Rubyによる
並行並列プログラミング

笹田 耕一
<ko1@st.inc>

STORES

今日のトピック

- Rubyでの並行・並列処理のための仕組み
  - プロセス、Ractor、スレッド、Fiber scheduler, Fiber
  - Ruby 3.3 から導入される M:N スケジューラでいろいろ軽量に
- 共有データを複数スレッドで読み書きするなら同期は大変
  - 大変なので、なるべくやらないのがよい
  - 適切なロックが必要
  - Transactional Memory を用いた Ractor::TVar という別解もあるよ

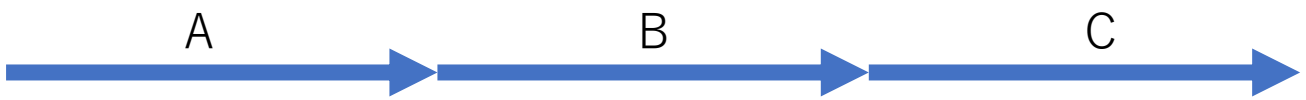
自己紹介：笹田耕一

- 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~)
  - …
- Ruby Association Director (2012~)

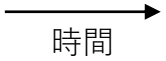
STORES

プログラミング

- コンピューターに仕事（タスク）を処理してもらうこと

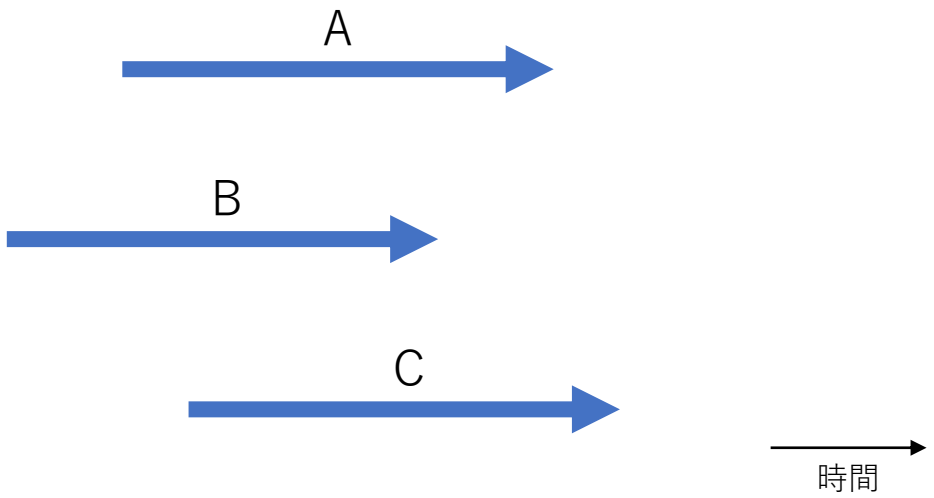



```
# Ruby で書くと
doA()
doB()
doC()
```



並列処理（Parallel processing）

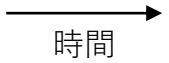
- 複数の仕事を同時にやってもらうこと
  - A, B, C がそれぞれ関係ない仕事の場合、同時にやってもらえる
    - たとえば B は A の結果が必要である場合、A の後にやらないといけないから
  - 同時にやってくれると**速く終わる**



並行処理（Concurrent processing）

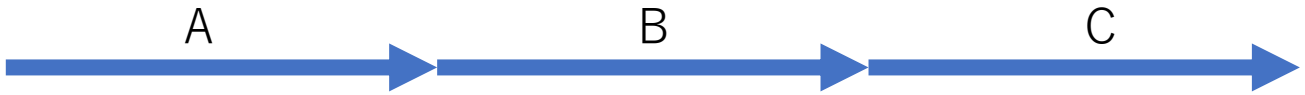
- 複数の仕事をやってもらうこと
  - 同時に**やってもいい**し、同時じゃなくてもいい
    - 並列処理は並行処理の一種
  - まっていれば複数の仕事が**いつか終わる**
    - 特に指定がない場合、順番はわからない
    - I/O などで待ちになると別の処理を実行してもいい

