Ruby master - Misc #14735

thread-safe operations in a hash could be documented

05/03/2018 06:55 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Status:	Open
Priority:	Normal
Assignee:	

Description

Hi, sometimes I find myself fetching data from the database through multiple queries concurrently. For example, suppose the application support multiple data-types which are independent from each other and we need to perform a set of operations per data-type. Usually I'd run one method for extracting the related data per data-type and would run them concurrently. Something like this:

```
require 'thread'
result = {} # assume this is thread-safe in MRI for now
data_types.map do |data_type, processor|
  Thread.start{ result[data_type] = processor.call }
end.each &:join
```

This code is quite simple and seems to always work with MRI. A more explicit equivalent code that should also work on other Ruby implementations without GIL would be probably written like:

```
require 'thread'
result = {}
result_semaphore = Mutex.new
data_types.map do |data_type, processor|
  Thread.start do
    result_for_data_type = processor.call # expensive call
    result_semaphore.synchronize{ result[data_type] = result_for_data_type }
  end
end.each &:join
```

As you can see, it's much more code than the previous one. As I said initially, I use such pattern every now and then, so I'd love to be able to write the first code and being confident that it would always work as expected in MRI.

I've tried the following in order to see if I could cause an thread-unsafe case with MRI but it always return "[100000, 100000, nil]":

```
require 'thread'
h = {}
(1..100000).map do |i|
  Thread.start{ h[i] = i }
end.each &:join

p h.keys.uniq.size, h.values.uniq.size, h.find{|k, v| k != v }
```

Is it just by chance? Or may I assume that will always be the case. Maybe it would be interesting to document somewhere what could be assumed to be true regarding thread-safeness for many methods. For example, there could be some link in the Hash documentation such as: "If you'd like to understand how each method behave in a multi-thread environment read this document" and point to another page explaining how it works.

By the way, the 'concurrent' gem seems to assume Hash is thread-safe in MRI as you can see here:

https://github.com/ruby-concurrency/concurrent-ruby/blob/master/lib/concurrent/hash.rb#L5-L16

```
module Concurrent
  if Concurrent.on_cruby?
    class Hash < ::Hash;
    end
  #...
  end
end
```

Is this officially documented somewhere?

History

#1 - 05/04/2018 12:22 AM - normalperson (Eric Wong)

rr.rosas@gmail.com wrote:

> result = {} # assume this is thread-safe in MRI for now
> data_types.map do |data_type, processor|
> Thread.start{ result[data_type] = processor.call }
> end.each &:join

It's only thread-safe in MRI if data_type is a common key type:

```
String/Symbol/Integer/nil/true/false/Float
```

> Is it just by chance?

Yes, probably...

> By the way, the 'concurrent' gem seems to assume Hash is thread-safe in MRI as you can see here:

'thread_safe' hit problems with that assumption:

```
https://bugs.ruby-lang.org/issues/14357
```

#2 - 05/04/2018 01:09 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Thanks, Eric. So, if I got this right, even though MRI tries to make it safe for common key types (it's a string in my case), I shouldn't rely on that as this would be considered an MRI implementation details and might change in the future. Or may I assume it would be always safe to assign values to different keys concurrently with MRI as long as I'd be using common keys (string/symbol/integer)?