Ruby's Concurrency Management: Now and Future

Koichi Sasada
ko1@cookpad.com
Today’s talk

• Supported features
  • Process
  • Thread
  • Fiber

• Features under consideration
  • Guild
  • Auto-Fiber
## Today’s talk

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

So many gems
such as Rails, pry, thin, ... and so on.

RubyGems/Bundler

Ruby interpreter

i gigantum umeris insidentes
Standing on the shoulders of giants
Normal MRI developer’s view

Ruby script

Parse

AST
Abstract Syntax Tree

Compile (codegen)

Ruby Bytecode

Interpret on RubyVM

Bundled Libraries

Gem Libraries

Embedded classes and methods

Threading

Evaluator

Object management Garbage collector
So many gems
such as Rails, pry, thin, ... and so on.
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
Concurrenty

“In computer science, concurrency is the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units. 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
Task A
Task B

Parallel (and concurrent) execution
Physical concept
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 receiving  post process

Download B  wait for receiving  post process

Download C  wait for receiving  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_f_name, 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 receiving  post process

Download B  wait for receiving  post process

Download C

... and download C after that
Downloader example
Without concurrency support

```ruby
# 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

Ruby process

- obj
  - Serialize
  - string

pipe

file

string

Ruby process

- obj
  - Deserialize
  - string
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

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

Ruby process

Thread A

write

SpeakerObject

@name = ‘Yuki’
@gender = ‘male’

(2) A changes the name to “Yuki”
Mutate shared objects
Lucky case

Ruby process

Thread A

write

SpeakerObject
@name = 'Yuki'
@gender = 'female'

(3) A changes the gender to “female”

Thread B
Mutate shared objects
Lucky case

Ruby process

Thread
A

SpeakerObject
@name = ‘Yuki’
@gender = ‘female’

(4) Complete.
A and B can read correct speaker.

Thread
B
Mutate shared objects
Problematic case

Thread A

Thread B

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'

(2) A changes the name to “Yuki”
Mutate shared objects
Problematic case

Ruby process

Thread A

SpeakerObject

@name = 'Yuki'
@gender = 'male'

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

Thread A

Locked by A

SpeakerObject

@name = ‘ko1’
@gender = ‘male’

Thread B

(1) Thread A tries to change the Speaker to “Yuki” (female). Lock an obj.
Mutate shared objects
With lock

Thread A

Locked by A

Ruby process

SpeakerObject

@name = ‘Yuki’
@gender = ‘male’

(2) A changes the name to “Yuki”
Mutate shared objects
With lock

(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 😊)
## Compare Process with Thread

<table>
<thead>
<tr>
<th></th>
<th>Process</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Switch on time</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Switch on I/O</td>
<td>Auto</td>
<td>Auto</td>
</tr>
<tr>
<td>Next target</td>
<td>Auto</td>
<td>Auto</td>
</tr>
<tr>
<td>Parallel run</td>
<td>Yes</td>
<td>No (on MRI)</td>
</tr>
<tr>
<td>Shared data</td>
<td>N/A</td>
<td>Everything</td>
</tr>
<tr>
<td>Communication</td>
<td>Hard (high-overhead)</td>
<td>Easy (lightweight)</td>
</tr>
<tr>
<td>Programming difficulty</td>
<td>Hard</td>
<td>Difficult</td>
</tr>
<tr>
<td>Debugging difficulty</td>
<td>Easy?</td>
<td>Hard</td>
</tr>
</tbody>
</table>
Fiber
User-defined context switching
Fiber example
Infinite generator

```ruby
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

```ruby
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

```ruby
fib = Fiber.new do
  a = b = 1
  loop { a, b = b, a+b
    Fiber.yield
  } 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)
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

<table>
<thead>
<tr>
<th></th>
<th>Proc (method)</th>
<th>Fiber</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start</strong></td>
<td>OK: call</td>
<td>OK: Fiber#resume</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>OK: block (method) parameters</td>
<td>OK: block parameters</td>
</tr>
<tr>
<td><strong>Suspend</strong></td>
<td>NG: N/A</td>
<td>OK: Fiber.yield</td>
</tr>
<tr>
<td><strong>Continue</strong></td>
<td>NG: N/A</td>
<td>OK: Fiber#resume</td>
</tr>
</tbody>
</table>

![Diagram](image.png)