C の処理が一時中断して B を先に終わらせた例



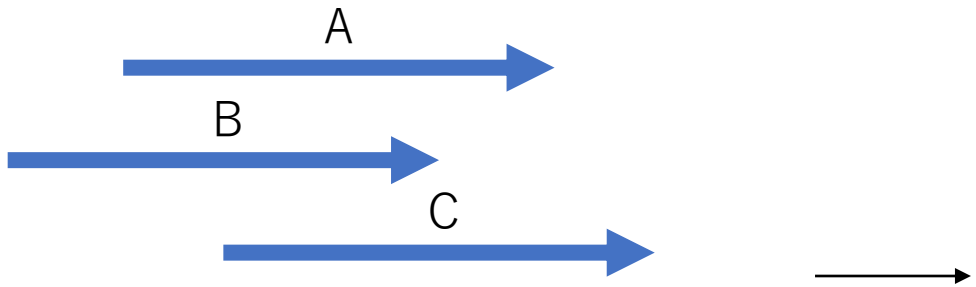
逐次・並行・並列プログラミング

- 逐次プログラミング
  - “ふつう” にかいたもの
  - 順番に実行→順番にしかできない



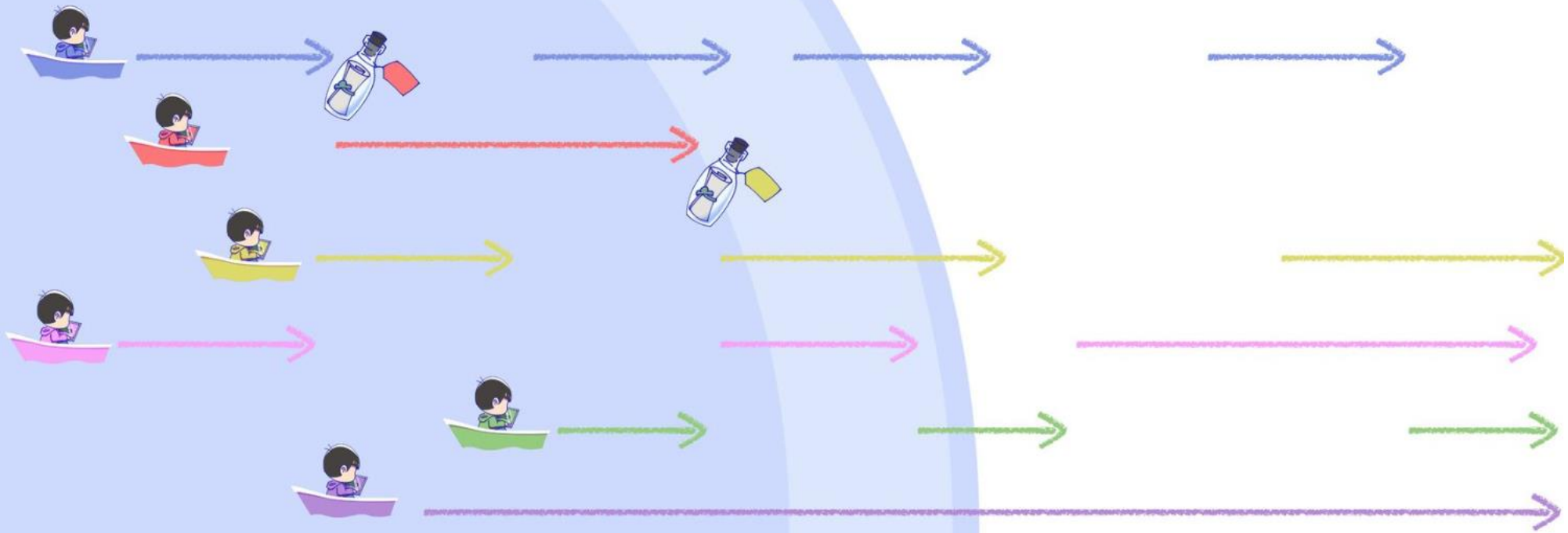
- 並行プログラミング
  - 独立した複数の処理をそれぞれ独立に書くと書きやすい
  - システムが**いい感じに**実行
    - C が中断したら B をやる、みたいなのは逐次では書きづらい

- 並列プログラミング
  - 独立した複数の処理を同時に実行
  - プログラムを高速に実行したい



Complex embedded systems

- Lots of processes, and explicit and implicit IPC

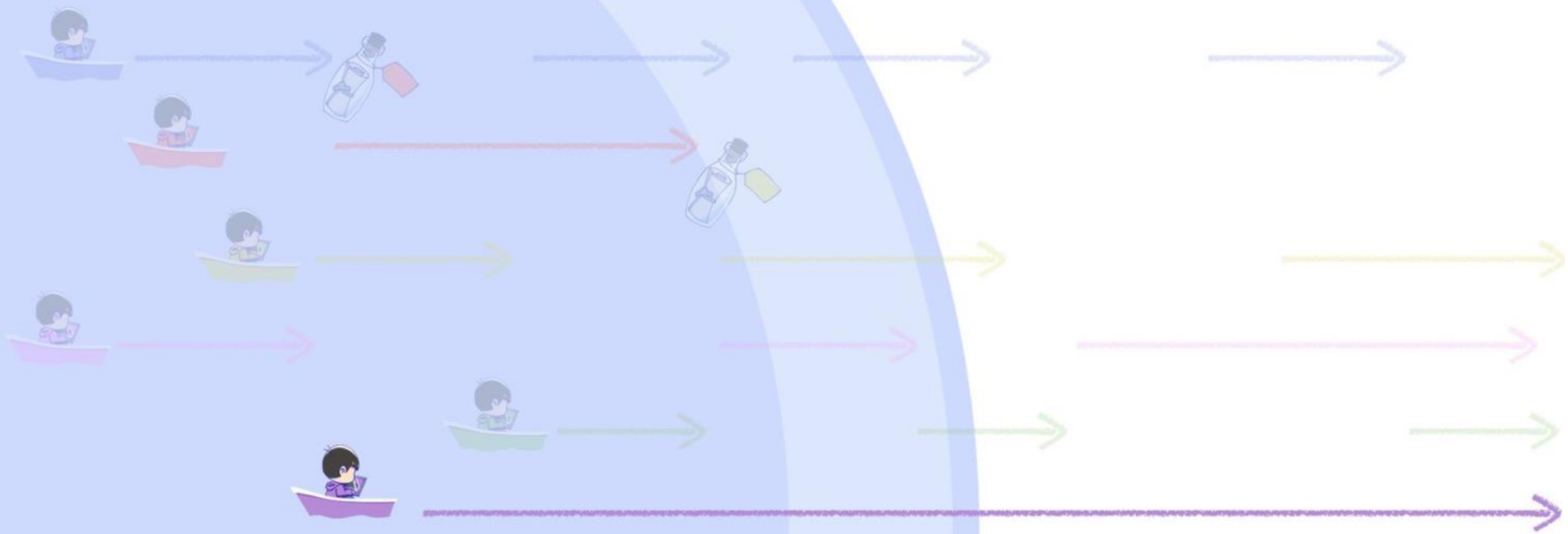


無数のプロセスとプロセス間通信がある。明示的だったり暗黙的だったりするよ

Quoted from “Learn Ractor” by Masatoshi SEKI (RubyKaigi 2023)

Concurrency is everywhere

- Sequential execution is a special case



10 むしろ、逐次実行は特殊なケースである

Quoted from “Learn Ractor” by Masatoshi SEKI (RubyKaigi 2023)

Ruby における
並行並列プログラミングの道具

並行並列処理のためのRubyの道具

- **マルチプロセス**
- Ractor (from Ruby 3.0)
- **Thread**
- Fiber scheduler (from Ruby 3.0)
- **Fiber**

大きな独立した処理

小さな処理

「何を選ぶか」は
「何を並行処理したいか」で決まる

マルチプロセス

- 道具
  - Process クラス
  - dRuby
  - Active Job, Sidekiq, …
  - Unicorn, pitchfork
- 特徴
  - 並列実行可能
  - データの共有は read only（copy on write）
    - I/O などでコミュニケーション
    - 互いに影響が少ないので強制終了などさせやすい
  - OSプロセスの管理は面倒くさいのでツールに任せるべき

マルチスレッド

- 道具
  - Thread クラス
  - Puma
  - Concurrent-ruby
- 特徴
  - **並行**実行可能
    - I/O や時間などで自動的に切り替わるモデル
    - 並列実行はしない（CRubyでは）
  - データはすべてスレッド間で共有
    - 全体のメモリは小さく収めることが可能
  - 1 Rubyスレッドにつき1ネイティブスレッド（OSスレッド）
    - 大量に作りづらい
    - プロセスよりは軽い

