Ruby trunk - Feature #14736

Thread selector for flexible cooperative fiber based concurrency

05/04/2018 03:12 AM - ioquatix (Samuel Williams)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

Ruby concurrency can be greatly enhanced by concurrent, deterministic IO.

Fibers have been shown many times to be a great abstraction for this purpose. The retain normal code flow and don't require any kind of Thread synchronisation. They are enjoyable to write code with because you don't have to concern yourself with thread synchronisation or other complicated issues.

The basic idea is that if some operation would block, it yields the Fiber, and other Fibers within the thread can continue to execute.

There are a number of ways to implement this. Here is a proof of concept to amend the existing rb_io_wait_readable/rb_io_wait_writable.

https://github.com/ruby/ruby/pull/1870

This design minimally affects the Ruby implementation and allows flexibility for selector implementation. With a small amount of work, we can support EventMachine (65 million downloads), NIO4r (21 million downloads). It would be trivial to back port.

This PR isn't complete but I am seeking feedback. If it's a good idea, I will do my best to see it through to completion, including support for EventMachine and NIO4r.

Related issues:

Related to Ruby trunk - Feature #13618: [PATCH] auto fiber schedule for rb_wa...	**Assigned**

History

#1 - 05/05/2018 02:51 AM - ioquatix (Samuel Williams)

As part of this, I've started working on https://bugs.ruby-lang.org/issues/14739

#2 - 06/12/2018 03:57 AM - ioquatix (Samuel Williams)

I've been playing around with port scanners. Implemented in Go (goroutines), Python (asyncio) and Ruby (async).

I wrote up the results here: https://github.com/socketry/async-await/tree/master/examples/port_scanner

It was just an attempt to gauge the performance of the different implementations. It's by no means an authoritative comparison.

What I found interesting was that Ruby (async) was faster than Python (async) by about 2x. Go was faster again by about 2x. However, Go can use multiple CPU cores, and so because it utilised ~5 hardware threads, it was in effect about 10x faster.

I found that quite fascinating.

I don't believe we can easily adopt a model like goroutines in Ruby. However, it might be possible to adapt some of the good ideas from it.

#3 - 06/13/2018 01:03 AM - normalperson (Eric Wong)

- File port_scanner_threadlet.rb added

samuel@oriontransfer.net wrote:

> I've been playing around with port scanners. Implemented in Go
> (goroutines), Python (asyncio) and Ruby (async).
>
> I wrote up the results here:
> https://github.com/socketry/async-await/tree/master/examples/port_scanner

Attached is the implementation for Threadlet/auto-fiber/wachamacallit
rebased against ruby trunk r63641:

https://80x24.org/spew/20180613003524.9256-1-e@80x24.org/raw

On a busy Linux VM, Threadlet was close to your Go implementation
in speed (timing results were unstable, however) and Ruby async
was around 3x slower behind (even with timing instabilities).

I kept on getting errors with the Python3 version
("Event loop is closed") so I never let it finish

I needed to deal with EPIPE because the system I tested on had RDS (16385)
enabled in the kernel which was triggering EPIPE (I don't know Go or Python):

```
diff --git a/examples/port_scanner/port_scanner.go b/examples/port_scanner/port_scanner.go
index 45f2d1c..ad0f049 100755
--- a/examples/port_scanner/port_scanner.go
+++ b/examples/port_scanner/port_scanner.go
@@ -55,7 +55,7 @@ func checkPortOpen(ip string, port int, timeout time.Duration) {
        } else if strings.Contains(err.Error(), "refused") {
            // fmt.Println(port, "closed", err.Error())
        } else {
-           panic(err)
+           fmt.Println(port, "err", err.Error())
        }
        return
    }
diff --git a/examples/port_scanner/port_scanner.py b/examples/port_scanner/port_scanner.py
index 372f0b3..ca9d41a 100755
--- a/examples/port_scanner/port_scanner.py
+++ b/examples/port_scanner/port_scanner.py
@@ -22,6 +22,8 @@ class PortScanner:
 # print("{} closed".format(port))
except asyncio.TimeoutError:
print("{} timeout".format(port))
+            except SystemError:
+                print("{} error".format(port))

def start(self, timeout=1.0):
self.loop.run_until_complete(asyncio.gather(
diff --git a/examples/port_scanner/port_scanner.rb b/examples/port_scanner/port_scanner.rb
index 0e4160e..3ac0109 100755
--- a/examples/port_scanner/port_scanner.rb
+++ b/examples/port_scanner/port_scanner.rb
@@ -25,6 +25,8 @@ class PortScanner
 # puts "#{port} closed"
rescue Async::TimeoutError
puts "#{port} timeout"
+  rescue SystemCallError => e
+    puts "#{port} #{e.message}"
end

async def start(timeout = 1.0)
```

