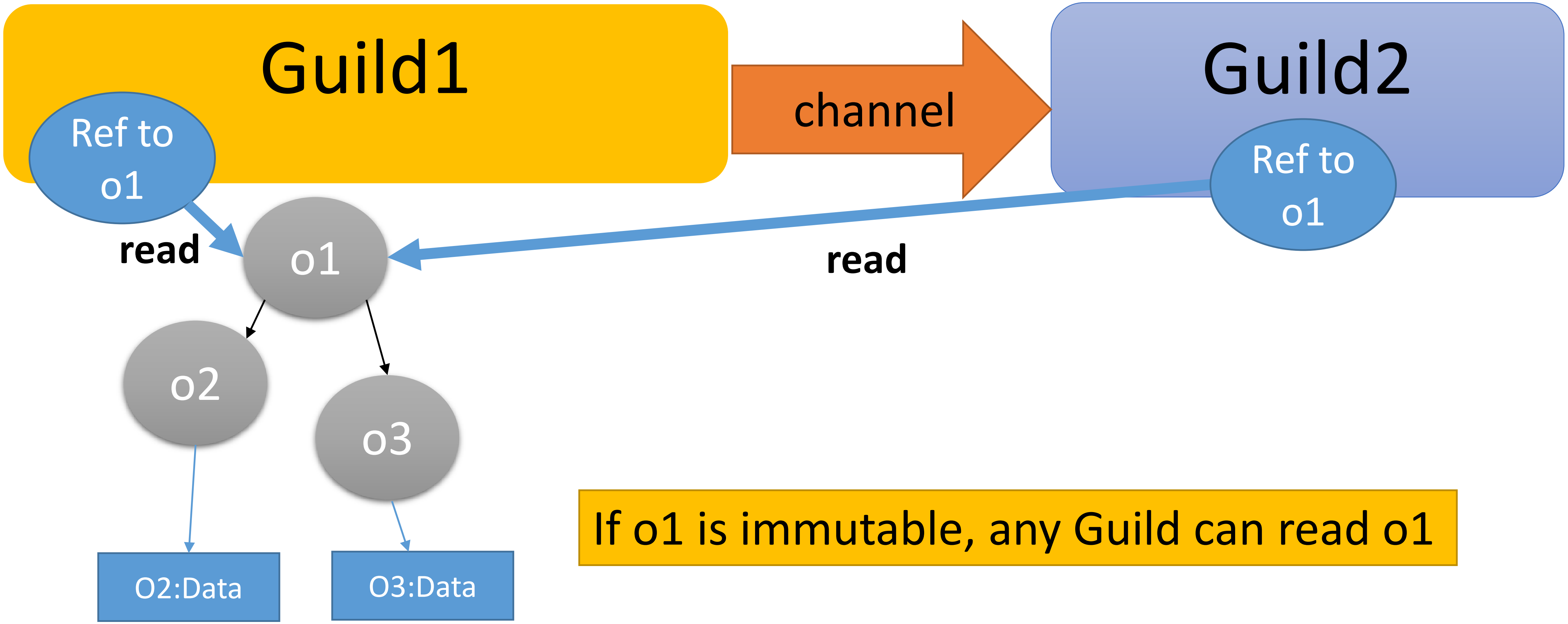


From Guild1 perspective, transferred objects are invalidated

Sharing immutable objects
We can share reference to immutable objects

channel.transfer(o1)