Fiber

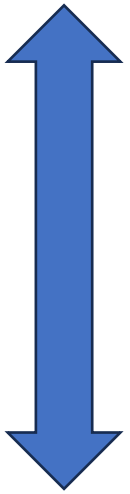
- 道具
  - Fiber クラス
    - 他の言語では coroutine という用語も
- 特徴
  - 手動で並行処理を記述可能
    - すべて自分で記述
  - データはすべて共有
  - メモリしか使わないので軽量
    - 大量に作っても比較的大丈夫

並行処理の道具

	プロセス	スレッド	Fiber
データ共有	×	○	○
並列実行	**○**	×	×
生成管理コスト	×	○	◎
コンテキスト 切り替え	**自動**	**自動**	手動

並行並列処理のためのRubyの道具

- マルチプロセス
- **Ractor (from Ruby 3.0)**
- Thread
- **Fiber scheduler (from Ruby 3.0)**
- Fiber



大きな独立した処理

小さな処理

「何を選ぶか」は
「何を並行処理したいか」で決まる

Fiber scheduler (Ruby 3.0~)

- 道具
  - Falcon（rack 用サーバ、puma のレイヤに相当）
  - Async
- 特徴
  - Fiber をスレッドのように使うための仕組み
    - ブロックする処理のタイミングで Fiber を切り替える**スケジューラを Ruby で記述可能**
    - 記述可能だが、既存のスケジューラを使うのがよい
  - だいたいスレッドと同じような使い心地
  - スケジューラ実装としての async と、それで Rack ウェブアプリケーションを記述するための falcon フレームワーク

Ractor (Ruby 3.0~)

- 道具
  - Ractor クラス
  - （まだあまり道具がない…）
- 特徴
  - 並行並列処理が可能
    - Ractor 間でデータの共有を意図的にできないようにすることで、安全な並列処理を実現
  - データの共有には強い制限
    - ユーザアプリケーションを書き換えないといけない

並行処理の道具

	プロセス	Ractor	スレッド	Fiber scheduler	Fiber
データ共有	×	△	○	○	○
並列実行	○	○	×	×	×
生成管理コスト	×	○	○	◎	◎
コンテキスト切り替え	**自動**	**自動**	**自動**	**自動**	手動

M:N Scheduler の導入（Ruby 3.3~）

- 道具
  - Thread (および Ractor)
  - 環境変数 “RUBY_MN_THREADS=1” を指定して実行
- 特徴
  - スレッドの実装を、Fiber scheduler のように、ネイティブスレッドではなくユーザレベルスレッドを用いることで、軽量に管理
  - 大量のスレッド（Ractor）を生成可能に
  - M:N スケジューラについて詳細:
  Rubyの並列並行処理のこれまでとこれから - クックパッド開発者ブログ

並行処理の道具

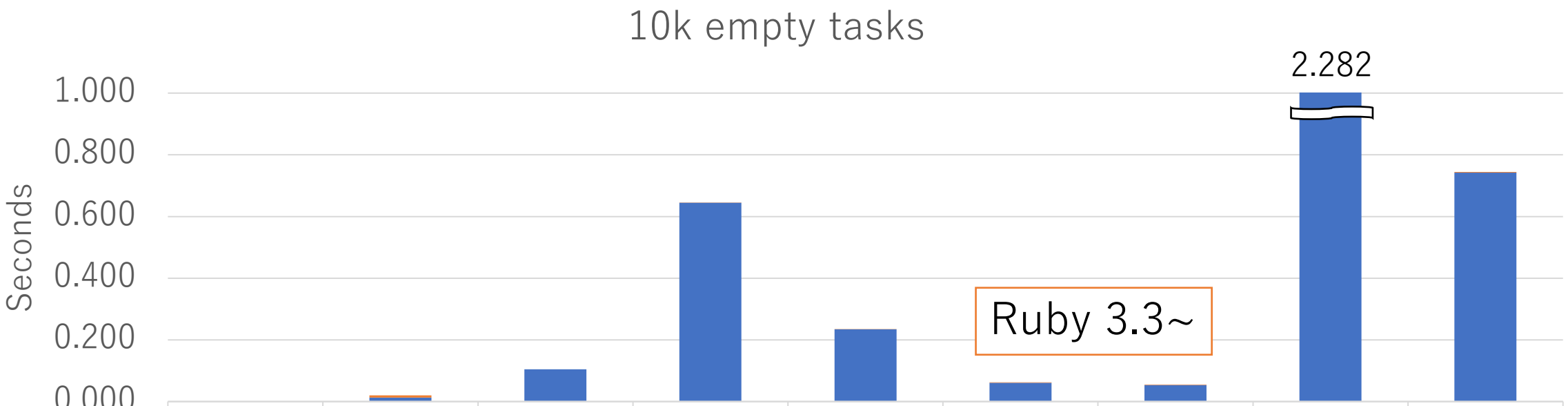
	プロセス	Ractor +M:N	スレッド + M:N	Fiber scheduler	Fiber
データ共有	×	△	○	○	○
並列実行	○	○	×	×	×
生成管理コスト	×	○ → ◎	○→ ◎	◎	◎
コンテキスト切り替え	**自動**	**自動**	**自動**	**自動**	手動

理論的にはあまり変わらない
あとはアプリへの組み込みやすさ、実装の作りこみの差

Benchmark
Empty concurrent tasks

https://gist.github.com/ko1/ff2cc7d83ae5f73a23ad3ba69a930b58
ruby 3.3.0dev (2023-11-07T04:38:37Z master d6f5c27525)

```
def task = nil
```



	seq	fib	async	threads	threads/ NGC	threads/ MN	threads/ MN,NGC	ractors	ractors/ NGC
wait	0.000	0.007	0.000	0.001	0.001	0.001	0.001	0.003	0.002
make	0.000	0.013	0.105	0.645	0.234	0.061	0.054	2.279	0.743

