Ruby - Feature #19141

Add thread-owned Monitor to protect thread-local resources

11/21/2022 05:50 PM - wildmaples (Maple Ong)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Background

In Ruby v3.0.2, Monitor was modified to be owned by fibers instead of threads for reasons as described in this issue and so it is also consistent with Mutex. Before the change to Monitor, Mutex was modified to per-fiber instead of thread (issue, PR) which caused some problems (See: comment).

Problem

We are now encountering a problem where using Enumerator (implemented transparently using Fiber, so the user is not aware) within a Fiber-owned process, which causes a deadlock. That means any framework using Monitor is incompatible to be used with Enumerator.

In general, there are many types of thread-local resources (connections for example), so it would make sense to have a thread-owned monitor to protect them. I think few resources are really fiber-owned.

Specifics

- Concurrent Ruby is still designed with per-thread locking, which causes similar incompatibilities. (Read: issue)
- Systems test in Rails implements locking using Monitor, resulting in deadlock in these known cases:
  - when cache clearing (Read: issue)
  - database transactions when used with Enumerator (Read: comment)

Demo

```
# ruby 2.7.6p219 (2022-04-12 revision c9c2245c0a) [arm64-darwin21]
# Thread #<Thread:0x000000014a8eb228 demo.rb:8 run>, fiber #<Fiber:0x000000014a8eaf80 (resumed)>, locked true, owned true
# Thread #<Thread:0x000000014a8eb228 demo.rb:8 run>, fiber #<Fiber:0x000000014a8eacb0 demo.rb:13 (resumed)>, locked true, owned true

# ruby 3.1.2p20 (2022-04-12 revision 4491bb740a) [arm64-darwin21]
# Thread #<Thread:0x0000000102329a08 demo.rb:8 run>, fiber #<Fiber:0x0000000102329828 (resumed)>, locked true, owned true
# Thread #<Thread:0x0000000102329a08 demo.rb:8 run>, fiber #<Fiber:0x00000001023294e0 demo.rb:13 (resumed)>, locked true, owned false

require 'fiber'
require 'monitor'

puts RUBY_DESCRIPTION

# We have a single monitor - we're pretending it protects some thread-local resources
m = Monitor.new

# We'll create an explicit thread
t = Thread.new do
  # Lock to protect our thread-local resource
  m.enter

  puts "Thread #{Thread.current}, fiber #{Fiber.current}, locked #{m.mon_locked?}, owned #{m.mon_owned?}"

```

```
# The Enumerator here creates a fiber, which runs on the same thread, so would want to use the sam
e thread-local resource
  e = Enumerator.new do |y|
    # In 2.7 this is fine, in 3.0 it's not, as the fiber thinks it doesn't have the lock
    puts "Thread #{Thread.current}, fiber #{Fiber.current}, locked #{m.mon_locked?}, owned #{m.
mon_owned?}"

    # This would deadlock
    # m.enter

    y.yield 1
  end
  e.next
end

t.join
```

Possible Solutions

- Allow Monitor to be per thread or fiber through a flag
- Having Thread::Monitor and Fiber::Monitor as two separate classes. Leave Monitor as it is right now. However, this may not be possible due to the Thread::Mutex alias

These options would give us more flexibility in which type of Monitor to use.

Related issues:

Related to Ruby - Feature #19078: Introduce `Fiber#storage` for inheritable f...	**Closed**
Related to Ruby - Feature #16792: Make Mutex held per Fiber instead of per Th...	**Closed**

History

#1 - 11/21/2022 07:58 PM - Eregon (Benoit Daloze)

- Related to Feature #19078: Introduce `Fiber#storage` for inheritable fiber-scoped variables. added

#2 - 11/21/2022 08:00 PM - Eregon (Benoit Daloze)

- Description updated

#3 - 11/21/2022 10:02 PM - byroot (Jean Boussier)

@Eregon (Benoit Daloze) how does this relate to Fiber#storage?

#4 - 11/21/2022 10:04 PM - Eregon (Benoit Daloze)

> That means any framework using Monitor is incompatible to be used with Enumerator.

It's only problematic if both the parent Fiber and the Enumerator-using-Fiber try to acquire the same Monitor while the other holds it.

