Ruby - Feature #19078

Introduce `Fiber#storage` for inheritable fiber-scoped variables.

10/22/2022 09:56 PM - ioquatix (Samuel Williams)

Status:	Closed
Priority:	Normal
Assignee:	ioquatix (Samuel Williams)
Target version:	

Description

Pull Request: https://github.com/ruby/ruby/pull/6612

This is an evolution of the previous ideas:

- https://bugs.ruby-lang.org/issues/19058
- https://bugs.ruby-lang.org/issues/19062

This PR introduces fiber scoped variables, and is a solution for problems like https://github.com/ioquatix/ioquatix/discussions/17.

The main interface is:

```
Fiber[key] = value
Fiber[key] # => value
```

The variables are scoped (local to) a fiber and inherited into child fibers and threads.

```
Fiber[:request_id] = SecureRandom.hex(16)

Fiber.new do
  p Fiber[:request_id] # prints the above request id
end
```

The fiber scoped variables are stored and can be accessed:

```
Fiber.current.storage # => returns a Hash (copy) of the internal storage.
Fiber.current.storage= # => assigns a Hash (copy) to the internal storage.
```

Fiber itself has one new keyword argument:

```
Fiber.new(..., storage: hash, false, undef, nil)
```

This can control how the fiber variables are setup in a child context.

To minimise the performance overhead of some of the implementation choices, we are also simultaneously implementing https://bugs.ruby-lang.org/issues/19077.

Examples

Request loop

```
Thread.new do
  while request = queue.pop
    Fiber.new(storage: {id: SecureRandom.hex(16)}) do
      handle_request.call(request)
    end
  end
end
```

OR

```
Thread.new do
  while request = queue.pop
    Fiber.current.storage = {id: SecureRandom.hex(16)}
```

```
    handle_request.call(request)
  end
end
```

Related issues:

Related to Ruby - Feature #19058: Introduce `Fiber.inheritable` attributes/va...	**Closed**
Related to Ruby - Feature #19062: Introduce `Fiber#locals` for shared inherit...	**Closed**
Related to Ruby - Feature #19141: Add thread-owned Monitor to protect thread-...	**Open**

History

#1 - 10/22/2022 09:56 PM - ioquatix (Samuel Williams)

- Tracker changed from Bug to Feature

- Backport deleted (2.7: UNKNOWN, 3.0: UNKNOWN, 3.1: UNKNOWN)

#2 - 10/22/2022 10:31 PM - ioquatix (Samuel Williams)

- Description updated

#3 - 10/22/2022 10:33 PM - ioquatix (Samuel Williams)

- Description updated

#4 - 10/24/2022 11:07 PM - tenderlovemaking (Aaron Patterson)

I think this would be very helpful in Rails where we want to keep track of a "global" database connection. Right now the database connection is a thread local, but that doesn't work for webservers that use Fibers as a concurrency model. This seems like a good way to set and release resources that are global to each request (but only for that request).

#5 - 10/24/2022 11:09 PM - tenderlovemaking (Aaron Patterson)

(Sorry, I hit submit too soon)

Currently Rails applications have to be configured with a particular concurrency model, which they do via the "Isolated Execution State". You can see the code here.

I would really like it if the user (and also us Rails maintainers) didn't need to care about what type of concurrency strategy the webserver uses.

#6 - 10/26/2022 10:21 AM - byroot (Jean Boussier)

I'm generally in favor of this feature. Lots of Ruby code rely on "global" state (generally implemented as fiber locals) so whenever you spawn a new fiber or thread you lose all that context which generally breaks code.

Things I think should be added to the ticket description:

Inheritance

A fiber storage is initialized as a shallow-copy of the parent fiber. Overwriting a key in the child, doesn't impact the parent, think of it as POSIX environment variables:

```
Fiber[:request_id] = SecureRandom.hex(16) # aaaaa

Fiber.new do
  p Fiber[:request_id] # prints aaa
  Fiber[:request_id] = SecureRandom.hex(16) # bbbb
  p Fiber[:request_id] # prints bbbb
end
p Fiber[:request_id] # prints aaaa
```

However it is a shallow copy, so if you mutate one of the value, you may end up with shared mutable state, but that's on you.

