Ruby master - Feature #13893

Add Fiber#[] and Fiber#[]= and restore Thread#[] and Thread#[]= to their original behavior

09/12/2017 01:54 PM - cremes (Chuck Remes)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Ruby 3 API cleanup suggestion.

The Thread and Fiber classes have a very odd API for setting/getting thread local and fiber local variables. With Ruby 3 coming soon, this is a perfect opportunity to make this API more coherent and return to the Principal of Least Surprise. The concept of Fibers and Threads should be completely separated and we should no longer assume that a Fiber is attached to any particular Thread.

I suggest this:

```
class Fiber
  # Gets a fiber-local variable.
  def [](index)
    ...
  end

  # Sets a fiber-local variable.
  def []=(index, value)
    ...
  end

  # Returns true if the given +key+ exists as a fiber-local variable.
  def key?(key)
    ...
  end

  # Returns an array of fiber-local variable names as symbols.
  def keys
    ...
  end
end

class Thread
  # Gets a thread-local variable.
  def [](index)
    ...
  end

  # Sets a thread-local variable.
  def []=(index, value)
    ...
  end

  # Returns true if the given +key+ exists as a thread-local variable.
  def key?(key)
    ...
  end

  # Returns an array of thread-local variable names as symbols.
  def keys
    ...
  end
end
```

Also, remove Thread#thread_variable?, Thread#thread_variable_get, Thread#variable_set, and Thread#thread_variables since that behavior is already covered by Thread#key?, Thread#keys, Thread#[], and Thread#[]=. The APIs for both Thread and Fiber are more

coherent and less surprising with these changes.

History

#1 - 09/12/2017 04:20 PM - Eregon (Benoit Daloze)

I agree this would be much nicer and consistent.

I can't evaluate the impact on compatibility, but it's likely quite big.

#2 - 09/12/2017 04:35 PM - Eregon (Benoit Daloze)

Another idea which is related to #13245 would be to have these as class methods on Fiber and Thread (Thread[:foo]; Thread[:foo] = :bar; Thread.keys; Thread.key? :foo),
which would limit only accessing variables of the current fiber/thread.
It would allow to keep old methods for compatibility and maybe progressively deprecate them.

#3 - 09/14/2017 11:36 AM - shevegen (Robert A. Heiler)

I have nothing against the suggestion itself. However, there is no general, universal "Principal of Least Surprise".

To the topic, Threads and Fibers - I have used Threads sometimes but rarely. I haven't yet used Fibers. Without knowing any particular use case, I have a hard time trying to want to use something new.

For me, well aside from the suggestion here, I find this all way too complicated. Threads were actually quite simple... no idea how Fibers fit into anything there... or Mutex. I hope someone will simplify this all, no matter how. :)

As for ruby 3.0, I do not think it is "coming soon". Ruby 2.5 will come this year, I am not sure that ruby 3.0 will already be the very next ... matz said in one presentation that there is "tons of work" still about to happen for ruby 3, so I don't think that is all going to happen over night (if all the mentioned features that matz spoke about will go into Ruby 3.x, and I guess the core team wants the first ruby 3 release to be stable rather than unstable, which also will take a bit).

#4 - 09/14/2017 05:02 PM - cremes (Chuck Remes)

Ideally some of these changes could be rolled out in 2.x versions with deprecation notices for those methods whose behavior will change or those methods will be eliminated. For the 3.0 release, the deprecated methods should be removed entirely. Yes, these are breaking changes. Ruby3 is a wonderful opportunity to make the core APIs more consistent. If we miss this chance, it will be years (a decade?) before we get another chance to fix these inconsistencies.

#5 - 09/25/2017 12:45 PM - shyouhei (Shyouhei Urabe)

This sounds in fact cleaner. However I'm afraid it is too difficult to design a transition path. Fiber#[] could be okay but changing Thread#[] seems almost impossible... Can we warn users about such change in some way?

#6 - 11/30/2017 05:58 PM - cremes (Chuck Remes)

I have an alternative suggestion. Since shyouhei (Shyouhei Urabe) says that my original suggestion is too difficult a transition path, here is an alternative.

```
      # Convenience class to be used by Thread and Fiber. Those classes
      # have an odd way of dealing with thread-local and fiber-local variables.
      # This is a storage mechanism to replace the standard mechanism.
      #
      class Local
        extend Forwardable

        def_delegators :@storage, :[], :[]=, :key?, :keys

        def initialize
          @storage = {}
        end
      end

      # Done as a module so we can easily add it to a running Thread or Fiber via #extend.
      # e.g. add this to root Thread via Thread.main.extend(LocalMixin)
      #
      module LocalMixin
        def local
          @local ||= Local.new
        end
      end
```

This adds Thread#local and Fiber#local instance methods. We can then access them very easily:

```
Thread.current.local[:key] = 'thread local value'
```

```
Fiber.current.local[:key] = 'fiber local value'
```

This has several advantages:

- Cleans up the namespace by explicitly putting all "local" variables under the #local instance method accessor
- Does NOT break any existing code; the current thread and fiber local methods will continue to work
- We could potentially modify the Local class to use the Thread#thread_variable_set/get methods and the Thread#[]/[]= methods internally. This way users could access their local storage via either mechanism during the transition.

I can create a pull request with specs if anyone is interested in sponsoring this change.