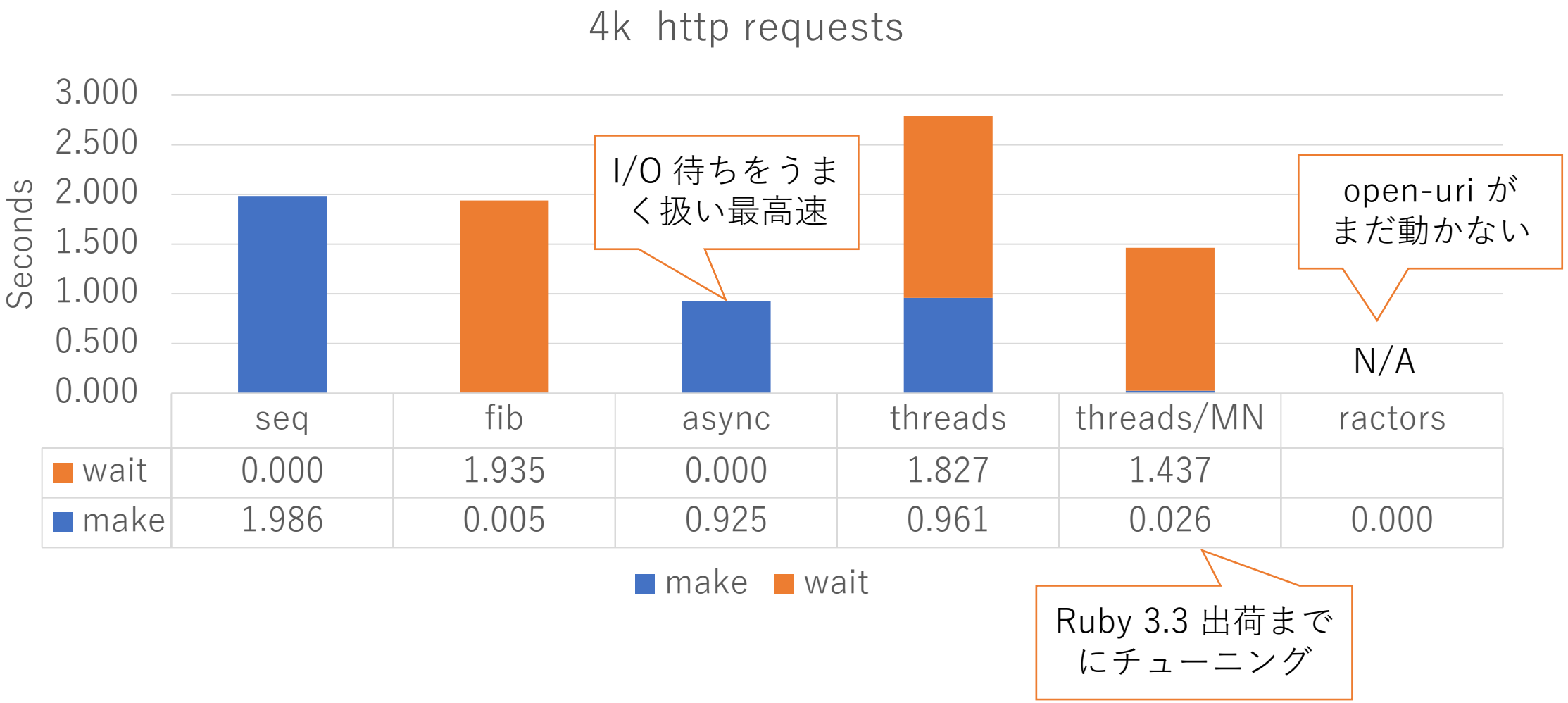
seq: Sequential execution
fib: manual Fiber scheduling (=~ seq)
async: Fiber scheduler with async.gem

make wait

MN: Enable MN Threads
NGC: Disable GC

Benchmark
Concurrent HTTP requests

```
def task =
  URI.open(...){|f| f.read}
```



	seq	fib	async	threads	threads/MN	ractors
wait	0.000	1.935	0.000	1.827	1.437	
make	1.986	0.005	0.925	0.961	0.026	0.000

Thread API と同期

Thread の生成

- Thread.new{ … } で … を並行に処理するスレッドを生成
- … が終わるまでスレッドオブジェクトは生き続ける

```
t1 = Thread.new do
  doA()
end

t2 = Thread.new do
  doB()
end
```

Thread の待ち合わせ

- Thread#value もしくは Thread#join でスレッド終了を待つ
  - #value はブロックの返り値を返す

```
t1 = Thread.new{ doA() }
t2 = Thread.new{ doB() }
p t1.value #=> doA() の結果
p t2.value #=> doB() の結果
```

- Queue / SizedQueue を利用する
  - より汎用的な待ち方

```
q = Queue.new
t1 = Thread.new{ q << doA() }
t2 = Thread.new{ q << doB() }
2.times{ p q.pop } # ２つの実行結果を待つ
```

Thread のエラーハンドリング

- スレッド実行時にエラーでスレッドが死んだ場合、Thread#value もしくは Thread#join でエラーを返す

```
t1 = Thread.new{ raise "foo" }

p t1.value #=> foo (RuntimeError)
```

- エラーレポート
  - Thread#abort_on_exception = true (default: false)
    - エラーでスレッドが死んだらRubyプロセスを終了する
  - Thread#report_on_exception = true (default: **true**)
    - エラーがスレッドが死んだら標準出力にエラーを表示
- 真面目にやるなら、エラーで死なないようにエラー処理を記述

Threadの中断

- Thread#kill (#terminate) でスレッドを無理やり中断
- Thread#raise でスレッドに例外を発生させ中断
- どういうタイミングで中断されるかわからないので、非推奨

```
t1 = Thread.new{ while true; end }
     # t1 は無限ループ

t1.kill
t1.join
```