What keys can be used?

Symbols only like Thread#[] ? Would seem sensible to me.

Do we need the same API on Thread?

Could be useful for variables that are thread scoped and should be inherited too. Problem is Thread#[] is already used for fiber-local variables.

Overall I agree with @tenderlove, this would make Rails' IsolatedExecutionState obsolete, which is good, and if used properly would save the classic issue of code breaking in iterator fibers because some fiber local state was lost.

#7 - 10/26/2022 08:47 PM - ioquatix (Samuel Williams)

> What keys can be used?

Does Ruby provide any "Hash" -> "Symbol Table" object or mapping/conversion? Otherwise we have to do it ourselves which is O(N) vs O(1) for hash dup (CoW) internally. So it will be worse performance to force all keys to be symbols (not better).

However, if Ruby has some kind of "Symbol Table" that we can trust, it is possible to be better performance, slightly.

In addition, I've seen some questionable code as a result of this design decision, things like:

```
def thing
  Thread.current[:"#{self.class}_#{self.object_id}"] ||= ...
end
```

I feel like this encourages bad practices because people are forced to convert objects to the hash.

Maybe a better option would be to require the hash to be compare-by-identity or equivalent. This should be equally fast, and from the PoV of the internal storage, equivalent to using a symbol table (now it's literally just a pointer comparison).

#8 - 10/26/2022 08:49 PM - ioquatix (Samuel Williams)

> Do we need the same API on Thread?

> Could be useful for variables that are thread scoped and should be inherited too. Problem is Thread#[] is already used for fiber-local variables.

This API does not intersect with any current Thread APIs, like Thread[] or Thread[]= or Thread.current.storage.

However, I'm not sure what you are expecting to be different. Are these just aliases for the fiber equivalent? The storage is contained within the execution context, for the current thread, it's always the current fiber.

#9 - 10/27/2022 10:15 AM - byroot (Jean Boussier)

> I feel like this encourages bad practices because people are forced to convert objects to the hash.

Often times, this is done to avoid holding a reference to the object in the thread, effectively making it immortal.

If we take the example of the connection_pool gem, I'm not sure how you'd do this with an identity hash:
https://github.com/mperham/connection_pool/blob/428c06f34209ee7a99f1ace5b96e567841c00d1c/lib/connection_pool.rb#L97

> However, I'm not sure what you are expecting to be different. Are these just aliases for the fiber equivalent?

My question was whether we should have a similar inheritable store but that is thread local rather than fiber local. But let's forget it for now.

#10 - 10/28/2022 01:06 AM - ioquatix (Samuel Williams)

> Often times, this is done to avoid holding a reference to the object in the thread, effectively making it immortal.

I think the correct way to make opaque per-instance keys is something like this:

```
class ConnectionPool
  def initialize
    @connection_key = Object.new
    @count_key = Object.new
  end
```

```
  def checkout(options = {})
    if connection = ::Fiber[@connection_key]
      ::Fiber[@count_key] += 1
    else
      connection = ::Fiber[@connection_key] = @available.pop(options[:timeout] || @timeout)
      ::Fiber[@count_key] = 1
```

```
    end

    return connection
  end
end
```

I would say the above implementation is invalid, because inheriting the count and connection is likely to cause buggy behaviour, however the key point is, using Object.new for opaque keys should be totally fine.

#11 - 11/08/2022 03:07 PM - Eregon (Benoit Daloze)

- Related to Feature #19058: Introduce `Fiber.inheritable` attributes/variables for dealing with shared state. added

- Related to Feature #19062: Introduce `Fiber#locals` for shared inheritable state. added

#12 - 11/08/2022 03:48 PM - Eregon (Benoit Daloze)

This design looks good to me.

For the keys I would prefer Symbol only, just like for fiber/thread-locals. Consistency is good there.
Compare-by-identity seems acceptable too.
One concern is compare-by-identity is more expensive and complicated on TruffleRuby for primitives like int/long which can represent the same value.
Compare-by-identity also means using String with this new APIs would be a bug, which seems unfortunate, also because fiber/thread-locals accept strings.