o1 = channel.receive

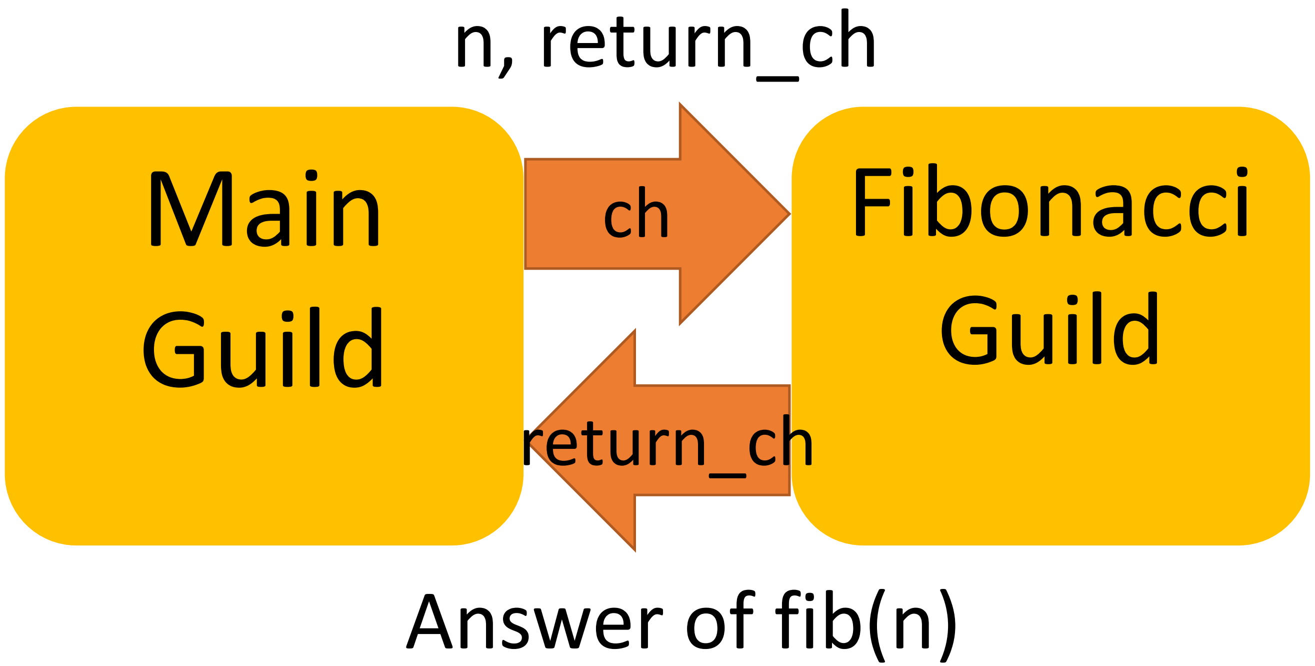


If o1 is immutable, any Guild can read o1

Use-case 1: master – worker type

```
def fib(n) ... end
g_fib = Guild.new(script: %q{
  ch = Guild.default_channel
  while n, return_ch = ch.receive
    return_ch.transfer fib(n)
  end
})

ch = Guild::Channel.new
g_fib.transfer([3, ch])
p ch.receive
```