#4 - 07/04/2018 08:08 AM - ioquatix (Samuel Williams)

normalperson (Eric Wong) that's awesome, great effort! I really appreciate you taking these PRs seriously and the effort you are putting into it.

I'd love to have a systematic comparison. Honestly, the port scanner isn't a great benchmark because port scanning is either limited by system resources (scanning localhost) or heavily network bound (scanning remote system).

Considering that async is pure ruby on a single thread, 3x slower is actually pretty good. If we merge this PR (Thread.selector), I can probably improve performance quite a bit.

Your implementation of Threadlet could be easily implemented in terms of the PR suggested here. But this PR allows flexibility for other implementations (like NIO4r). I still think that flexibility is pretty important, especially considering it will allow JRuby to implement it without too many changes.

The benefit of stackful coroutines is they make blocking completely transparent. If you look at the go code, it's a lot more complex, requires synchronisation, etc. The feedback I had about async was it was super easy to use. My priority is good API followed by performance.

Honestly, I really liked the performance of the go code, and the design is really great, but I don't think it's good for general purpose computing. Most people will find the required locking/synchronisation too complex.

#5 - 07/04/2018 08:17 AM - ioquatix (Samuel Williams)

Some of the benefits of this PR are:

- Makes it possible to implement different models for concurrency.
- Easy to implement by other Ruby implementations.
- Baseline interface for implementing scheduling, on which other schedulers can be constructed.
- Minimal changes to MRI code (less bugs/less maintenance).
- Doesn't introduce any new class, only one new public attr.

#6 - 07/04/2018 04:39 PM - funny_falcon (Yura Sokolov)

Just remark: make test example to use Fiber.transfer.
If patch will be accepted, example will be copied, and usage of Fiber.yield is not composable with Enumerator.

#7 - 07/05/2018 04:12 AM - ioquatix (Samuel Williams)

Are you saying that calling Fiber.yield is not valid within an enumerator?

#8 - 07/05/2018 07:25 AM - funny_falcon (Yura Sokolov)

Yes. While usually Enumerator doesn't call to Fiber.yield, it is called if Enumerator is used as external iterator:

```
> def aga; yield 1; Fiber.yield 4; yield 8; end
> to_enum(:aga).to_a
Traceback (most recent call last):
        6: from /usr/bin/irb:11:in `<main>'
        5: from (irb):76
        4: from (irb):76:in `to_a'
        3: from (irb):76:in `each'
        2: from (irb):68:in `aga'
        1: from (irb):68:in `yield'
FiberError (can't yield from root fiber)
> e = to_enum(:aga)
=> #<Enumerator: main:aga>
> e.next
=> 1
> e.next
=> 4
> e.next
=> 8
> [:a,:b,:c].each.zip(to_enum(:aga))
=> [[:a, 1], [:b, 4], [:c, 8]]
```

#9 - 07/05/2018 07:48 AM - ioquatix (Samuel Williams)

For the first case, you naturally can't call Fiber.yield in that context... but this works:

```
#!/usr/bin/env ruby

require 'fiber'

fiber = Fiber.new do
    def aga; yield 1; Fiber.yield 4; yield 8; end
    puts to_enum(:aga).to_a
end

value = fiber.resume
puts "Got #{value}"
while fiber.alive?
    fiber.resume
end

# Prints:
# Got 4
# 1
# 8
```

