Ruby - Feature #20875

Atomic initialization for Ractor local storage

11/06/2024 08:28 AM - ko1 (Koichi Sasada)

Status:	Closed
Priority:	Normal
Assignee:	
Target version:	

Description

Motivation

Now there is no way to initialize Ractor local storage in multi-thread.

For example, if we want to introduce per-Ractor counter, which should be protected with a per-Ractor Mutex for multi-threading support.

```
def init
  Ractor[:cnt] = 0
  Ractor[:mtx] = Mutex.new
end

def inc
  init unless Ractor[:cnt]
  Ractor[:mtx].synchronize do
    Ractor[:cnt] += 1
  end
end
```

In this code, if inc was called on multiple threads, init can be called with multiple threads and cnt can not be synchronized correctly.

Proposal

Let's introduce Ractor.local_storage_init(sym){ block } to initialize values in Ractor local storage.

If there is no slot for sym, synchronize with per-Ractor mutex and call block and the slot will be filled with the evaluation with the block result. The return value of this method will be the filled value.
Otherwise, returning corresponding value will be returned.

The implementation is like that (in C):

```
class Ractor
  def self.local_storage_init(sym)
    Ractor.per_ractor_mutex.synchronize do
      if Ractor.local_storage_has_key?(sym)
        Ractor[:sym]
      else
        Ractor[:sym] = yield
      end
    end
  end
end
```

The above examples will be rewritten with the following code:

```
def inc
  Ractor.local_storage_init(:mtx) do
    Ractor[:cnt] = 0
    Mutex.new
  end.synchronize do
    Ractor[:cnt] += 1
  end
end
```

Discussion

Approach

There is another approach like pthread_atfork, maybe like Ractor.atcreate{ init }. A library registers a callback which will be called when a new ractor is created.
However, there are many Ractors which don't use the library, so that atcreate can be huge overhead for Ractor creation.

Naming

I propose local_storage_init, but not sure it matches.
I also proposed Ractor.local_variable_init(sym), but Matz said he doesn't like this naming because it should not be a "variable".
(there is a Thread#thread_variable_get method, though).

On another aspect, lcoal_storage_init seems it clears all of ractor local storage slots.

Reentrancy

This proposal uses Mutex, so it is not reentrant. I believe it should be simple and using Monitor is too much.
(but it is not big issue, though)

Implementation

https://github.com/ruby/ruby/pull/12014

Associated revisions

Revision 0bdb38ba6be208064a514c12a9b80328645689f8 - 12/12/2024 09:22 PM - ko1 (Koichi Sasada)

Ractor.set_if_absent(key)

to initialize ractor local storage in thread-safety.
[Feature #20875]

Revision 0bdb38ba6be208064a514c12a9b80328645689f8 - 12/12/2024 09:22 PM - ko1 (Koichi Sasada)

Ractor.set_if_absent(key)

to initialize ractor local storage in thread-safety.
[Feature #20875]

Revision 0bdb38ba - 12/12/2024 09:22 PM - ko1 (Koichi Sasada)

Ractor.set_if_absent(key)

to initialize ractor local storage in thread-safety.
[Feature #20875]

History

#1 - 11/06/2024 08:28 AM - ko1 (Koichi Sasada)

- Description updated

#2 - 11/06/2024 08:29 AM - ko1 (Koichi Sasada)

- Description updated

#3 - 11/06/2024 08:31 AM - ko1 (Koichi Sasada)

- Description updated

#4 - 11/06/2024 08:51 AM - ko1 (Koichi Sasada)

- Description updated

#5 - 11/28/2024 07:48 AM - ko1 (Koichi Sasada)

@matz (Yukihiro Matsumoto) how about Ractor.local_storage_once(key){ ... }? It is from pthread_once.

Ractor.once(key){ ... } seems too short?

```
def inc
  Ractor.local_storage_once(:mtx) do
    Ractor[:cnt] = 0
    Mutex.new
  end.synchronize do
    Ractor[:cnt] += 1
  end
end
```

or

```
def inc
  Ractor.once(:mtx) do
    Ractor[:cnt] = 0
    Mutex.new
  end.synchronize do
    Ractor[:cnt] += 1
  end
end
```

#6 - 12/02/2024 04:15 PM - Dan0042 (Daniel DeLorme)

Would it be possible to make Ractor[:mtx] ||= Mutex.new behave in an atomic way? Like maybe add a special []||= method which is automatically called such that Ractor[:mtx] ||= Mutex.new becomes equivalent to Ractor[:mtx] || Ractor.send(:"[]||=", :mtx, Mutex.new)

I'm just throwing out the general idea here, because it would be nice to use common ruby idioms instead of yet another special API to handle concurrent behavior.