> In general, there are many types of thread-local resources (connections for example), so it would make sense to have a thread-owned monitor to protect them. I think few resources are really fiber-owned.

Whether a resource is per Thread or Fiber does not depend on the resource, the resource typically needs to be protected from concurrent access.
What it depends on is whether there is a Fiber scheduler and so whether Fibers are used for concurrency.
For example, a Fiber might hold the resource or lock while transferring to another Fiber, and that can cause big problems for the resource.
And this transfer can happen on any blocking IO call with the Fiber scheduler so basically anywhere.

Based on that I don't think a per-thread Monitor is really a solution, the fibers of that thread might not always be nicely nested or release the monitor after they are done executing, and so they can cause the same problem as a global resource or no Monitor.

Also from an implementation point of view it seems really hard if not impossible to implement a per-Thread monitor on the JVM, where the concurrency unit is java.lang.Thread and the new Loom VirtualThreads inherits from java.lang.Thread and so every lock is "per Fiber" in Java too now.
BTW, Mutex has always been per-Fiber on JRuby and TruffleRuby due to that.
My intuition is locks always need to be "per finest concurrency unit" (but I might be wrong).
OTOH for storage both fiber-local, thread-local and inherited fiber-scoped (#19078) make sense. All of these except fiber-local need their own synchronization otherwise they could be mutated concurrently.

That said, we do need to find solutions to the various issues you linked.
For concurrent-ruby I think the solution is relatively clear based on my last comment.

For https://github.com/rails/rails/issues/45994#issuecomment-1319687582 it seems to me a per-thread Monitor isn't really the right solution.
What if someone creates a Thread inside that transaction, should that also work?
Fibers with a Fiber scheduler are basically similar to threads, except fibers of a given threads don't execute at the same time, but they are a form of concurrency and they can switch at fairly unpredictable places.

I do remember someone mentioning an issue with Enumerator-using-Fiber and transactions in a Rails app. Enumerator-using-Fiber is kind of problematic for that because it doesn't necessarily respect the transaction length and might also e.g. keep the connection longer than expected.

In general I think there are very few use-cases for Enumerator-using-Fiber (which is Enumerator#{peek*,next*} and #zip with non-arrays).
They are expensive and they don't execute on the same stack and so any state they need and which should only be held only for some period is really difficult to manage.
People do use it every now and then though so it'd be nice to find a solution.

#5 - 11/22/2022 05:10 AM - ioquatix (Samuel Williams)

Why do you need a recursive mutex for this code? Why not just lock inside the enumerator around whatever shared resource you are protecting?

http://www.zaval.org/resources/library/butenhof1.html

> 1. The biggest of all the big problems with recursive mutexes is that they encourage you to completely lose track of your locking scheme and scope. This is deadly. Evil. It's the "thread eater". You hold locks for the absolutely shortest possible time. Period. Always. If you're calling something with a lock held simply because you don't know it's held, or because you don't know whether the callee needs the mutex, then you're holding it too long. You're aiming a shotgun at your application and pulling the trigger. You presumably started using threads to get concurrency; but you've just PREVENTED concurrency.

#6 - 11/22/2022 08:46 AM - byroot (Jean Boussier)

> Based on that I don't think a per-thread Monitor is really a solution

In the case of the original issue they are with Rails connection, they don't use a Fiber scheduler, they just use an enumerator for control flow which happens to use a Fiber.

If you use a fiber scheduler with Rails, you are supposed to change a config that makes the connection pool hand different connections to different fibers.

But they don't do that, they use threads, so they should be able to use fibers for control flow with a single connection as they're guaranteed not to have concurrent access.

> What if someone creates a Thread inside that transaction, should that also work?

No that wouldn't and that shouldn't. You can't compare fibers and threads here. Fibers are not preemptible so it's totally valid to use them for control flow with a shared resource (if you wish).

> Fibers with a Fiber scheduler are basically similar to threads

Yes, but again no fiber scheduler here. Just coroutines.