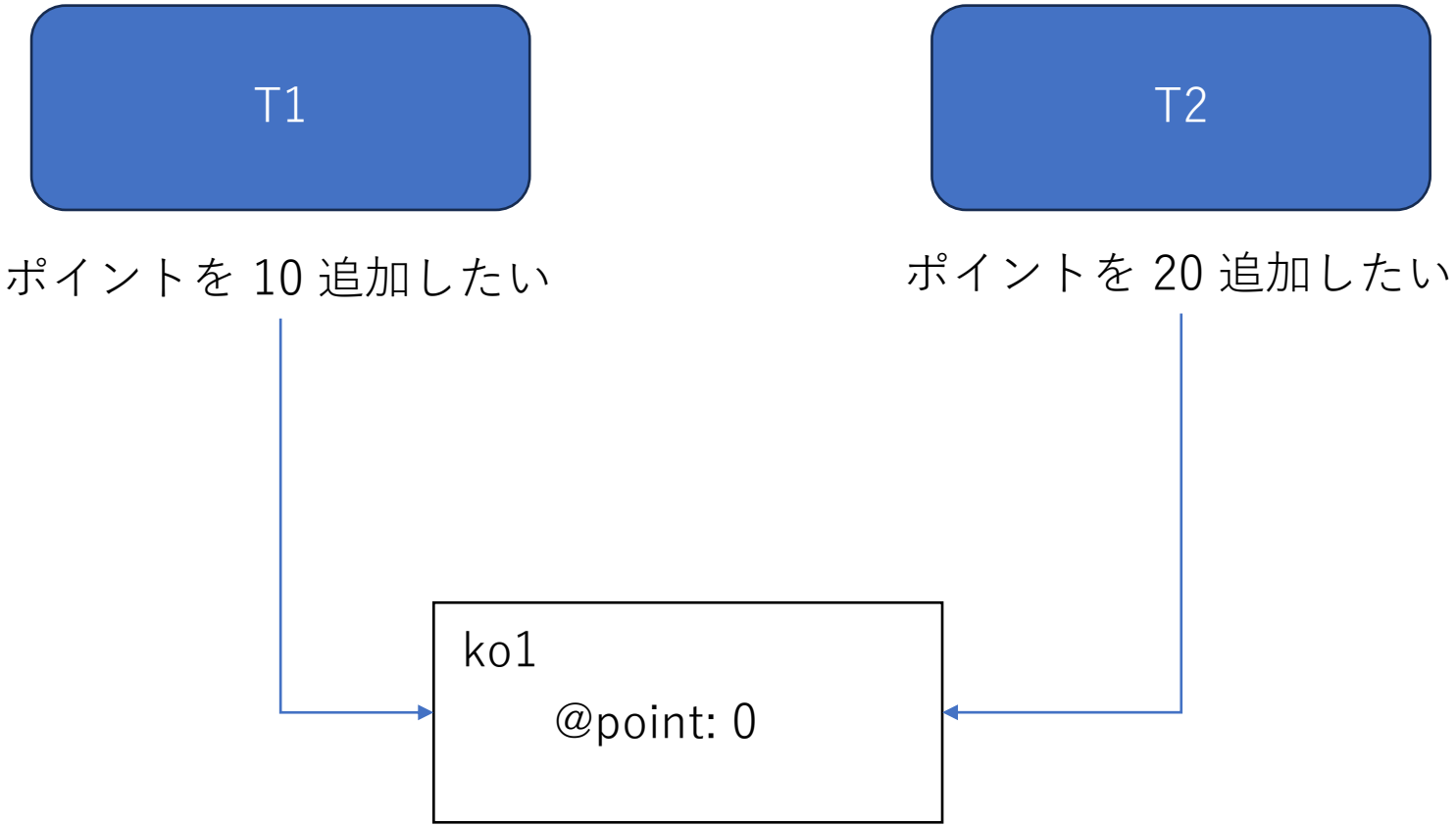
Threadの同期

- データの読み書きの一貫性保持
  - 同時に読み書きするとまずいことが起こる
  → 同期 (synchronization) が必要


```
class Person
  attr_accessor :point

  def initialize
    @point = 0
  end

  def inc n
    @point += n
  end
end

ko1 = Person.new
```

Threadの同期

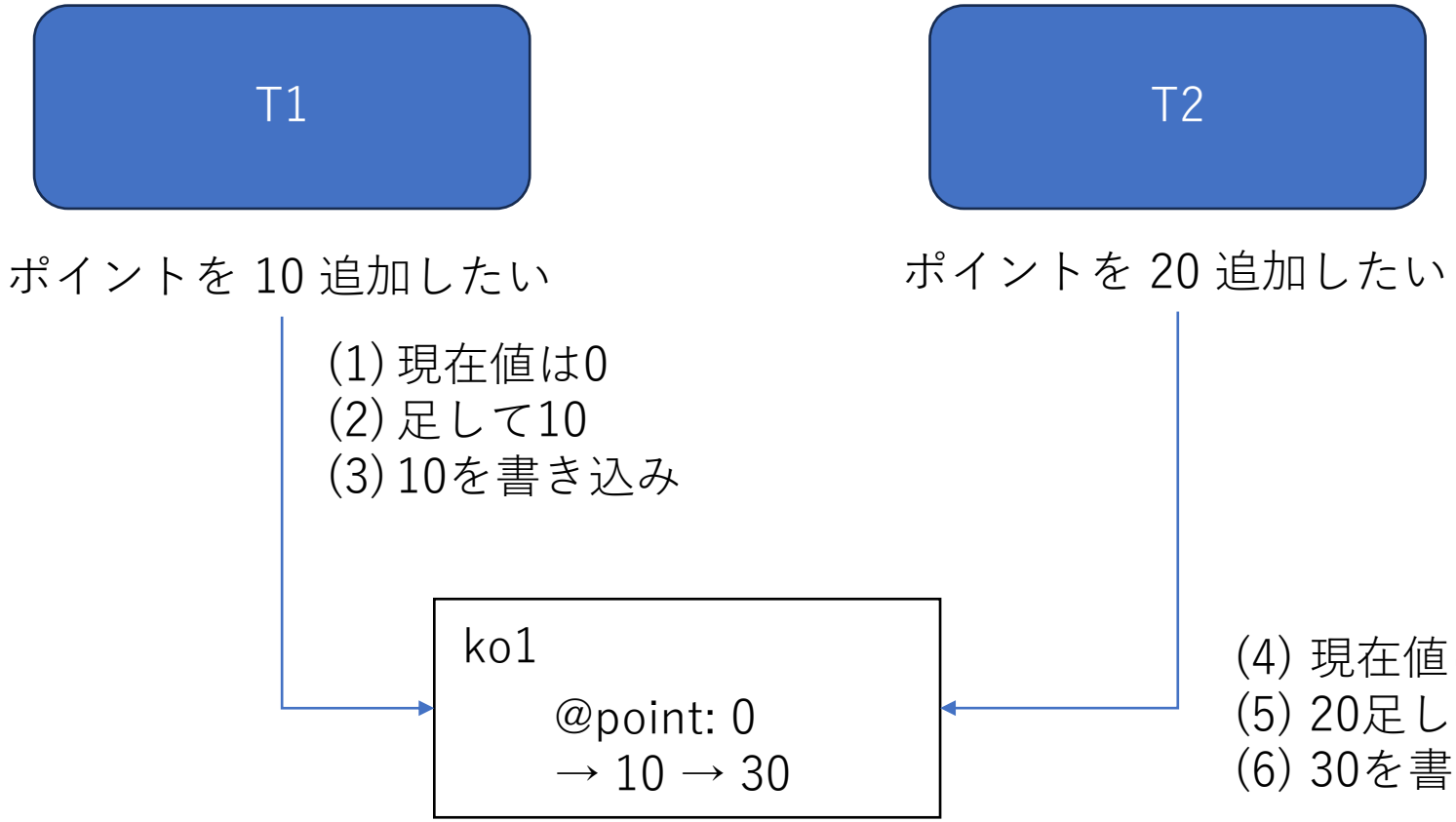
- スレッド 1 とスレッド 2 が同時にポイント追加（10と20）




```
t1 = Thread.new do
  ko1.inc(10)
end
t2 = Thread.new do
  ko1.inc(20)
end
t1.join; t2.join
p ko1.point #=> 30…?
```