#7 - 12/04/2024 01:49 AM - ko1 (Koichi Sasada)

Dan0042 (Daniel DeLorme) wrote in #note-6:

> Would it be possible to make Ractor[:mtx] ||= Mutex.new behave in an atomic way?

On x[y()] ||= z(), z() can change the context and it violates atomicity.

#8 - 12/04/2024 04:09 PM - Dan0042 (Daniel DeLorme)

ko1 (Koichi Sasada) wrote in #note-7:

> On x[y()] ||= z(), z() can change the context and it violates atomicity.

Hmm, I don't think so? Of course for the regular ||= that would be the case, but in something like x.assign_if_unset(k(), v()) I don't see how v() can violate atomicity. In the worst case the result of v() would be thrown away.

But I'm talking here about a fairly general mechanism for atomic operations, so perhaps it's out of scope for this ticket.

#9 - 12/08/2024 06:51 AM - ko1 (Koichi Sasada)

Matz said that Ractor.local_storage_once(key){ init_block } can be acceptable if returning the assigned value is out-of-scope, even if it returns assigned value.

other ideas:

- Ractor.local_storage_fetch(key){} from Hash#fetch{}, but Hash#fetch doesn't set the block value, so it can lead wrong understanding.
- Ractor.local_storage_safe_init(key){} (by mame)
  - Ractor.local_storage_thread_safe_init(key){}
  - a bit long?

#10 - 12/09/2024 12:09 PM - Eregon (Benoit Daloze)

ko1 (Koichi Sasada) wrote in #note-9:

> can be acceptable if returning the assigned value is out-of-scope, even if it returns assigned value.

What does this mean?
It seems clear such a method should return the same as Ractor[key], after potentially computing + storing the value.

BTW, it sounds like Map#computeIfAbsent in Java and Concurrent::Map#compute_if_absent in concurrent-ruby.
Maybe Ruby should have Concurrent::Map built-in, or Hash#compute_if_absent built-in.
Then each Ractor could have such a map/hash and this could be solved like Ractor[:mtx] ||= Ractor.map.compute_if_absent(:mtx) { Mutex.new }.
In fact maybe that map could be the storage of ractor-local variables, and then Ractor.map.compute_if_absent(:mtx) { Mutex.new } is enough.

#11 - 12/12/2024 04:26 AM - ko1 (Koichi Sasada)

> Then each Ractor could have such a map/hash and this could be solved like Ractor[:mtx] ||= Ractor.map.compute_if_absent(:mtx) { Mutex.new }.

Ractor[:mtx] ||= is needed?

#12 - 12/12/2024 07:17 AM - ko1 (Koichi Sasada)

Ah, it assigs from Ractor.map.

#13 - 12/12/2024 07:58 PM - ko1 (Koichi Sasada)

quote from https://github.com/ruby/dev-meeting-log/blob/master/2024/DevMeeting-2024-12-12.md

- ko1: compute_if_absent terminology is introduced.
- matz: compute is not known in Ruby world. Ractor.store_if_absent(key){ init_block } is aceptable. I prefer than init/once.
- ko1Ractor.local_storage_store_if_absent(key){ init_block } (local_storage) is not needed?
- matz: too long.
- mame: atomicity (thread-safety) is not represented with this name. Is it okay?
- matz: no problem.

Conclusion:

- matz: Accept Ractor.store_if_absent(key){ init_block }

#14 - 12/12/2024 09:02 PM - Dan0042 (Daniel DeLorme)

"store_if_absent" is fairly verbose; I should point out that "add" is a common name for this operation. For example there's Set#add, and the memcached ADD command.

#15 - 12/12/2024 09:17 PM - ko1 (Koichi Sasada)

store is from Hash#store.

#16 - 12/12/2024 09:43 PM - ko1 (Koichi Sasada)

- Status changed from Open to Closed

Applied in changeset git|0bdb38ba6be208064a514c12a9b8032864568 9f8.

Ractor.set_if_absent(key)

to initialize ractor local storage in thread-safety.
[Feature #20875]