Ruby - Feature #13821

Allow fibers to be resumed across threads

08/16/2017 05:19 PM - cremes (Chuck Remes)

Status:	Assigned	
Priority:	Normal	
Assignee:	ko1 (Koichi Sasada)	
Target version:		

Description

Given a Fiber created in ThreadA, Ruby 2.4.1 (and earlier releases) raise a FiberError if the fiber is resumed in ThreadB or any other thread other than the one that created the original Fiber.

Sample code attached to demonstrate problem.

If Fibers are truly encapsulating all of the data for the continuation, we should be allowed to move them between Threads and resume their operation.

Why?

One use-case is to support the async-await asynchronous programming model. In that model, a method marked async runs *synchronously* until the #await method is encountered. At that point the method is suspended and control is returned to the caller. When the #await method completes (asynchronously) then it may resume the suspended method and continue. The only way to capture this program state, suspend and resume, is via a Fiber.

example:

```
class Wait
  include AsyncAwait

  def dofirst
    async do
      puts 'Synchronously print dofirst.'
      result = await { dosecond }
      puts 'dosecond is complete'
      result
    end
  end

  def dosecond
    async do
      puts 'Synchronously print dosecond from async task.'
      slept = await { sleep 3 }
      puts 'Sleep complete'
      slept
    end
  end

  def run
    task = dofirst
    puts 'Received task'
    p AsyncAwait::Task.await(task)
  end
end

Wait.new.run

# Expected output:
# Synchronous print dofirst.
# Received task
# Synchronously print dosecond from async task.
# Sleep complete
# dosecond is complete
# 3
```

Right now the best way to accomplish suspension of the #dofirst and #dosecond commands and allow them to run asynchronously is by passing those blocks to *another thread* (other than the callers thread) so they can be encapsulated in a new Fiber and then yielded. When it's time to resume after #await completes, that other thread must lookup the fiber and resume it. This is lots of extra code and logic to make sure that fibers are only resumed on the threads that created them. Allowing Fibers to migrate between threads would eliminate this problem.

History

#1 - 09/05/2017 05:43 PM - kernigh (George Koehler)

Fibers still can't move across threads in

```
ruby 2.5.0dev (2017-09-04 trunk 59742) [x86_64-openbsd6.1]
```

Because of this, I can't take an Enumerator across threads:

```
count = 1.step
puts count.next    #=> 1
puts count.next    #=> 2
Thread.new {
  puts count.next  #=> FiberError
}.join
```

If Ruby would allow fibers to cross threads, then it might be possible with only some platforms. I find that Ruby (in cont.c) has three different ways for fibers.

1. It uses CreateFiber/SwitchToFiber in Microsoft Windows.
2. It uses makecontext/swapcontext in some POSIX systems (but not NetBSD, Solaris, Hurd).
3. It uses continuations in all other platforms.

Each fiber needs its own stack for C code. With continuations, each fiber continues on the stack of its thread. When Ruby switches fibers, it copies their stacks to and from the thread stack. C code can make pointers to the stack, so the address of the stack can never change. With continuations, if Ruby resumes a fiber on a different thread, then it would copy the fiber stack to a different thread stack, the address would change, and C code would crash. Therefore, fibers can't cross threads in platforms using continuations.

I don't know whether fibers can cross threads in platforms using CreateFiber or makecontext. I also don't know whether Ruby can garbage-collect a thread that created fibers that crossed to other threads.

#2 - 09/12/2017 12:17 PM - Eregon (Benoit Daloze)

I think this is first of all a problem for semantics.

If we allow fibers to be resumed on another Thread, we allow multiple fibers originally from the same thread to execute concurrently
(so they no longer see the effects of each other perfectly but are exposed to race conditions like threads).

It also means before and after Fiber.yield, the value of Thread.current can change if the Fiber is resumed on another Thread.
This in turns breaks Fiber-locals with the current Thread.current[key] API.

It's also problematic for locks and other resources which are per-thread (some of them native so they cannot be tricked to use the initial Thread of the Fiber):

```
Fiber.new { shared = Object.new; lock.synchronize { shared.a += 1; Fiber.yield; shared.b -= 1 } }
```

The unlock operation will fail because it's on a different thread than the lock operation if the fiber is resumed on another thread.

#3 - 09/12/2017 01:44 PM - cremes (Chuck Remes)

I understand how this request could allow for race conditions between Fibers. Right now we are relying on the fact that they can only run on a single thread to enforce this particular semantic. I also agree that this is useful in certain situations.