I want to represent inheritable fiber-scoped variables interally as Ruby objects using Shapes in TruffleRuby.
I think CRuby should do the same.
That means there is no need for the large complexity of copy-on-write Hash (and cost on maintenance), and the actual copying is faster, it's just copying an array of values and done.
COW is also weird perf-wise as it penalizes the first write significantly (the first write is O(n)).
Access is also faster, a Shape check + a read/write vs hash lookups which have many more indirections and work to do.

Also I feel the whole API is easier to understand (e.g. mutating the Hash has no effect) if we detach the external representation (Hash) from the internal one (object with shape).

Regarding threads I think we shouldn't inherit automatically in new threads, and rather do it explicitly (via Fiber.current.storage=) in the rare cases it's needed.

Fiber#{storage,storage=} should only be allowed to be called on the current Fiber, and raise an exception if that's not the case.

#13 - 11/08/2022 09:04 PM - byroot (Jean Boussier)

> Compare-by-identity also means using String with this new APIs would be a bug

Agreed, that's what weirds me out a bit. It's quite a big downside.

The upside if for objects that want to store instance state in fiber/thread local data (e.g. the ThreadSafeLevel example), but ultimately this is workable with the current "hack" @key = :"#{self.class.name}_#{object_id}".

> That means there is no need for the large complexity of copy-on-write Hash (and cost on maintenance)

I think they can potentially be a performance improvement, but not to thread storage specifically. So I'd like to see them implemented, but IMO it's entirely orthogonal to this issue.

> Regarding threads I think we shouldn't inherit automatically in new threads, and rather do it explicitly (via Fiber.current.storage=) in the rare cases it's needed.

I'm on the fence on this one. Usually the code spawning a thread, and the code using thread/fiber storage and needing it to be inherited are entirely decoupled. So you'd need to go convince all your dependencies that spawn threads (e.g. puma, sidekiq, etc) to add that one line of code. So I'd prefer if it was always implicitly copied.

Also from my point of view Thread.new creates both a new thread and a new fiber, so implicitly it's akin to Fiber.new and should inherit.

> Fiber#{storage,storage=} should only be allowed to be called on the current Fiber, and raise an exception if that's not the case.

I get what you want that for truffle, but I fear this may be an hindrance to debugging and introspection. I would like to be able to run Thread.list.map(&:storage) to inspect the state of my process.

storage= from another fiber is definitely a big no-no, but is there really no way Fiber#storage can be synchronized on Truffle, even if that means it's

slower?

#14 - 11/09/2022 12:31 AM - marcotc (Marco Costa)

> Regarding threads I think we shouldn't inherit automatically in new threads, and rather do it explicitly (via Fiber.current.storage=) in the rare cases it's needed.

I'm on the fence on this one. Usually the code spawning a thread, and the code using thread/fiber storage and needing it to be inherited are entirely decoupled. So you'd need to go convince all your dependencies that spawn threads (e.g. puma, sidekiq, etc) to add that one line of code. So I'd prefer if it was always implicitly copied.

For cross-cutting concerns, like telemetry, automatic inheritance works best; asking a gem user to add one extra line per Fiber created would create room for error.

I think thinking about the opposite use case, users that explicitly want a clean Fiber storage for newly created Fibers, would help here: how common is such use case? I can't think of a reason to have this as the default, except for saving on the performance cost of copying the storage.

#15 - 11/21/2022 07:58 PM - Eregon (Benoit Daloze)

- Related to Feature #19141: Add thread-owned Monitor to protect thread-local resources added

#16 - 11/22/2022 12:20 PM - Eregon (Benoit Daloze)

I've been reading https://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part2.html#scope-variables again and parts of the JEP.
For that approach to work it really needs structured concurrency (well explained here).