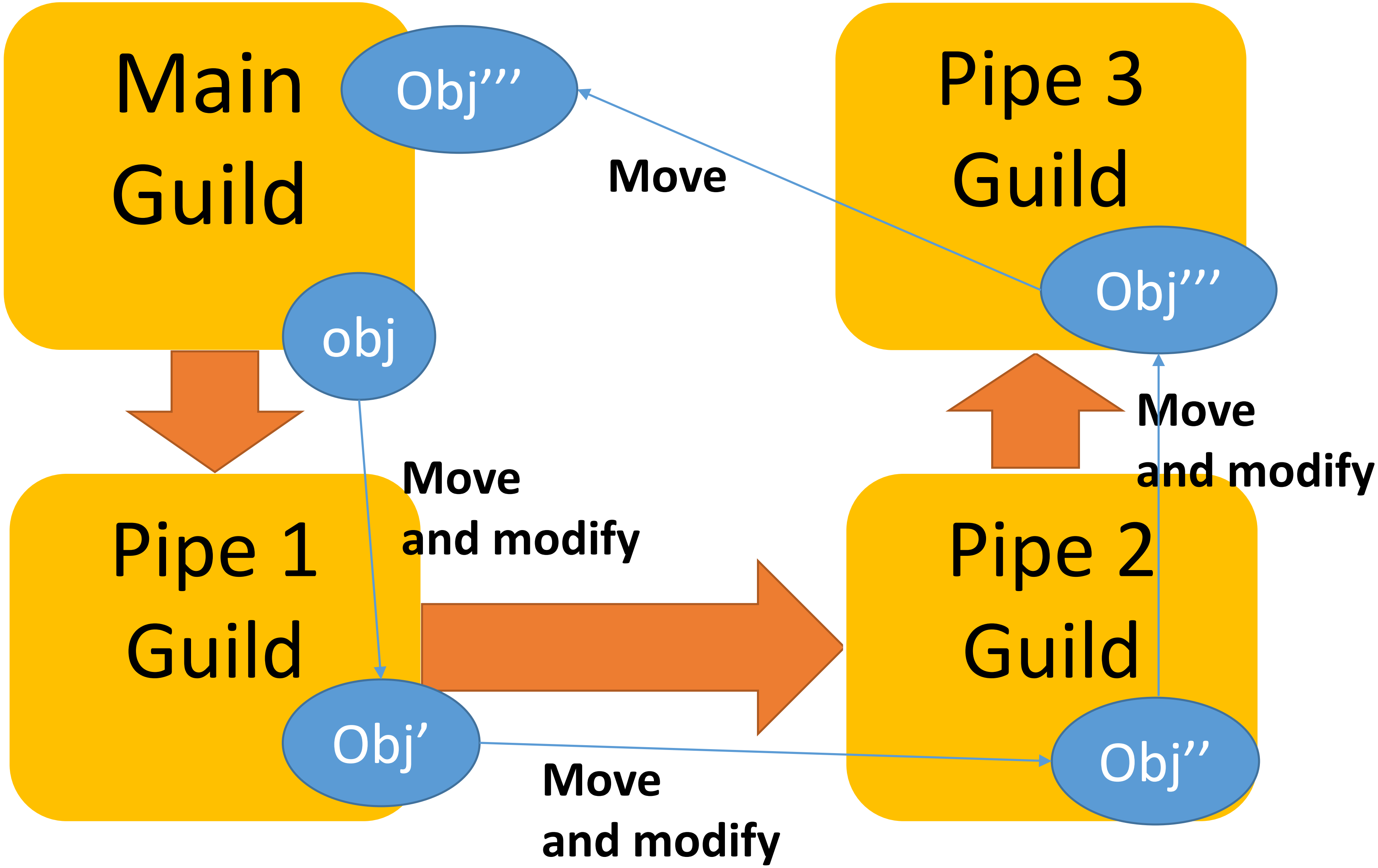
NOTE: Making other Fibonacci guilds, you can compute fib(n) in parallel

Use-case 2: pipeline

```
result_ch = Guild::Channel.new
g_pipe3 = Guild.new(script: %q{
  while obj = Guild.default_channel.receive
    obj = modify_obj3(obj)
    Guild.argv[0].transfer_membership(obj)
  end
}, argv: [result_ch])
g_pipe2 = Guild.new(script: %q{
  while obj = Guild.default_channel.receive
    obj = modify_obj2(obj)
    Guild.argv[0].transfer_membership(obj)
  end
}, argv: [g_pipe3])
g_pipe1 = Guild.new(script: %q{
  while obj = Guild.default_channel.receive
    obj = modify_obj1(obj)
    Guild.argv[0].transfer_membership(obj)
  end
}, argv: [g_pipe2])

obj = SomeClass.new

g_pipe1.transfer_membership(obj)
obj = result_ch.receive
```



Compare with Process, Guild, Thread

	Process	**Guild**	**Thread**
Available	Yes	**No**	Yes
Switch on time	Yes	Yes	Yes
Switch on I/O	Auto	Auto	Auto
Next target	Auto	Auto	Auto
Parallel run	**Yes**	**Yes**	No (on MRI)
Shared data	**N/A**	**(mostly) N/A**	Everything
Comm.	Hard	**Maybe Easy**	**Easy**
Programming difficulty	Hard	**Easy**	Difficult
Debugging difficulty	Easy?	**Maybe Easy**	Hard