But it is still quite useful to allow for Fibers to migrate between threads. Perhaps we could allow for both possibilities with a minor change to the Fiber API.

```
class Fiber
  def initialize(migrate: false)
    ...
  end

  def [](index)
    ...
  end

  def []=(index, value)
    ...
  end
```

```
end
```

By default we would retain the existing behavior where the Fiber is "locked" to its originating Thread. But if you call Fiber.new(migrate: true) then the Fiber is free to float among multiple threads. When doing so, the programmer is *explicitly* agreeing to no longer rely upon the original semantics. If they yield a fiber inside of a synchronized section then they understand it will likely break if resumed on another thread. Likewise, they do not rely upon Thread#[] and related methods to set/get fiber locals.

That Thread API for fiber locals is broken anyway... the #[] and #[]= methods on Thread should set/get thread locals as they did originally. There should be Fiber#[] and Fiber#[]= methods on the Fiber class. Conflating the two separate concepts all into the Thread class is no good. With Ruby 3 on the way this is the perfect time to fix problems like that. I'll open a separate ticket to suggest that as an improvement to the Thread and Fiber classes.

#4 - 09/12/2017 01:56 PM - cremes (Chuck Remes)

Added ticket 13893 (https://bugs.ruby-lang.org/issues/13893) to track a feature request to cleanup fiber-local and thread-local handling in the Fiber and Thread classes.

#5 - 09/12/2017 04:32 PM - Eregon (Benoit Daloze)

cremes (Chuck Remes) wrote:

> By default we would retain the existing behavior where the Fiber is "locked" to its originating Thread. But if you call Fiber.new(migrate: true) then the Fiber is free to float among multiple threads. When doing so, the programmer is *explicitly* agreeing to no longer rely upon the original semantics. If they yield a fiber inside of a synchronized section then they understand it will likely break if resumed on another thread. Likewise, they do not rely upon Thread#[] and related methods to set/get fiber locals.

This would give up on using these fibers with any library using Mutex, Thread.current and similar Thread primitives if any of these is used across a Fiber.yield.
It's a considerable cost for reusing code.

Could you share the code you have for the implementation of AsyncAwait?

#6 - 09/15/2017 01:40 PM - cremes (Chuck Remes)

Yes, the Fiber.new(migrate: true) would mean the programmer is taking responsibility for NOT wrapping that Fiber up in mutexes or relying on the default behavior. I think this is reasonable.

As for the async/await code I've written, it hasn't been published yet. I can shoot you a tarball if you want to look at it but it's still alpha quality (no tests). I'll ping you on the TruffleRuby project to get your email.

#7 - 09/15/2017 04:08 PM - cremes (Chuck Remes)

I took a look at the C++ Boost library boost::fiber documentation. It allows fibers to be detached/attached between threads. Perhaps an explicit API like this is a better approach? See here: http://www.boost.org/doc/libs/1_62_0/libs/fiber/doc/html/fiber/migration.html

This puts the responsibility onto the programmer to Fiber#detach from its current thread and Fiber#attach(thread) to a new thread. The limitation is that a Fiber cannot be moved if it is *blocked* or if it is currently running.

By making the detach/attach explicit, then the programmer is assuming 100% responsibility to make sure the fiber hasn't yielded while holding locks or other operations that assume the Fiber is locked to a thread.

#8 - 09/25/2017 12:41 PM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Assigned

- Assignee set to ko1 (Koichi Sasada)

In the today's developer meeting Ko1 said that migrating fibers across threads is currently not possible. I think he would like to explain why, so let me assign this issue to him.

#9 - 02/21/2018 06:39 AM - ko1 (Koichi Sasada)

Sorry for long absent.

The point is the gap between native-thread and Ruby's thread/fibers.

Ruby can use C-extensions and some C-extensions depends on external libraries.
A few external libraries can depend on (native) thread local storage and we can't observe it.
There may be other possible issues on it because of this semantics gap.

We choose a conservative solution.

If we can control everything, we can make Fibers across multiple threads (like gorotine, Win32's Fiber and so on).

#10 - 06/06/2018 10:30 PM - bascule (Tony Arcieri)

Eregon (Benoit Daloze) wrote:

> It's also problematic for locks and other resources which are per-thread (some of them native so they cannot be tricked to use the initial Thread of the Fiber):
>
> ```
> Fiber.new { shared = Object.new; lock.synchronize { shared.a += 1; Fiber.yield; shared.b -= 1 } }
> ```
>
> The unlock operation will fail because it's on a different thread than the lock operation if the fiber is resumed on another thread.

