Progress report of "Ruby 3 Concurrency"

Cookpad Inc.
Koichi Sasada
<ko1@cookpad.com>
Today’s topic

• Difficulty of **Thread programming**
• New concurrent abstraction for Ruby 3 named **Guild**
  – To overcome threading difficulties
• Introduce current Guild development progress
  – Current “Semantics”
  – Current API design and sample code we can run
  – Preliminary performance evaluation
Koichi Sasada
http://atdot.net/~ko1/
• Programmer
  – 2006-2012 Faculty at Univ.
  – 2012-2017 Heroku, Inc.
  – 2017- Cookpad Inc.
• Job: MRI development
  – MRI: Matz Ruby Interpreter
  – Taking a charge of core parts
    • VM, Threads, GC, etc
One of Japanese translators of “Programming Elixir”

Written by Dave Thomas
Translated by
Koichi Sasada
Yuki Torii
2016 Ohmsha
Recent achievements for Ruby 2.6

• **Speedup `Proc#call`** because we don't need to care about `$SAFE` any more. [Feature #14318]. With `lc_fizzbuzz` benchmark which uses so many `Proc#call` we can measure x1.4 improvements [Bug #10212].

• **Speedup `block.call` where `block` is passed** block parameter. [Feature #14330] Ruby 2.5 improves block passing performance. [Feature #14045] Additionally, Ruby 2.6 improves the performance of passed block calling.
A proposal of new concurrency model for Ruby 3
Motivation

Productivity (most important for Ruby)
• Thread programming is too difficult
• Making correct/safe concurrent programs easily

Performance by Parallel execution
• Making parallel programs
• Threads can make concurrent programs, but can’t run them in parallel on MRI (CRuby)
• People want to utilize Multi/many CPU cores
RubyKaigi2016 Proposal

Guild: new concurrency abstraction for Ruby 3

- Idea: **DO NOT SHARE** mutable objects between Guilds
  → No data races, no race conditions

Replace Threads to Guilds
DIFFICULTY OF MULTI-THREADS PROGRAMMING AND HOW TO SOLVE IT?
Multi-threads programming is difficult

- Introduce data race, race condition
- Introduce deadlock, livelock
- Difficulty on debugging because of nondeterministic behavior
  - difficult to reproduce same problem

- Difficult to tune performance

Difficult to make correct (bug-free) programs

Difficult to make fast programs
Inter-thread communication

\[ v = \text{Object.new} \]
\[ \$g = \text{Object.new} \]
\[ \text{Thread.new do} \]
\[ \quad p \ [v, \ \$g] \]
\[ \text{end} \]
\[ p \ [v, \ \$g] \]

We can share objects directly between threads very easily.
Mutate shared objects
Lucky case

Ruby process

Thread A

@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”

Thread B
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

read

SpeakerObject

@name = ‘Yuki’
@gender = ‘female’

Thread B

read

(4) Complete.
A and B can read correct speaker.
Mutate shared objects
Problematic case

Ruby process

SpeakerObject
@name = 'ko1'
@gender = 'male'

(1) Thread A tries to change the SpeakerObject 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

Ruby process

Locked by A

Thread A

@name = 'ko1'
@gender = 'male'

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

Thread B
Thread A

Locked by A

SpeakerObject

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

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

Ruby process

Locked by A

SpeakerObject

Thread A

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

Thread B

(3) Before complete the changing, B tries to read, but prohibited by a lock
Difficulty of multi-threads programs
Easy to share objects between Threads

• We need to synchronize all sharing **mutable objects** correctly
  – Easy to share objects, but difficult to recognize
    • We can track on a small program, but…
    • Difficult to track them on **big programs**, especially on **programs using many gems**
• We need to check **whole source codes includes libraries**, or believe **library documents** (but documents should be correct)
Goal of Ruby 3 concurrency

• Easy to make “Correct” concurrent program
  – Restrict sharing mutable objects between threads
  – Introducing Objects ISOLATION mechanism
• Support parallel programming
  – Running programs simultaneously on multi-cores
  – Introducing MINIMUM synchronizations to MRI
• Keep compatibility with Ruby 2
Key idea

Problem of multi-thread programming:
Easy to share mutable objects

Idea:
Do not allow to share mutable objects without any restriction
Options (1)
Make all objects immutable
Like Elixir!!