> Why do you need a recursive mutex for this code?

```
Post.transaction do # while the transaction is active we should prevent other threads from using it.
  Post.first # we need to recursively acquire the transaction
end
```

#7 - 11/22/2022 11:18 AM - Eregon (Benoit Daloze)

It's not only an issue with Fiber scheduler although that is the most obvious way to illustrate.
After all Fiber scheduler is not the only usage of resume/transfer/yield, any gem or app might use those as well and not always in clearly-defined places / in a structured concurrency manner.

Suppose we have a Thread::Monitor, in the example below (with no Fiber scheduler) we end up in two bad cases:

- holding the lock after the .synchronize{} returned
- unlocking the lock preemptively, potentially breaking the resource

How do we address this with Thread::Monitor?
With Fiber-based Monitor there is a clear exception as soon as entering f.

```
M = Thread::Monitor.new

f = Fiber.new {
  M.synchronize {
    # using protected resource
    Fiber.yield # or other_fiber.transfer
    # using protected resource
  }
}

def transaction(m)
  t = Transaction.new
  M.synchronize {
    yield
  }
ensure
  t.commit_or_abort
end

transaction {
  f.resume
}
# M is still held by the main Thread (or conceptually by f or by the main Fiber), that could cause confusing d
eadlocks

transaction {
  f.resume
  # M is not held here!
  # now another Fiber/Thread could corrupt the resource
  some_more_logic
}
```

#8 - 11/22/2022 11:22 AM - byroot (Jean Boussier)

- Related to Feature #16792: Make Mutex held per Fiber instead of per Thread added

#9 - 11/22/2022 11:26 AM - byroot (Jean Boussier)

> How do we address this with Thread::Monitor?

If you are trying to say that one can shoot themselves in the foot with a ThreadMonitor and explicit fibers, then yes that's true.

But these folks had it working for years (and still working since this currently prevents them to upgrade to 3.x), so to me this feels like a valid use case that was broken by the change and that would be worth supporting.

#10 - 11/22/2022 01:40 PM - Eregon (Benoit Daloze)

I had a call with @byroot (Jean Boussier) about this issue.

First of all, in the cases it's possible it's best to avoid Enumerator#{next*,peek*} as that avoids this issue, but also passing extra state to it, etc.
If the Enumerator is used for e.g. batched queries or DB accesses, a good way is to reimplement those with manual Enumerators/"iterators", which don't need a Fiber.

If we introduce something like a per-thread monitor, I think it's very important to track the fibers which locked it in a stack, and only allow unlock by the Fiber who locked last.
That addresses most of the concerns of my example above and guarantees the mutex/monitor is dealt with in a structured way and that the critical sections being synchronize'd can actually be meaningful.

The problem is such a per-thread monitor would still not let Enumerator-with-Fiber work on a Fiber-based server like Falcon.
So it is an incomplete solution, but it may still be helpful.

I have also commented on https://github.com/rails/rails/issues/45994#issuecomment-1323691630 with more details regarding the Rails issue.
I think the real solution is not a per-thread monitor but one of the other solutions mentioned there, which work also with Fiber-based servers and in all conditions.

#11 - 11/22/2022 05:24 PM - chrisseaton (Chris Seaton)

Maybe Enumerator should not use a full fiber - but something less that behaves like the thread it's run on? An underlying issue is that the user doesn't know they're using a fiber, and really how can we expect them to?

#12 - 11/23/2022 11:09 AM - Eregon (Benoit Daloze)

chrisseaton (Chris Seaton) wrote in #note-11:

> Maybe Enumerator should not use a full fiber - but something less that behaves like the thread it's run on?

It still needs its own stack so it can suspend anywhere.
For Enumerators created by e.g. Array#each and other Enumerable methods without block for a core enumerable object it may be fine to not have its own Fiber, but for Enumerator.new or with a custom (not core) each that's much less clear, because there can be arbitrary logic in such an Enumerator, notably those database accesses in that Rails issue. And in that case it might not be correct to share the fiber locals, the owned locks, use the same Fiber for Fiber.current, etc.
It would also mean Fiber.current inside that Enumerator could be changing (if Enumerator#next is called by different Fibers).

> An underlying issue is that the user doesn't know they're using a fiber, and really how can we expect them to?

We should document that in Enumerator#{next*,peek*} clearly and the pitfalls of it.
There is already a mention of it but it's no obvious (See class-level notes about external iterators.).