There's a simple solution to this: track if a given fiber is holding mutexes (e.g. keep a count of them) and if it is, make Fiber#resume raise an exception if it is resumed in a different thread from the one where it was originally yielded.

That way you eliminate the nasty edge case, but still allow fibers which aren't holding mutexes (or whatever other synchronization primitives you're worried about) to be resumed in a different thread.

The same solution could work for thread-local storage: disallow fiber cross-thread fiber resumption if thread local storage is in use.

#11 - 01/27/2019 08:28 PM - shan (Shannon Skipper)

bascule (Tony Arcieri) wrote:

> There's a simple solution to this: track if a given fiber is holding mutexes (e.g. keep a count of them) and if it is, make Fiber#resume raise an exception if it is resumed in a different thread from the one where it was originally yielded.

I'd love this. I've been frustrated by not being able to share simple Enumerators across Threads.

#12 - 02/15/2019 10:09 AM - shan (Shannon Skipper)

If automatic detection of whether a Fiber is shareable across Threads isn't viable, it would be really, really nice to have a Fiber.new(migrate: true)-like option that could also be enabled for Enumerators.

#13 - 12/28/2023 12:32 PM - ioquatix (Samuel Williams)

One issue was the use of the copy-coroutine. As we've removed this and replaced it with a pthread fallback, we can now potentially allow fibers to be resumed across threads. I'll put together a PR so we can experiment.

#14 - 12/28/2023 01:05 PM - Eregon (Benoit Daloze)

bascule (Tony Arcieri) wrote in #note-10:

> There's a simple solution to this: track if a given fiber is holding mutexes (e.g. keep a count of them) and if it is, make Fiber#resume raise an exception if it is resumed in a different thread from the one where it was originally yielded.
>
> That way you eliminate the nasty edge case, but still allow fibers which aren't holding mutexes (or whatever other synchronization primitives you're worried about) to be resumed in a different thread.

This may work for ::Mutex, but it won't work for any other synchronization.
For example any of the locks/barriers/countdown latch in concurrent-ruby.
Or even locks in C extensions.

> The same solution could work for thread-local storage: disallow fiber cross-thread fiber resumption if thread local storage is in use.

I think this will be so limiting that it makes creating a Fiber which can migrate between threads almost always useless in practice as such a Fiber can't e.g. call a gem as that might use thread-local storage or synchronization internally.

@shan

Regarding Enumerator, only next/peek create a Fiber, all other methods are fine and do not need a Fiber.
Using an Enumerator in a next/peek way from multiple threads is something that is fundamentally very tricky, and that Ruby currently does not really have a concept to support it.
Basically you'd want a thread-safe Java Iterator(#hasNext/#next), but there is no such protocol in Ruby.
That's really tricky, and few data structures can actually support that.
I think it has very limited value, because next/peek is rare, and doing next/peek in threads concurrently seems asking for concurrency bugs and data races.
A Queue seems a much better thing to use to get some elements on one thread and some on another.

@ioquatix (Samuel Williams)

> I'll put together a PR so we can experiment.

This proposal breaks Fiber semantics so fundamentally I think if such a feature is added it should have another name than Fiber, to make it clear it can migrate threads and is subject to all thread races.
In fact since it would have the exact same concurrency semantics as Thread, so I think it should include thread in the name.

BTW, now CRuby has M-N threads (#19842), is there any reason to try this over using RUBY_MN_THREADS=1?

One could also simulate this by creating every Fiber in a Thread with RUBY_MN_THREADS=1 as threads should be cheaper that way.
That also avoids all the problems of a Fiber changing threads dynamically, while not restricting to run a Fiber on a given thread.

Migrating a Fiber to another threads seems to also have little value on CRuby due to the GVL, as it won't improve parallelism.

#15 - 12/28/2023 09:20 PM - ioquatix (Samuel Williams)

> This proposal breaks Fiber semantics so fundamentally

I don't disagree with you, and I don't have a strong opinion one way or another, but there is nothing about coroutine semantics which prevents them from being passed between threads which is what the MN implementation is already doing (it uses the same coroutines). So, I'd like to experiment with it.

Alternatively, maybe we can re-implement Enumerator with Thread+Queue which is more flexible, at least that deals with one of the problems that is often brought up in these discussions.

Files

fiber_across_threads.rb	377 Bytes	08/16/2017	cremes (Chuck Remes)
wait.rb	728 Bytes	08/16/2017	cremes (Chuck Remes)