One issue is we don't currently have a standard way to express structured concurrency in Ruby.
Even though that could be very useful, also for e.g. #19141.
Maybe it could be something like creating a "concurrency scope" using a block like:

```
structured_concurrency do |c|
  c << Fiber.new { ... }
  c << Thread.new { ... }
  # Fiber/Thread could also maybe automatically register in c without `c <<`
end
```

and that at the end of the block would wait that every Fiber/Thread created in there finishes (which would mean calling resume repetitively for Fibers, and join for Threads).
I guess Async has some similar concepts, right? cc @ioquatix (Samuel Williams)

From my previous comment in https://bugs.ruby-lang.org/issues/19062#note-29 in this issue we are defining inheritable fiber locals.
I think that makes sense and in some cases it's probably useful the storage is available longer than the parent Fiber lives (even though that can be dangerous too).
I just wonder if we might want structured concurrency and related concepts too in Ruby at some point.

#17 - 11/22/2022 05:32 PM - ioquatix (Samuel Williams)

Async already has the concept of structured concurrency.

> I've been reading https://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part2.html#scope-variables again and parts of the JEP.

The inspiration for our model is dynamic binding/scope (similar to defined in LISP), and there is a lot of overlap.

> One issue is we don't currently have a standard way to express structured concurrency in Ruby.

We can already write it like this:

```
Fiber.schedule do
  # Create child tasks.
  Fiber.schedule {}
  Fiber.schedule {}
```

> we are defining inheritable fiber locals

Do you want to rename it from storage to locals?

> I think that makes sense and in some cases it's probably useful the storage is available longer than the parent Fiber lives (even though that can be dangerous too).

Can you explain the cases? I can't think of any.

> I just wonder if we might want structured concurrency and related concepts too in Ruby at some point.

Is the above nested fiber good enough or not?

#18 - 11/23/2022 10:54 AM - Eregon (Benoit Daloze)

ioquatix (Samuel Williams) wrote in #note-17:

> We can already write it like this:
>
> ```
> Fiber.schedule do
>   # Create child tasks.
>   Fiber.schedule {}
>   Fiber.schedule {}
> ```

That doesn't wait for the child tasks at the end of the parent task, does it?
If it doesn't wait it's not structured concurrency (children outlive the parent).

> Is the above nested fiber good enough or not?

No due to that, but also that code only applies to nonblocking Fibers, but for structured concurrency I think it should apply to all children Fibers and Threads.

> Do you want to rename it from storage to locals?

No, because they are not really local (they are shared between different Fibers). I used that term because Java calls them InheritableThreadLocal.

> > I think that makes sense and in some cases it's probably useful the storage is available longer than the parent Fiber lives (even though that can be dangerous too).
>
> Can you explain the cases? I can't think of any.

That's interesting, I thought you had use cases for this.
IIRC you once mentioned maybe some kind of background job which then can hold on the current user (or current user id) or so.
If we don't have use cases for this, maybe we should try to enforce these variables are used in a structured way (but it sounds difficult).

#19 - 11/23/2022 07:55 PM - ioquatix (Samuel Williams)

> If it doesn't wait it's not structured concurrency (children outlive the parent).

The problem with being strict is that there are some cases where you need the child to outlive the parent. So, strict structured concurrency is insufficient for some concurrency models, and in general works best for map-reduce style problems which is only a subset of all possible asynchronous execution models.

I agree, structured concurrency is nice, so Async has support for "wait for these children tasks to complete".

> No, because they are not really local (they are shared between different Fibers)

That's not a strict distinction. Anyone can write this:

```
storage = Hash.new
Thread.current[:storage] = storage
Thread.new do
  Thread.current[:storage] = storage
end
```

Just because the data is shared doesn't mean it's not local too.

Anyway.. just my 2c on the matter.

> That's interesting, I thought you had use cases for this.

Yes, plenty, any kind of persistent connection which needs an active ping/pong or background message processing. I just thought since you brought it up you had some specific examples in mind.

#20 - 11/29/2022 09:56 PM - ivoanjo (Ivo Anjo)

marcotc (Marco Costa) wrote in #note-14:

> > Regarding threads I think we shouldn't inherit automatically in new threads, and rather do it explicitly (via Fiber.current.storage=) in the rare cases it's needed.
>
> I'm on the fence on this one. Usually the code spawning a thread, and the code using thread/fiber storage and needing it to be inherited are entirely decoupled. So you'd need to go convince all your dependencies that spawn threads (e.g. puma, sidekiq, etc) to add that one line of code. So I'd prefer if it was always implicitly copied.