Threadの同期

- スレッド 1 とスレッド 2 が同時にポイント追加（10と20）

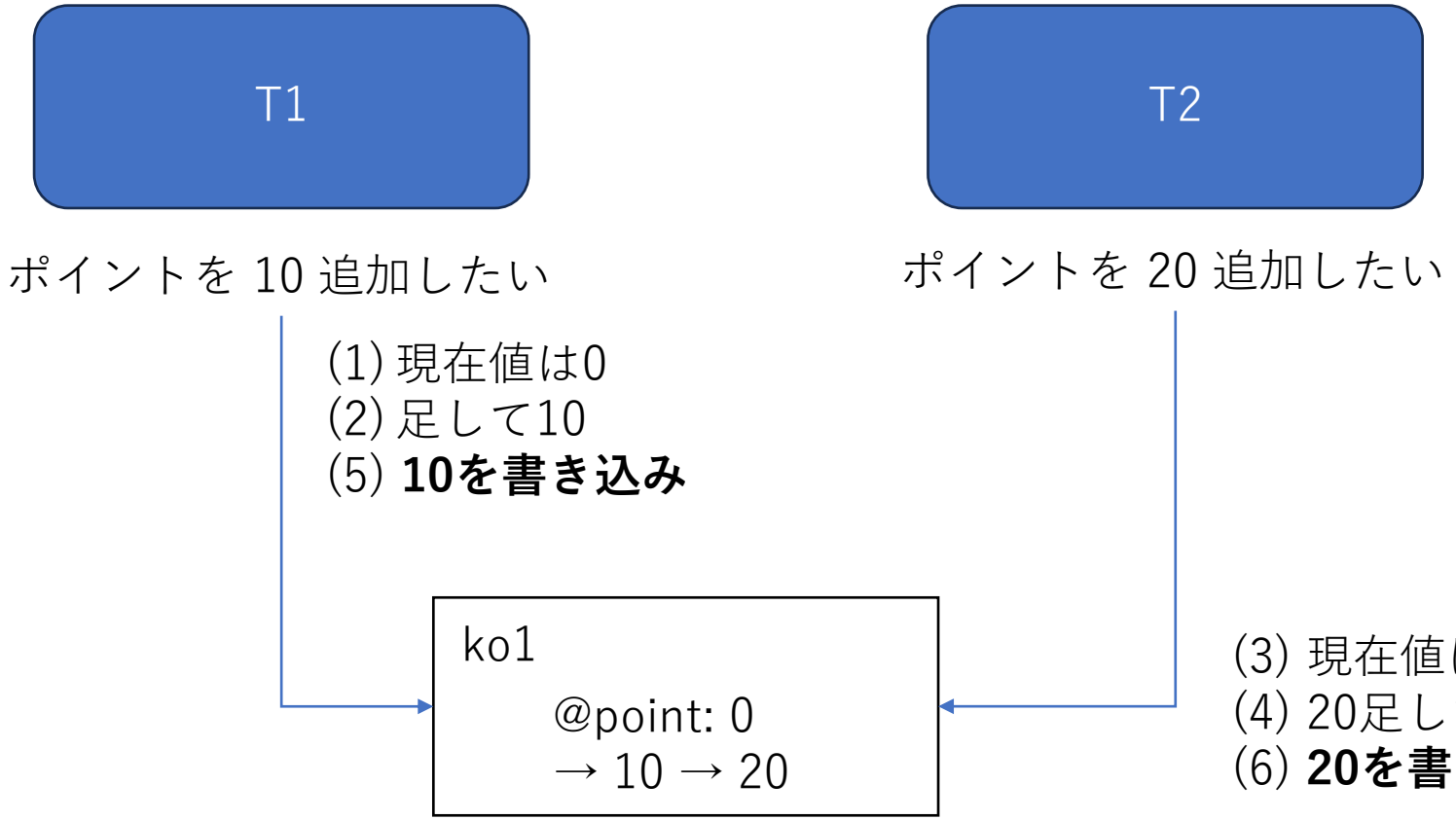



```
t1 = Thread.new do
  ko1.inc(10)
end
t2 = Thread.new do
  ko1.inc(20)
end
t1.join; t2.join
p ko1.point #=> 30…?
```