But it break Ruby’s compatibility!!
Option (2)
Copy everything

Like shell script (pipe), dRuby, …

But it is difficult (sometimes) and copying causes overhead.
Options

• (1) Make all objects immutable  
  – Good: No mutable sharing  
  – Bad: Huge incompatibility issue

• (2) Copy everything  
  – Good: No mutable sharing, no compatible problem  
  – Bad:  
    • No sharing objects is difficult to make programs  
    • Copy overhead

• (3) Share only “shareable” objects
Options (3)

Share only “shareable” objects

Good: (Normal) mutable objects can’t share between concurrent entities
Good: Easy to share “shareable” objects
Good: No compatible issue (at least on only 1 concurrent entity)
GUILD
NEW CONCURRENT ABSTRACTION FOR RUBY3
Guild Guide

• Guilds, Threads and Fibers
  – Relations between Guilds, Threads and Fibers
  – How to create Guilds in Ruby code?
• Inter-Guild communication
  – Isolation design: Shareable and non-shareable objects
  – Send by copy and move
• Example patterns
Guilds, Threads and Fibers

- 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)
Making Guilds

g1 = Guild.new do
  expr1
end

g2 = Guild.new do
  expr2
end

# Two new Guilds and Threads are created
# expr1 and expr2 are run in parallel
Inter-Guild communication
Share only “shareable” objects

Guild 1

normal mutable obj1

Shareable obj1

Guild 2

normal mutable obj2

Shareable obj2
Design “Shareable” and “non-sharable”

- On concurrent programs, most of objects are not shared (thread-local)
  - **Tons** of local objects and **a few** sharing objects
  - We can introduce sharing objects which requires synchronization to make correct concurrent programs but they cause additional overhead
Design “Shareable” and “non-sharable”

- **Non-shareable** objects
  - (normal) Mutable objects (String, Array, …)
  - They are **member of only one Guild**
  - Using only 1 Guild, it compatible with Ruby 2

**Guild 1**

- obj
- obj
- obj

**Guild 2**

- obj
- obj

NG!!

Can’t access (read/write)
Design “Shareable” and “non-sharable”

• Shareable objects
  – (1) Immutable objects (Numeric, Symbol, …)
  – (2) Class/Module objects
  – (3) Special mutable objects
  – (4) Isolated Proc
Shareable objects
(1) Immutable objects

- **Immutable objects** can be shared with any guilds
  - Because no mutable operations for them
- “Immutable” != “Frozen”
  - `a1 = [1, 2, 3].freeze: a1 is Immutable`
  - `a2 = [1, Object.new, 3].freeze: a2 is not Immutable`
  - Maybe we will introduce deep freeze feature
- **Numeric objects, symbols, true, false, nil** are immutable (from Ruby 2.0, 2.1, 2.2)
- **Frozen string objects** are immutable (if they don’t have instance variables)
Shareable objects
(2) Class/Module objects

- All objects (includes any sharable objects) point to own classes
  - Good: Sharing class/module objects makes program easier
  - Bad: They can points other mutable objects with Constants, @@class_variable and @instance_variables

```ruby
class C
  Const = [1, 2, 3] # Const points a mutable array
end
# We will introduce special protocol for them
```
Shareable objects
(3) Special mutable objects

• Introduce shared/concurrent data structure
  – Shared hash, array, …
  – Software transactional memory (from Clojure, …), …
  – Guild objects and so on

• They require special process to force synchronization explicitly
  → Correct concurrent programs

• Compared with normal Array, Hash, … they require special synchronization protocol to access
Shareable objects
(4) Isolated Proc

• normal Proc can points mutable objects with outer local variable (free-variables)

  ```ruby
  a = []; Proc.new{p a}.call
  ```

• Introduce **Isolated Proc** (made by `Proc#isolate`) which is prohibited to access outer variables

  ```ruby
  a = []; Proc.new{p a}.isolate.call
  #=> RuntimeError (can't access a)
  ```

  (there are more details but skip)
# Initial block for Guild is isolated proc

g1 = Guild.new do
  expr1  # Make isolated block and invoke
end

g2 = Guild.new do
  p g1 #=> RuntimeError (can’t access “g1”)  # because block is isolated
end
Inter-Guild communication API

- send/receive semantics
- Address is represented by Guild itself like Erlang/Elixir processes
- Sending shareable objects means sending only references to the objects (lightweight)
- Two methods to send non-shareable objects
  - (1) COPY
  - (2) MOVE
Sending objects between Guilds

g1 = Guild.new do # create Isolated Proc
  n = Guild.receive
  r = fib(n)
  Guild.parent.send(r)
end

g1 << 30

p Guild.receive #=> 1346269
Sending shareable objects

\[ g_2 \triangleleft\triangleright o_1 \]

\[ \text{Guild1: } g_1 \]

\[ o_1 = \text{Guild.receive} \]

\[ \text{Guild2: } g_2 \]

\[ o_2: \text{Data} \]

\[ o_3: \text{Data} \]
Sending non-shareable objects
(1) Send by **Copy**

g2 \(\ll o1\)

**Guild1**

- o1
- o2
- o3

- Q2:Data
- Q3:Data

**Guild2**

- o1
- o2
- o3

- O2:Data
- O3:Data

**COPY**

channel
Sending non-shareable objects
(2) Send by **Move**

```
g2.move(o1)
```

```
o1 = Guild.receive
```