For cross-cutting concerns, like telemetry, automatic inheritance works best; asking a gem user to add one extra line per Fiber created would create room for error.

I think thinking about the opposite use case, users that explicitly want a clean Fiber storage for newly created Fibers, would help here: how common is such use case? I can't think of a reason to have this as the default, except for saving on the performance cost of copying the storage.

To a bit of info on top of this (and disclaimer, I work with Marco on the ddtrace gem, one really nice property of the automatic inheritance is that it makes easy something that is otherwise quite hard to do automatically from regular Ruby code. E.g. to simulate such an automatic mechanism, one needs to monkey patch thread and fiber creation, which is really awkward and error-prone (ask me how I know this).

#21 - 11/30/2022 05:11 PM - Dan0042 (Daniel DeLorme)

Maybe I'm too late here, but I have some thoughts/concerns about this.

First I should say I totally agree with the general idea. We need some way to inherit "context state".

So far I've seen 2 use cases described.
1 - store a request_id or correlation_id, for logging purposes (or even the full request object)
2 - store a connection (typically database) that can't be used at the same time by another thread/fiber
It would be nice if we had 1-2 other use cases; that would allow to evaluate this design from more angles.

We already have per-thread-but-actually-fiber storage, and a really-per-thread storage, and this is adding a third type. What about when we want per-ractor storage... a fourth type? Per-ractor inheritable storage... a fifth type? Rather than fragmenting storages I would prefer to unify them. Maybe it's a crazy idea, but what if we set fiber-inheritable values via Thread.current.store(k, v, level: Fiber, inherit: true) and read them via Thread.current[k] ? For writing it's a somewhat complex/advanced interface, but choosing which storage to use is a complex/advanced question anyway. And when reading a value I think we typically don't care about which storage it is in; it would be easier to have a single point of access.

When should a value be inherited? For usecase#1 (request_id), it seems to be all the time. But for usecase#2 (connection) I think it varies. If you have a sub-fiber that does its own requests to the DB, you don't want to inherit the connection. But if it's an iterator fiber you do want to inherit the connection. So it depends on what "kind" of fiber, how it is used. Actually, for an iterator fiber, you'd probably want to inherit all state, even if it wasn't explicitly specified as inheritable. So maybe something like Fiber.new(inherit: true) would be better for that case.

What about when transfering control from one fiber to another?

```
producer = Fiber.new{loop{
  puts "on behalf of #{Thread.current[:request_id] || '?'}"
  puts "producing #{v=rand}"
  Fiber.yield(v)
}}
Thread.current[:request_id] = SecureRandom.hex(16)
producer.resume
#on behalf of ?
#producing 0.8068165204559555
```

In the code above we could argue that the producer should "inherit" the request_id via resume.

#22 - 12/01/2022 04:59 AM - matz (Yukihiro Matsumoto)

This proposal contains 4 features:

(1) fiber[key]/fiber[key]=val - accepted.
(2) fiber.storage => hash - accepted
(3) fiber.storage=hash - should be experimental
(4) Fiber.new(...,storage: hash|true|false|nil) - accepted, but true/false should be experimental

The feature (3) can be bigger side effect to clear third-party storage, so we need some experiment.
The feature (4) with true/false are hard to read the intention (true to inherit with dup, false to inherit without dup), so we need experiment here as well.

Matz.

#23 - 12/01/2022 08:44 AM - ioquatix (Samuel Williams)

Thanks so much for your time and discussion @matz (Yukihiro Matsumoto) et al.

#24 - 12/01/2022 10:05 AM - ioquatix (Samuel Williams)

- Status changed from Open to Closed

It was merged.