#7 - 11/30/2017 09:11 PM - Eregon (Benoit Daloze)

cremes (Chuck Remes) wrote:

> I have an alternative suggestion. Since shyouhei (Shyouhei Urabe) says that my original suggestion is too difficult a transition path, here is an alternative.
> ...
> This adds Thread#local and Fiber#local instance methods. We can then access them very easily:
>
> ```
> Thread.current.local[:key] = 'thread local value'
> Fiber.current.local[:key] = 'fiber local value'
> ```

This looks good to me.

I think we can make it a bit nicer with:

```
Thread.local[:key] = 'thread local value'
Fiber.local[:key] = 'fiber local value'
```

It's a bit shorter and nicer to read for me.

But also with that API it discourages to get/set the thread/fiber-locals of another thread/fiber,
which is against the purpose of thread/fiber-local variables.
That API does not prevent it though, since one could save the value of Thread.local and use it in another thread
(we could enforce it with exceptions or have Thread/Fiber.local return an object always resolving the current Thread).
Sorry for bringing #13245 again but I think it is a very important concern for a new API: communicate the right usage.

One detail, where would we define this Local class?
::Local sounds too risky. Thread::Local or Fiber::Local sounds misleading.
Or maybe Thread.local and Fiber.local should just return a Hash instance for the current Thread/Fiber?

#8 - 12/01/2017 04:55 PM - cremes (Chuck Remes)

> One detail, where would we define this Local class?
> ::Local sounds too risky. Thread::Local or Fiber::Local sounds misleading.

The Local class is meant to be private. It's behavior should be the same/similar for both Fiber and Thread. Putting it under the Thread namespace makes sense to me since Fibers are also subordinate to Threads (i.e. they are locked to their creating thread, can't migrate between threads, cannot exist independent of a Thread, etc.).

Syntax:

```
  Fiber.local[:key] = 'fiber local value'
  Thread.local[:key] = 'thread local value'
vs
  Fiber.current.local[:key] = 'fiber local value'
  Thread.current.local[:key] = 'thread local value'
```

I understand why you prefer the first syntax since it is shorter. However, I think it obscures the ownership of the thread-local or fiber-local data when making it available via a class method. This local storage does NOT belong to the class; it belongs to an instance of the class. I think in this situation that requiring a few extra keystrokes is necessary to highlight the correct ownership of the data by making it required to access the data via an instance of Thread or Fiber.

Regarding discouraging modification of locals from a different thread/fiber, the Local class should be modified to capture the Thread.current and Fiber.current at the time of creation. This can be checked when accessing the local storage and throw an Exception when a different thread or fiber tries to modify local storage that it does not own. Good catch!

Here's some updated code:

```
  class Local
```

```
    def initialize
      @storage = {}
      @thread_creator = Thread.current
      @fiber_creator = Fiber.current
    end

    def [](key)
      ownership_check
      @storage[key]
    end

    def []=(key, value)
      ownership_check
      @storage[key] = value
    end

    def key?(key)
      ownership_check
      @storage.key?(key)
    end

    def keys
      ownership_check
      @storage.keys
    end

    private

    def ownership_check
      return if @thread_creator == Thread.current && @fiber_creator == Fiber.current
      raise ThreadError, "Access to local storage disallowed from non-originating thread or fiber!"
    end
  end
```

#9 - 08/08/2020 11:10 PM - ioquatix (Samuel Williams)

Also discussed here:

https://bugs.ruby-lang.org/issues/8215

#10 - 08/10/2020 08:40 AM - Eregon (Benoit Daloze)

One consideration: Fiber.current is not available until require "fiber".
So an API based on Fiber.current doesn't seem nice while require "fiber" is needed (will be NoMethodError + confusion if not required).

I think it would be too incompatible to change Thread.current#[]/[]= to anything else.
It would need first a deprecation (but so many usages it's going to be very annoying), then removal for some years, then adding back as thread-locals.
As such Fiber#[]/[]= don't seem nice, because they would just be redundant with Thread.current#[]/[]= or make it even more confusing.

The .local API seems nice and avoid those issues.

Fiber.local wouldn't need any check, but Fiber.current.local would need checks.
That's suboptimal as Fiber.current.local will need to lookup the current Fiber/Thread twice instead of once.
On Rubies without a GIL or with Ractors, Thread.current and Fiber.current will be on its their some kind of (native) thread-local lookup.
Also Fiber.current.local needs Fiber.current has the require "fiber" issue mentioned above.

#11 - 08/10/2020 08:44 AM - Eregon (Benoit Daloze)

From #7 actually the checks would be needed in both cases, because one could save the value of Thread.current.local and pass it to another Thread.

So either .local API seems fine to me, but if it's Fiber.current.local then Fiber.current should become core.

#12 - 08/14/2020 01:48 AM - ioquatix (Samuel Williams)

Just one more potential interface:

```
class Fiber
  attr_accessor :my_fiber_local
end

class Thread
  attr_accessor :my_thread_local
end
```

To me, this actually seems like it should be the most logical way to add well defined/documented fiber and thread locals.

However, I also like the idea of Fiber[] and Fiber[]= as well as Thread[] and Thread[]=. However, given that in both cases keys/attribute names can clash, I don't see either one having a significant advantage. However, attributes could at least have warnings if clobbering existing attributes.