Auto Fiber

Another proposed concurrency support for Ruby 3

Problem of Fiber
Requires explicit switching

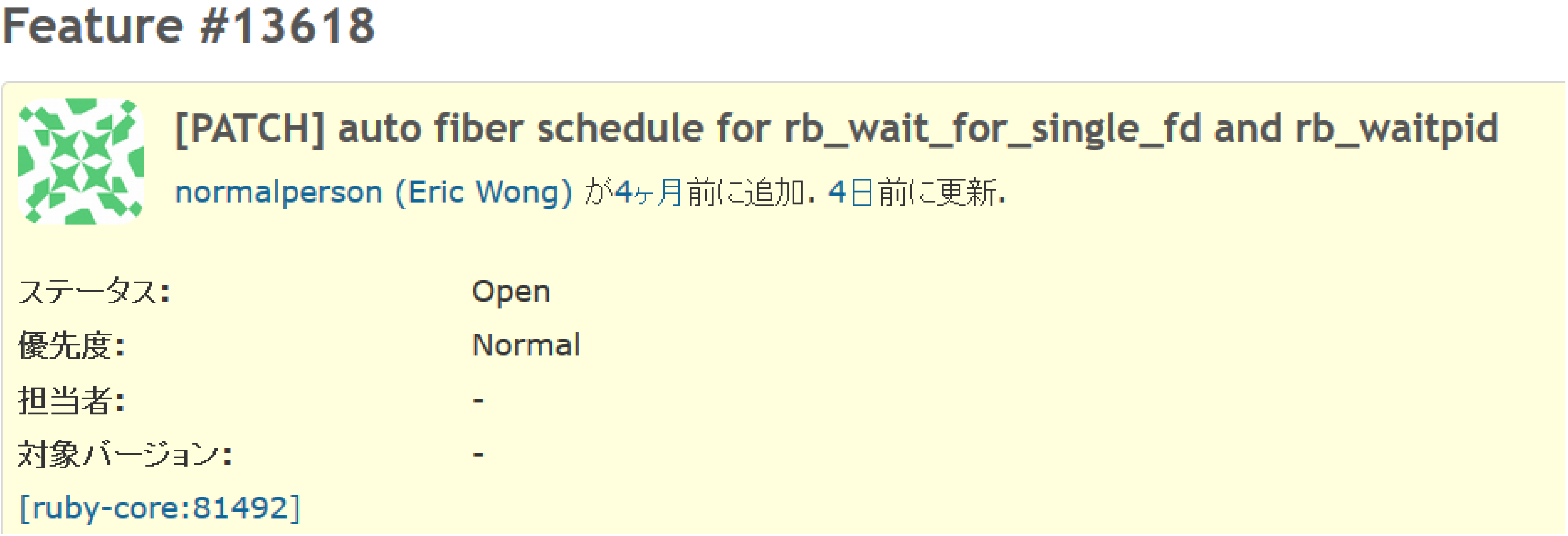
- “Fiber” enables writing scheduler by programmer