This also works:

```
#!/usr/bin/env ruby

require 'fiber'

fiber = Fiber.new do
    def aga; yield 1; Fiber.yield 4; yield 8; end
    e = to_enum(:aga)

    puts e.next
```

```
    puts e.next
    puts e.next
end

value = fiber.resume
puts "Got #{value.inspect}"
while fiber.alive?
    fiber.resume
end

# 1
# 4
# 8
# Got nil
```

The semantics of Fiber.yield from within the enumerator block depend on how it's being used. That's not something I was aware of. I see that it fails if you try to do this:

```
#!/usr/bin/env ruby

$LOAD_PATH << File.expand_path("../../lib", __dir__)

require 'async'
require 'async/io/stream'
require 'async/io/host_endpoint'
require 'async/io/protocol/line'

class Lines < Async::IO::Protocol::Line
    def each
        return to_enum unless block_given?

        while line = read_line
            yield line
        end
    end
end

input = Lines.new(
    Async::IO::Stream.new(
        Async::IO::Generic.new($stdin)
    )
)

Async.run do
    # This works:
    # input.each do |line|
    #   puts "... #{line}"
    # end

    # This doesn't:
    enumerator = input.each
    while line = enumerator.next
        puts "... #{line}"
    end
end
```

I can imagine, if you don't know the underlying task is asynchronous, that this might cause some frustration, fortunately the error is reasonably clear in this instance.

The problem is the nested fiber is not created in an Async::Task. I believe that with this PR in place, it would be possible to avoid this, because ALL Fiber created on the thread with a selector is asynchronous, so it shouldn't be a problem. I am interested in working further on this PR, and this is an interesting test case. Perhaps I will see if it can work or not.

#10 - 07/05/2018 08:33 AM - funny_falcon (Yura Sokolov)

I've shown to_enum(:aga).to_a to present the place where I wasn't right.

But if you look at your own second example, you will see that it doesn't do what it should do
(if Fiber.yield is replaced with yield point of your scheduler, for example, with task.sleep(0.01) or .socket.read),
because yield point should not affecte.next`.

And you've already shown this in third example.

Scheduler should use Fiber.transfer, because it is really scheduling primitive in its nature,
and because it is almost unknown and almost nowhere used.

#11 - 07/05/2018 08:36 AM - ioquatix (Samuel Williams)

Yes, I agree with what you say, and I agree with your conclusion, I was just giving an example where it failed with async which highlights the issue :)

#12 - 07/08/2018 02:02 AM - ioquatix (Samuel Williams)

I have updated the PR to use transfer in the scheduler, and I've added an example showing that it is composable with Enumerator.

#13 - 05/01/2019 02:33 PM - Eregon (Benoit Daloze)

- Related to Feature #13618: [PATCH] auto fiber schedule for rb_wait_for_single_fd and rb_waitpid added

#14 - 05/01/2019 03:11 PM - Eregon (Benoit Daloze)

We discussed this a bit at the Ruby developer meeting before RubyKaigi and matz (Yukihiro Matsumoto) asked me to share my opinion on the bug tracker.

I also attended ioquatix (Samuel Williams)'s talk at RubyKaigi and it was quite convincing.

Basically, the main outcome for me is that we can do pretty much all of "AutoFiber" #13618, with much less complexity and a completely trivial patch (see https://github.com/ruby/ruby/pull/1870/files).
With those hooks on "waiting for IO", we can do the scheduling entirely in Ruby and we have a lot more flexibility on what we want to do.
We don't need to introduce a third kind of mix of Fiber and Thread, we just reuse Fiber and Fiber#transfer.
We can simply use nio4r for efficient polling of file descriptors which is already portable.

AutoFiber (#13618) sounds nice, but it's only one model with no extension points or flexibility, and it's a lot of code to maintain in each Ruby implementation.
There are no silver bullets for concurrency, flexibility is key to best adapt to the case at hand.

If the concern is to propose a concrete concurrency model for Ruby 3, I think it's fine to show examples on how to do it with external gems (e.g., nio4r, async).
That just shows we live as a Ruby community.