#25 - 12/02/2022 09:34 AM - ioquatix (Samuel Williams)

I made a compatibility shim for older Rubies. It's not 100% perfect, but it's close enough to be useful.

https://github.com/ioquatix/fiber-storage

#26 - 12/02/2022 03:00 PM - nevans (Nicholas Evans)

I'm also a bit late to this. While I do have some nitpicks, I'll leave them for another ticket (if I ever get around to it). But, to answer a couple of @Dan0042's questions (and maybe others):

As mentioned by @Eregon (Benoit Daloze) above, one of the most important use-cases for this is structured concurrency. If you try to implement structured concurrency, you will eventually wind up needing and implementing some version of this feature. If you look at the documentation and API for go's and kotlin's versions of this feature, it's overwhelmingly about structured concurrency. @Eregon (Benoit Daloze) included a good link describing structured concurrency, above. If I remember correctly, this page is also a good introduction to the idea: https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/

Here's a short list of the same basic feature, as implemented in several other languages (I copied a couple of these links from the other tickets linked in the description):

- go's context package
  - Maybe the simplest implementation of the idea?
- Kotlin's Coroutine Contexts
- python's contextvars (integrated with asyncio, etc)
  - See PEP 567 (and PEP-550 which preceded it) for more info on design details, etc.
- JEP 429: Extent-Local Variables

This ticket is implemented a little bit differently than these, but is basically in the same category and has many of the same motivations, etc.

#27 - 12/02/2022 04:23 PM - nevans (Nicholas Evans)

@Dan0042 (Daniel DeLorme) wrote in #note-21:

> What about when we want per-ractor storage... a fourth type?

FWIW, Ractor#[] has existed since 3.0: https://docs.ruby-lang.org/en/3.0/Ractor.html#method-i-5B-5D :)

> Per-ractor inheritable storage... a fifth type?

In my opinion, absolutely not. The following is just a hypothetical--if and when I have an actual proposal, I'll post it in another ticket. :)

This feature is brand-new so it has some "rough edges", but I think should be updated to automatically share all ractor-sharable values. Of course, the big question is: what to do for non-ractor-sharable values? Hypothetically:

1. By default, simple raise an exception from Ractor.new, just like it does any other time you accidentally attempt to send non-sharable state into a new Ractor.
2. Add a kwarg to Ractor.new, and a method to the fiber storage, both of which will simply drop all non-ractor sharable storage. Either way: non-inheritance should always be an explicit opt-out.
3. Add an API for more fine-grained control at certain points, so that library authors can designate different behaviors via callback. Because often the desired behavior is more complicated than simply dropping or inheriting a variable: e.g. rather than inherit the current task id, a new fiber/thread/ractor might automatically create a brand new task which is a child of the current task.

For inspiration here, we might look to Kotlin, which delegates a great deal of functionality to the current coroutineContext[CoroutineDispatcher] (this would be handled entirely by library code, and invisible to most users), to coroutineContext[Job] (which combines with CoroutineScope and a few others to handle almost everything related to structured concurrency), and any ThreadContextElement that's been added to the context. That last one is mostly used for backward compatibility with libraries that expect thread-locals.

Similarly, go contexts delegate internally to context.Value(cancelContextKey), where cancelContextKey is a private package var which is used purely for comparison-by-identity. This is essentially the @key = Object.new trick demonstrated by @ioquatix (Samuel Williams) in #note-10, and it's the recommended usage pattern for context keys in go.

Anyway: my goal here isn't to propose any particular approach, but to show this feature might be used as a foundation for other future features. IMO, fine-grained inheritance control might be out-of-scope for 3.2 (at least, any API we put together now would need to be marked "experimental"). I simply wanted to show how other languages build on a similar foundation to handle the problems you mentioned.

#28 - 12/20/2022 01:42 PM - Iloeki (Loic Nageleisen)

This is really nice for some use cases, and I was planning to propose basically exactly what @ioquatix (Samuel Williams) proposed here to tackle these.