![Diagram](image.png)
Fiber example

Inner iterator to external iterator

```ruby
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 #=> 1\text{st} line
p etc_passwd_ex_iter.resume #=> 2\text{nd} line
...```
Fiber example
Inner iterator to external iterator

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

# Enumerator#next use Fiber implicitly
p iter.next #=> 1\text{st} line
p iter.next #=> 2\text{nd} 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
    }
}
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

<table>
<thead>
<tr>
<th></th>
<th>Process</th>
<th>Thread</th>
<th>Fiber</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Switch on time</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Switch on I/O</td>
<td>Auto</td>
<td>Auto</td>
<td>No</td>
</tr>
<tr>
<td>Next target</td>
<td>Auto</td>
<td>Auto</td>
<td>Specify</td>
</tr>
<tr>
<td>Parallel run</td>
<td>Yes</td>
<td>No (on MRI)</td>
<td>No</td>
</tr>
<tr>
<td>Shared data</td>
<td>N/A</td>
<td>Everything</td>
<td>Everything</td>
</tr>
<tr>
<td>Comm.</td>
<td>Hard</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Programming difficulty</td>
<td>Hard</td>
<td>Difficult</td>
<td>Easy</td>
</tr>
<tr>
<td>Debugging difficulty</td>
<td>Easy?</td>
<td>Hard</td>
<td>Easy</td>
</tr>
</tbody>
</table>
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)

G1:T1

G1:T2

G2:T3

Acquire GGL
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

![Diagram showing two guilds with objects and restrictions on access](image-url)
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

channel.transfer(o1)

Guild1

Guild2

o1 = channel.receive

COPY

channel

O2:Data

O3:Data

O1

O2

O3

O2:Data

O3:Data
Move using Channel

```
channel.transfer_membership(o1)
```

```
o1 = channel.receive
```

**Guild1**
- o1
- o2
- o3

**Guild2**

O2:Data
O3:Data

MOVE
Move using Channel

channel.transfer_membership(o1)

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
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
ch.transfer([3, ch])
p ch.receive

NOTE: Making other Fibonacci guilds, you can compute fib(n) in parallel
Use-case 2: pipeline

```ruby
result_ch = Guild::Channel.new

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])

pipe2 = Guild.new(script: "%q{
  while obj = Guild.default_channel.receive
    obj = modify_obj2(obj)
    Guild.argv[0].transfer_membership(obj)
  end
}, argv: [pipe3])

pipe1 = Guild.new(script: "%q{
  while obj = Guild.default_channel.receive
    obj = modify_obj1(obj)
    Guild.argv[0].transfer_membership(obj)
  end
}, argv: [pipe2])

obj = SomeClass.new

obj = pipe1.transfer_membership(obj)
obj = result_ch.receive
```
## Compare with Process, Guild, Thread

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

<table>
<thead>
<tr>
<th></th>
<th>Thread</th>
<th>Auto-Fiber</th>
<th>Fiber</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Switch on time</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Switch on I/O</td>
<td>Auto</td>
<td>Auto</td>
<td>No</td>
</tr>
<tr>
<td>Next target</td>
<td>Auto</td>
<td>Auto</td>
<td>Specify</td>
</tr>
<tr>
<td>Parallel run</td>
<td>No (on MRI)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Shared data</td>
<td>Everything</td>
<td>Everything</td>
<td>Everything</td>
</tr>
<tr>
<td>Comm.</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Programming difficulty</td>
<td>Difficult</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Debugging difficulty</td>
<td>Hard</td>
<td>Maybe hard</td>
<td>Easy</td>
</tr>
</tbody>
</table>
Today’s talk

• Supported features
  • Process
  • Thread
  • Fiber

• Features under consideration
  • Guild
  • Auto-Fiber
## Today’s talk

<table>
<thead>
<tr>
<th>Process</th>
<th>Guild</th>
<th>Thread</th>
<th>Auto-Fiber</th>
<th>Fiber</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Switch on time</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Switch on I/O</td>
<td>Auto</td>
<td>Auto</td>
<td>Auto</td>
<td>Auto</td>
</tr>
<tr>
<td>Next target</td>
<td>Auto</td>
<td>Auto</td>
<td>Auto</td>
<td>Auto</td>
</tr>
<tr>
<td>Parallel run</td>
<td>Yes</td>
<td>Yes</td>
<td>No (on MRI)</td>
<td>No</td>
</tr>
<tr>
<td>Shared data</td>
<td>N/A</td>
<td>(mostly) N/A</td>
<td>Everything</td>
<td>Everything</td>
</tr>
<tr>
<td>Comm.</td>
<td>Hard</td>
<td>Maybe Easy</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Programming difficulty</td>
<td>Hard</td>
<td>Easy</td>
<td>Difficult</td>
<td>Easy</td>
</tr>
<tr>
<td>Debugging difficulty</td>
<td>Easy?</td>
<td>Maybe Easy</td>
<td>Hard</td>
<td>Maybe hard</td>
</tr>
</tbody>
</table>
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>