問題ないケース

Threadの同期

- スレッド 1 とスレッド 2 が同時にポイント追加（10と20）

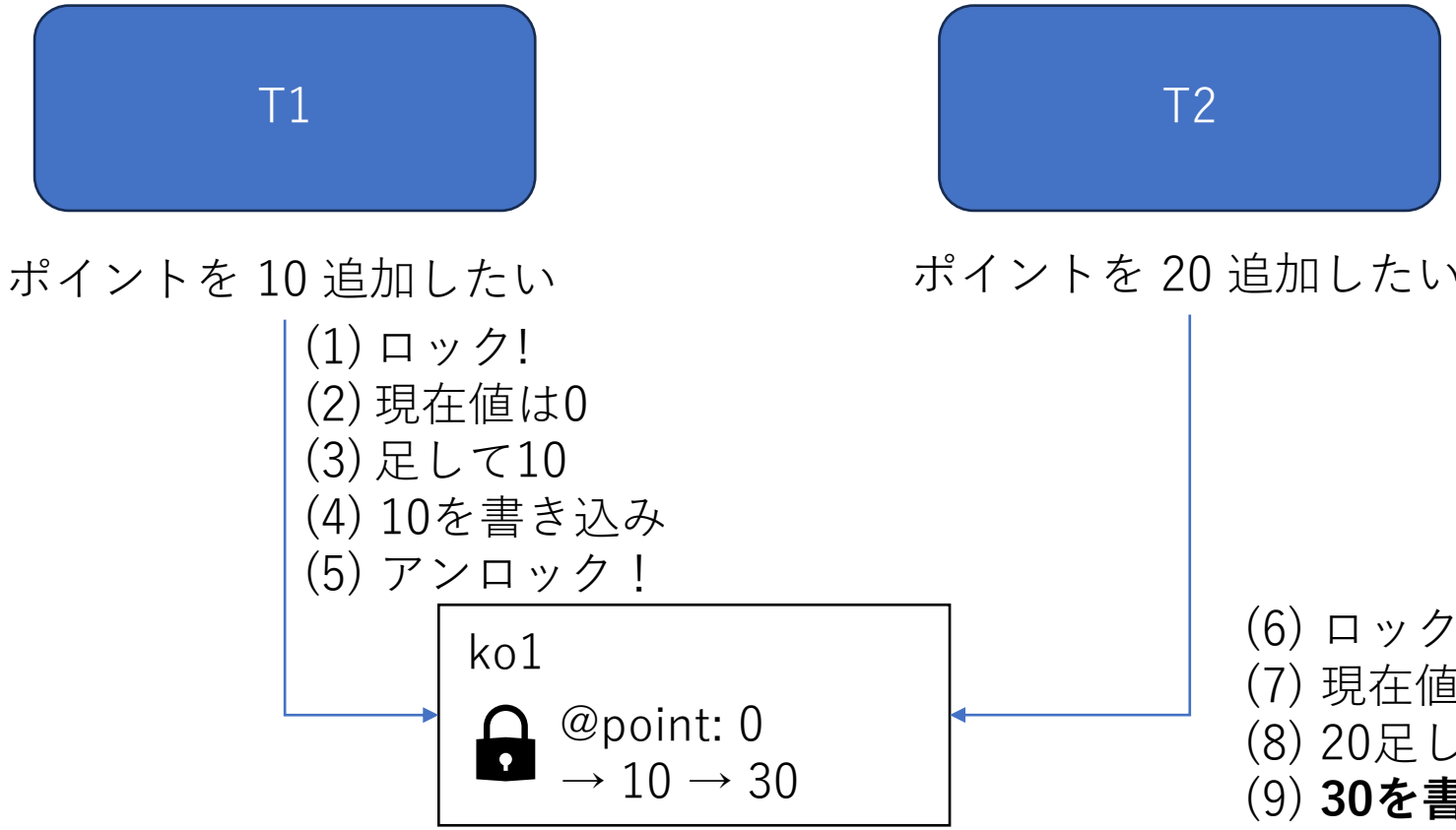



```
t1 = Thread.new do
  ko1.inc(10)
end
t2 = Thread.new do
  ko1.inc(20)
end
t1.join; t2.join
p ko1.point #=> 30…?
```

問題あるケース

Threadの同期
ロックの利用

- スレッド 1 とスレッド 2 が同時にポイント追加（10と20）




```
class Person
  def initialize
    @point = 0
    @m = Mutex.new
  end

  def inc n
    @m.synchronize{
      @point += n }
  end
end
```

ロックしているスレッドのみ書き込み

Threadの同期
ロックによる同期の問題点

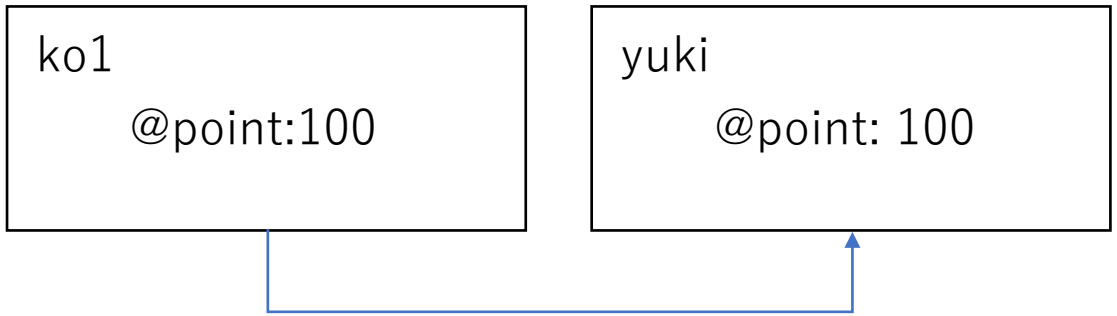
- 使うのを忘れちゃう
  - **ついうっかり**
  - ロックしなくても使える（読める・書き込める）
  - 問題が発生しても、気づきづらい
    - 大体の場合、問題が発生しない
    - 発生しても、微妙な差異に気づくのが困難（ポイントが 20 になっても…）
    - プログラムが複雑になると、気づくのが無茶苦茶難しい
  - Rust などでは型で強制
- 複数のロックを組み合わせるのが困難
  - 容易なデッドロック