One example: we create context objects that allow us to track what happens across a given HTTP request execution. Typically with Rack we can store that in the Rack environment, but it so happens that elements we instrument via module prepending may not have access to the Rack environment. To get the context we then have to store that context object in a thread local (which is actually fiber local). The consequence is that if some bit code then goes async/concurrent (another thread or fiber) then context is (of course) lost. This would allow to keep the context within which we want to track things. As it turns out one of our team members from Sqreen implemented AsyncLocalContext which is incredibly similar in its purpose: https://nodejs.org/api/async_context.html

Another example: Rails is doing unhappy things to thread locals when using ActionController::Live: it blindly copies thread locals to the new thread in an attempt to have the new thread behave like the old one. This is not good as it assumes none of these thread locals would be concurrently accessed or mutated, when by design thread locals are supposed to be, well, local to threads, thus can eschew thread safetiness safe nets. With this feature, Rails may be in a position to use it for what it needs and maybe stop its blind copying.

#29 - 12/20/2022 02:53 PM - Iloeki (Loic Nageleisen)

for the sake of thoroughness the nodejs one above was inspired by .Net
https://learn.microsoft.com/en-us/dotnet/api/system.threading.asynclocal-1?view=net-7.0

#30 - 12/20/2022 04:49 PM - Eregon (Benoit Daloze)

matz (Yukihiro Matsumoto) wrote in #note-22:

> (4) Fiber.new(...,storage: hash|true|false|nil) - accepted, but true/false should be experimental
> The feature (4) with true/false are hard to read the intention (true to inherit with dup, false to inherit without dup), so we need experiment here as well.

false is a violation of everything we discussed (with @byroot (Jean Boussier) etc).
It means no more isolation between Fibers, and setting a variable accidentally changes other Fiber storages, leading to tons of confusion.
I believe no language does this for very good reasons.
I don't know when you added this @ioquatix (Samuel Williams) but that is an unacceptable change to the design long discussed.

Also there is no experimental warning or anything as matz has asked.

I will remove storage: false now, this is the only safe thing to do before the release.
Also this is a necessary condition to avoid synchronization on Fiber[]/Fiber[]= for non-GIL Rubies.

BTW, Fiber#storage and Fiber#storage= do not check which thread calls them.
But a different thread should of course never be allowed to reassign the storage of a Fiber it doesn't belong to. Accessing it also feels wrong.

#31 - 12/20/2022 05:31 PM - Eregon (Benoit Daloze)

Fixes for the above are in https://github.com/ruby/ruby/pull/6972

> Also there is no experimental warning or anything as matz has asked.

There is some mention in the docs, but I think that's not enough, especially for Fiber#storage= so I added an experimental warning there.

#32 - 12/20/2022 06:22 PM - Eregon (Benoit Daloze)

Eregon (Benoit Daloze) wrote in #note-30:

> Also this is a necessary condition to avoid synchronization on Fiber[]/Fiber[]= for non-GIL Rubies.

Correction: that is not the case since at least storage cannot be shared between threads.

But it's still a problem for all the other reasons (EDIT: https://bugs.ruby-lang.org/issues/19062#note-28 is a good one).
These are fiber-scoped variables, hence it should never be possible to affect a parent fiber by a change in a child fiber (otherwise that just becomes global state).
If it's wanted to change something and have all fiber referencing it to see it, it's very easy, just an use an indirection and not the fiber-scoped variable itself.

#33 - 12/20/2022 09:33 PM - Eregon (Benoit Daloze)

I had call with @ioquatix (Samuel Williams), we agreed the changes of https://github.com/ruby/ruby/pull/6972 are OK.

It's unclear if Fiber#storage and Fiber#storage= need to be allowed to be called for a different Fiber of the same thread (e.g., might be useful for a debugger).
One concern is if Fibers become no longer fixed to a given Thread in the future then this condition wouldn't mean anything anymore, and we would have the issue of accessing state being concurrently mutated in another thread (on Rubies without GIL).
I think for debugging we should have a new API to "transfer" to a Fiber but not execute any code of that Fiber, just whatever the debugger wants to run there (e.g. get the backtrace, the stack, local variables, get the fiber locals, etc). A bit like thread N in gdb.

#34 - 12/21/2022 01:41 PM - Eregon (Benoit Daloze)

A good use-case for Fiber#storage=: https://github.com/ruby-concurrency/concurrent-ruby/issues/456#issuecomment-1361325461