**Guild1**

- o1
- o2
- o3

**Guild2**

- moved

---

```
O2:Data
O3:Data
```

**channel**
Sending non-shareable objects (2) Send by Move

g2.move(o1)

From Guild1 perspective, sent objects are invalidated
Sending non-shareable objects

(2) Send by **Move**

- If we don’t access sent objects after sending them (and there are many such cases), we can send them faster

- Examples
  - Huge string data
  - I/O operation (send request I/O to workers)
Summary of object sharing/non-sharing

• Shareable objects
  – Several types of shareable objects
  – We can share them between Guilds

• Non-sharable objects
  – Normal mutable objects (like String, Array, …)
  – Only one Guild can access such objects == membership
  – We can send them by COPY or MOVE

• Mutable objects are NOT shared accidentally as
Thread programming → Correct concurrent Prog.
Patterns

• (1) Master-worker pattern
• (2) Pipeline pattern
(1) Master-worker pattern

```ruby
# make N guilds
gs = N.times.map{
  Guild.new do
    n = Guild.receive
    Guild.parent << fib(n)
  end
}
# send task
gs.each{|g|
  g << P
}
# receive answers
N.times{
  p Guild.receive
}
```
(2) Pipeline pattern

• Run different tasks for one data
• Example

```ruby
str = ' foobarbaz 
str = str.strip.upcase.gsub('A', 'B') #=> "FOOBBRBBZ"
```

#=> There are 3 different tasks
str = str.strip
str = str.upcase
str = str.gsub('A', 'B')

# on Elixir
str |> String.trim
    |> String.upcase
    |> String.replace(...)
(2) Pipeline pattern

```
g_strip = Guild.new do
  next_guild = Guild.receive
  while str = Guild.receive
    next_guild.move str.strip
  end
  next_guild << nil
end

g_upcase = Guild.new do
  next_guild = Guild.receive
  while str = Guild.receive
    next_guild.move str.upcase
  end
  next_guild << nil
end

g_replace = Guild.new do
  next_guild = Guild.receive
  while str = Guild.receive
    next_guild.move str.gsub('A', 'B')
  end
  next_guild << nil
end

g_strip << g_upcase
  g_upcase << g_replace
  g_replace << Guild.current
  g_strip.move 'foobarbaz'
p Guild.receive
```

**Main Guild**

- `return`: Move

**Strip Guild**

- `strip`:
  - `strip`: Move

**Replace Guild**

- `replace`:
  - `replace`: Move

**Upcase Guild**

- `upcase`:
  - `upcase`: Move

**Return**

- `foobaz`
(2) Pipeline pattern
Framework for frequent patterns

class Guild
  # Make series of Guilds for a pipeline
  def self.pipeline *tasks
    task.guilds = tasks.map{|task|
      Guild.new do
        next_guild = Guild.recv
        while obj = Guild.recv
          next_guild.move task.call(obj)
        end
        next_guild.move nil
      end
    end
    next_g = Guild.current
    task.guilds.reverse_each{|g|
      g.send next_g
      next_g = g
    }
    task.guilds.first
  end
end

We need to design a library like OTP.
Supposed usecases

• Web application backend
  – Guild pool for request workers
  – Straight forward approach
Experimental results

Run fib(36) on 40 cores machine
(2 HT x 10 cores x 2 processors Xeon E5-2630 v4)
Experimental results

Run \text{fib}(36)\ on 40\ cores\ machine

(2\ HT\ x\ 10\ cores\ x\ 2\ processors\ Xeon\ E5-2630\ v4)
Note

There are more and more details to discuss

• Semantics: so many topics
  – Global variables
  – Instance variables for shareable objects
  – Class/Module specific shared data synchronization
    • Constants, class variables
    • Method table
  – C API compatibility for thread safety
  – Isolated Proc semantics and error detections
  – Supports (syntax, runtime) to make immutable objects
  – I/O? Current working directory?
  – How to define moving protocol?
  – What kind of “Transaction” is supported for shareable data?
  – ObjectSpace.each_object?
  – Signal handlers?
  – …

• Implementation: we need to revisit all of MRI code to change the assumption “GVL/GIL can protect all”
  – How to reduce internal locks for performance?
  – How to make parallel GC?
  – Rewrite MRI implementations to make them thread-safe (e.g.: Regex)
  – Introduce new C APIs to accept “ec” parameters to remove TLS access overhead
  – Really compatible with Ruby 2?
  – …
## Pros./Cons. Matrix

<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>
Today’s topic

• Difficulty of **Thread programming**
• New concurrent abstraction for Ruby 3 named **Guild**
  – To overcome threading difficulties
• Introduce current Guild development progress
  – Current “Semantics”
  – Current API design and sample code we can run
  – Preliminary performance evaluation
Thank you for your attention

Koichi Sasada
<ko1@cookpad.com>