Threadの同期
複数ロックの問題

- ポイントを移したい！！

ko1 から yuki へポイント移動したい

yuki から ko1 へポイント移動したい

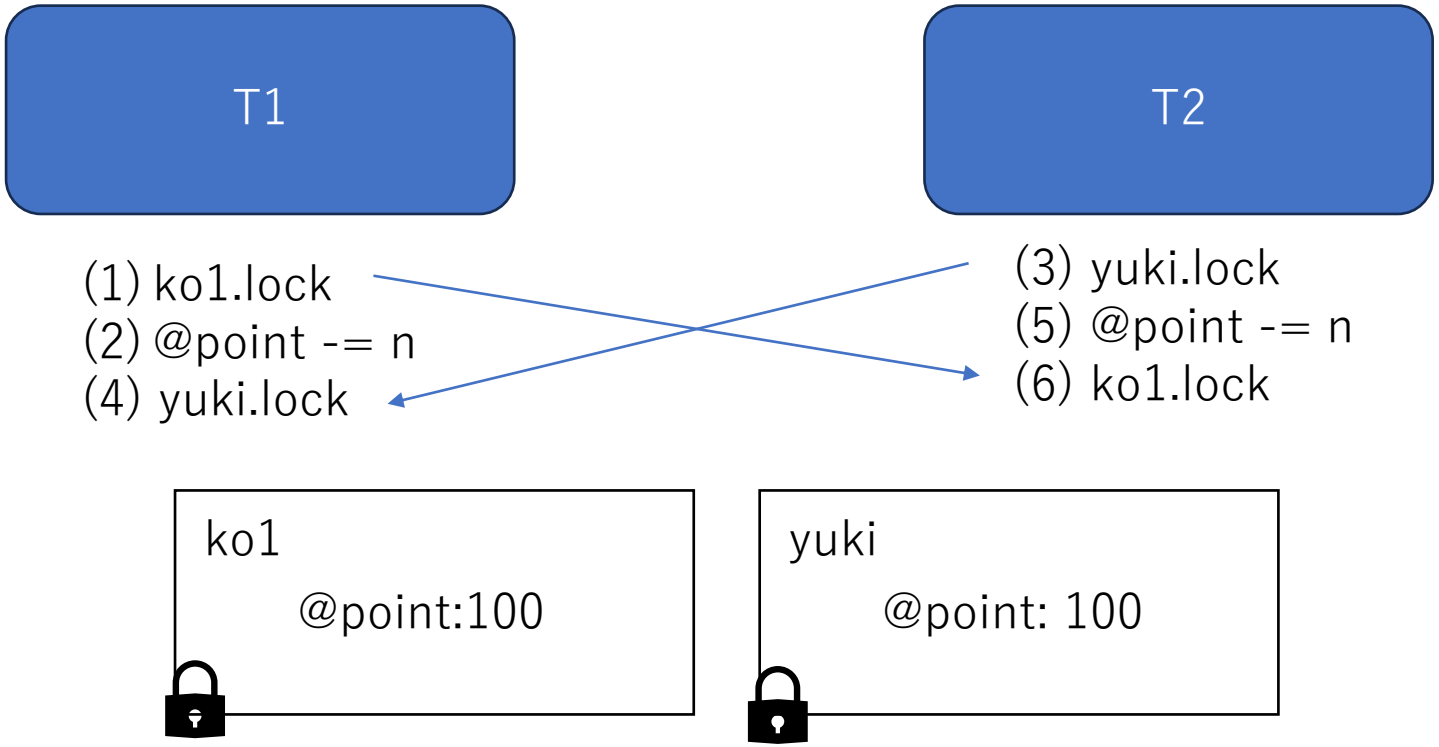



```
class Person
  def pass_to(other, n)
    @point -= n
    other.inc n
  end
end

t1 = Thread.new{
  ko1.pass_to(yuki, 20) }
t2 = Thread.new{
  yuki.pass_to(ko1, 30) }
```

ロックしないと一貫性の問題

Threadの同期
複数ロックの問題

- ポイントを移したい！！




```
class Person
  def pass_to(other, n)
    @m.synchronize{
      @point -= n
      other.inc n
    }
  end
end
```

相手のロックがすでにとられてる
→デッドロック！！

Threadの同期
複数ロックの問題

- 解決策
  - ロックの順番を毎回同じにする
    - 例: Ko1 <-> Yuki なら、K のほうが辞書順ではやいので、K と Y に関連する処理は必ず K->Y の順にする
  - 相手がロックされているのを検出したら、一度ロックを開放してやり直す
  - 個々のロックを用いず、ジャイアントロックを用いる
  - **Ruby でこんな並行処理を書かない**
    - この手のものは並行処理しない
      - スレッドは使わない（マルチプロセスや Ractor の利用を検討する）
      - データに触ることができる専用スレッドを用意する、など
    - 外部データベースに任せる

別解: Transactional memory による同期

- 悲観的ロック vs. 楽観的ロック
  - 悲観的：他が変更するかもしれないから変更不可にして処理
  - 楽観的：他も変更するかもしれないけど、もしダメならやりなおそう
- データベースの（楽観的）トランザクションで著名
  - 現在の値で処理をとりあえず実行
  - 最後のコミット時、すでにほかがコミットしていた（衝突した）とき、処理をやりなおす
  - コンポジション可能（トランザクションのネストが可能）

Transactional variable によるポイントシステム

- $ gem install ractor-tvar https://github.com/ko1/ractor-tvar
- 名前のとおり、Ractor 間で利用可能
- TVar#atomically で囲むとトランザクション
  - トランザクション内の TVar#value への書き込み、ブロック終了時にコミット
  - コンフリクトしていたら失敗、ブロックを再実行
- atomically ブロック外（トランザクション外）では値にアクセスできない → 忘れない

```
class Person
  attr_accessor :point

  def initialize
    @point = Ractor::TVar.new(0)
  end

  def inc n
    @point.atomically do
      tv.value += n
    end
  end
end
```

もしほかのスレッドが value を書き換えていたらコミット失敗 ブロック再実行

Transactional variable によるポイントシステム

- トランザクションはネスト可能
  - トランザクションの順序を問わないので、ロックのようなデッドロックの心配なし
  - コンフリクトしたら一番外側のトランザクションから再実行

```
class Person
 def pass_to other, n
    @point.atomically do
      other.point.atomically do
        @point.value -= n
        other.point.value += n
      end
    end
  end
end
```

まとめ

- Rubyでの並行・並列処理のための仕組み
  - プロセス、Ractor、スレッド、Fiber scheduler, Fiber
  - Ruby 3.3 から導入される M:N スケジューラでいろいろ軽量に
- 共有データを複数スレッドで読み書きするなら同期は大変
  - 適切なロックが必要
  - 大変なので、なるべくやらないのがよい
  - Transactional Memory を用いた Ractor::TVar という別解もあるよ

Rubyによる
並行並列プログラミング

笹田 耕一
<ko1@st.inc>

STORES