→ Programmers **need** to write own scheduler

  - We need to manage blocking operations like I/O blocking

Auto Fiber proposal

https://bugs.ruby-lang.org/issues/13618



Feature #13618

[PATCH] auto fiber schedule for rb_wait_for_single_fd and rb_waitpid

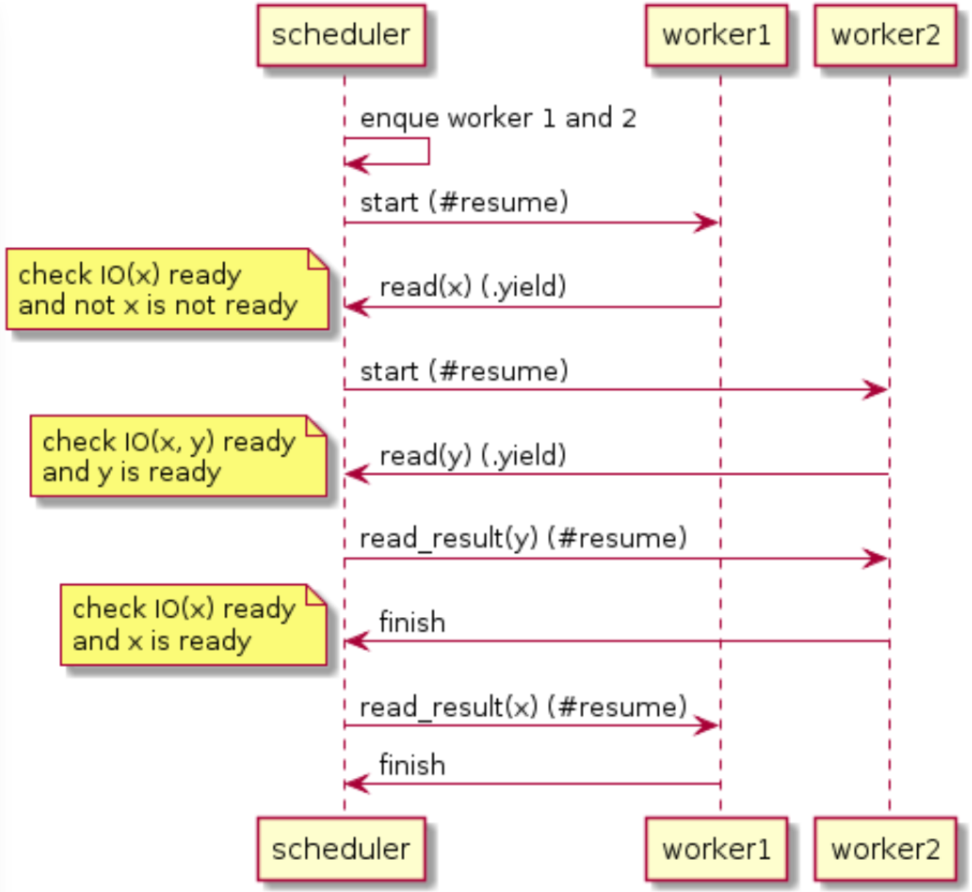
normalperson (Eric Wong) が4ヶ月前に追加. 4日前に更新.

ステータス:	Open
優先度:	Normal
担当者:	-
対象バージョン:	-

[ruby-core:81492]

Auto Fiber proposal
Automatic schedule on I/O blocking

- Support Fiber scheduler natively
  - Don’t need to return scheduler
- Switch Fibers on all blocking I/O (and other ops)
  - No need to change existing programs



Advantage and Disadvantage

- Advantage
  - Don’t need to modify existing programs
  - Lightweight as a Fiber
  - Safer than Threads (no preemption)
- Disadvantage
  - Introduce “non-deterministic” dangers same as Thread programs
    - Non atomic operations can intercept accidentally.

Change the name…?

Compare w/ Thread and (auto-)Fiber

	Thread	**Auto-Fiber**	**Fiber**
Available	Yes	**No**	Yes
Switch on time	Yes	**No**	**No**
Switch on I/O	Auto	**Auto**	No
Next target	Auto	Auto	**Specify**
Parallel run	No (on MRI)	No	No
Shared data	Everything	Everything	Everything
Comm.	**Easy**	**Easy**	**Easy**
Programming difficulty	Difficult	**Easy**	**Easy**
Debugging difficulty	Hard	Maybe hard	**Easy**

Today’s talk

- Supported features
  - Process
  - Thread
  - Fiber
- Features under consideration
  - Guild
  - Auto-Fiber

Today’s talk

	Process	**Guild**	**Thread**	**Auto-Fiber**	**Fiber**
Available	Yes	**No**	Yes	**No**	Yes
Switch on time	Yes	Yes	Yes	**No**	**No**
Switch on I/O	Auto	Auto	Auto	**Auto**	No
Next target	Auto	Auto	Auto	Auto	**Specify**
Parallel run	**Yes**	**Yes**	No (on MRI)	No	No
Shared data	**N/A**	**(mostly) N/A**	Everything	Everything	Everything
Comm.	Hard	**Maybe Easy**	**Easy**	**Easy**	**Easy**
Programming difficulty	Hard	**Easy**	Difficult	**Easy**	**Easy**
Debugging difficulty	Easy?	**Maybe Easy**	Hard	Maybe hard	**Easy**

References

- Fiber: RubyKaigi 2017 http://rubykaigi.org/2017/presentations/ko1.html
- Guild: RubyConf 2016 https://www.youtube.com/watch?v=mjzmUUQWqco
- Auto-fiber: Feature #13618 https://bugs.ruby-lang.org/issues/13618

Thank you for your attention

Koichi Sasada

<ko1@cookpad.com>

cookpad