Ruby - Feature #18926

Ractor should support mutexes and treat the block as critical section across ractors

07/19/2022 02:26 PM - chucke (Tiago Cardoso)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

This is an improvement suggestion in order to foster adoption of ractors. It may not be technically impossible or unfeasible for some reason, as it may lead to deadlocks, so feel free to discard it if massively hard to undertake.

There's a pattern, common to a lot of popular gems, and stdlib, in the wild, which I call "lazy-load-then-cache". Essentially, smth like:

```
class A
  @map = {}
  @map_mutex = Mutex.new

  def self.set_in_map(name, value)
    @map_mutex.synchronize do
      @map[name] = value
    end
  end

  def self.get_from_map(name)
    if not @map.key?(name)
      value = do_smth_heavy_to_figure_out(name)
      set_in_map(name, value)
      value
    else
      @map[name]
    end
  end
end
```

The main issues here regarding ractor safety are:

- @map is not frozen
- ractor does not support usage of mutexes

Examples:

- sequel: https://github.com/jeremyevans/sequel/blob/master/lib/sequel/database/connecting.rb#L76
- resolv:
  - https://github.com/ruby/resolv/blob/master/lib/resolv.rb#L187 (instance-based variation of the above)
  - https://github.com/ruby/resolv/blob/master/lib/resolv.rb#L1341 (not protected by mutex, but still a usage of a global counter)

While I've found a gem implementing a ractor-aware cache, while looking a bit outdated, it also makes use of ObjectSpace::WeakMap, which is probably a dealbreaker and a "hack" around ractor's limitations. It's also not necessarily a "drop-in" replacement for the pattern exemplified above.

Theoretically, ractor could support the pattern above, by allowing the usage of mutexes. It should however run mutex blocks exclusively across ractors, while also disabling "ractor checks". This means that a mutable @map could be mutated within a ractor, as long as protected by a mutex. However, it should be marked as frozen before the block terminates. So the code above should be modified to:

```
@map = {}.freeze
@map_mutex = Mutex.new

def self.set_in_map(name, value)
  @map_mutex.synchronize do
```

```
    @map = @map.merge(name => value)
    @map.freeze
  end
end
```

Again, there may be implementation limitations not enabling usage of such a pattern. But it's a telling sign when ractors can't support simple usages of stdlib. So this proposal aims at enabling yet another case which may diminish the overhead of supporting ractors going forward, thereby making ractors usable in more situations.

History

#1 - 07/23/2022 02:34 AM - ianks (Ian Ker-Seymer)

With all due respect for Ractor's incredible technical achievements so far, I unfortunately think that the "honeymoon phase" for Ractor may be complete. The community was incredibly bullish about its potential and the problems it could solve, but so far it has proven to be too burdensome. I could be misinformed, but I personally don't know of any serious projects making use of Ractor for this reason.

It's unfortunate, but not all hope is lost…

I think it would be wise to honestly examine the shortcomings of Ractor. And if needed, entirely re-design to make it more useful to the Ruby community. If that isn't possible, we should explore ways to communicate the shortcomings, and provide a clear path and solutions for these common issues.

A lot of hard work and technical accomplishments have gone into making Ractor even possible, and I have a deep respect for it. I am just worried Ruby users might lose faith in it.

#2 - 07/23/2022 11:40 AM - Eregon (Benoit Daloze)

In the example, allowing to mutate @map in a Ractor is fundamentally unsafe, because there could be another reference to @map which is accessed concurrently without the Mutex.
In fact there are two in that example, @map.key? and @map[name] inside get_from_map (and it is not enough to only synchronize the writes).
I don't think it is feasible in Ruby to track all references to an object in any kind of efficient manner, so this seems unfeasible to me at least with a Hash.

IMHO Ractor while fun are incompatible with the vast majority of Ruby code, gems, and many Ruby patterns.
If you want parallel execution of Ruby code compatible with the majority of Ruby code, use Threads on TruffleRuby or JRuby (also avoids the copying/freezing overheads whenever passing an object to another Ractor).

#3 - 07/23/2022 11:42 AM - Eregon (Benoit Daloze)

- Description updated

#4 - 07/26/2022 04:40 PM - chucke (Tiago Cardoso)

> In the example, allowing to mutate @map in a Ractor is fundamentally unsafe...

The proposal is for mutex blocks to synchronize across ractors, thereby making it safe. This would IMO make for a better default than the current one, where mutexes aren't usable across ractors.

> In fact there are two in that example, @map.key? and @map[name] inside get_from_map (and it is not enough to only synchronize the writes).

You are correct, a simple early-check in the mutex block could have fixed that. It was nonetheless a "naive" illustration of a common pattern seen across a lot of gems, which just can't be replicated, or has to be massively overwritten, using ractors.

My proposals for ractors are all towards making them usable, and diminishing the adoption overhead. I too agree that they're largely unusable in production at the moment, and ractors should perhaps not aim at being so "pure" and follow the erlang processes approach of absolutely no shared memory, and go the way of pragmatism (like go) and provide some escape hatches which make the ruby standard library, as well as a significant chunk of the ecosystem, usable. Then it could focus on being lightweight, provide M:N concurrency, and all of those things that are probably in the roadmap.

I maintain a branch in my HTTP library, where I track ractor support, and I recently managed to perform an HTTP request inside a ractor. https still does not work, as well as most other features, and a few shortcuts were taken which are definitely not getting merged in the main branch, but still that was progress.

I'll keep using threads in production though, don't worry :)