There are open issues for both proposals, e.g., how to handle disk IO such as File.read (which always reports "ready" by select() but might take a while on a large file), Mutex, ConditionVariable#wait, Queue#push and other blocking operations.
For instance, it sounds natural to schedule another Fiber in Mutex#lock if the Mutex is already locked, and then we'd need some way to notify the scheduler once the Mutex is unlocked to schedule the first Fiber.

Also, should we try to handle IOs which are not explicitly set as #nonblock=true?
Maybe we should make all sockets and pipes nonblock by default? They don't seem to be in 2.6.2.
Or is that intended as a mechanism so that scheduling is only triggered on IOs manually set as nonblock?

One question is how can we scope this functionality so that Fiber scheduling only happens in code where it's intended?
That's related to atomicity and concurrency concerns in the AutoFiber thread.

The selectors from the PR are per-Thread, but maybe we want something finer-grained.
I think the async gem shows one way to do that with a block: Async { ... } from https://github.com/socketry/async#async
Since we can control the selectors/scheduler in Ruby, it's easy to tune this and for instance introduce a construct to only perform blocking calls without scheduling inside a given block.
In other words, this API gives us the power to fine-tune scheduling so that we can ensure the semantics wanted for a given application.

#15 - 05/01/2019 03:15 PM - Eregon (Benoit Daloze)

In case it was not clear, I'm in favor of this change for Ruby 3.
I think it provides the essential primitives to build something powerful and flexible for efficient asynchronous IO in Ruby.

#16 - 05/01/2019 10:41 PM - ioquatix (Samuel Williams)

> how to handle disk IO such as File.read

While many platforms don't support non-blocking disk IO, liburing for linux and IOCP on Windows might present some opportunities for increased concurrency. Alternatively, some implementations (e.g. libuv) use a thread pool for blocking disk IO. So, I believe it should be up to the selector to decide how to handle it.

> The selectors from the PR are per-Thread, but maybe we want something finer-grained.

So, there is no reason why someone couldn't implement a thread-safe selector and assign it on multiple threads, or assign to Thread#selector multiple times depending on context. That's how async works using thread locals.

#17 - 05/01/2019 11:03 PM - ioquatix (Samuel Williams)

> Also, should we try to handle IOs which are not explicitly set as #nonblock=true?

I guess in my proof of concept I wanted to do the bare minimum to show if it would work. But using io/nonblock is a bit of an ugly hack, so I personally vote to make nonblock the default when it's running in a selector. However, there ARE some situations when you want blocking IO, so I'm not sure what is the best approach. It also doesn't work well with File.read and similar things.

Personally, from the user POV, non-blocking vs blocking shouldn't be a concern they have to deal with.

For example, think about Rack middleware. When running in a non-blocking context, all IO should ideally be non-blocking to maximise concurrency. It shouldn't require user to opt in explicitly.

> Mutex, ConditionVariable#wait, Queue#push

I wish I can say, no user should use such a Thread primitive in the context of non-blocking selector. Of course, some synchronisation primitives might be necessary, like condition, semaphore and queue. These can be easily implemented by using Fiber, but the question remains whether Thread Mutex, ConditionVariable and Queue should be supported and how should it work.

The implementation of Fiber variant is typically based on cooperatively shared resources and Fiber.yield/Fiber#resume. For example, a queue is just an array, there is no need for Mutex. The question is then, how should Mutex.synchronise work within a non-blocking context? Well, no change is required, but it will cause contention if it blocks a long time. I think it's acceptable trade off, but users should adopt concurrent primitives (can be supplied by a gem) built on top of Fiber.

The only time a user would encounter a problem is if they use Queue and have mutiple readers/writers within same non-blocking context. By the current design, it would deadlock in some situations, when a task is waiting for an item, and the producer task cannot run.

The only solution to this kind of problem is to have a more elaborate system like go which allows to transfer "coroutines" between threads. We could implement such a system, but the trade off is now that every task must deal with synchronisation issues and users must be aware of such issues.

I tried to find a way of describing such systems. While go implements N:M threading model, the proposal here is N:1:M threading model. Because we insert the "1", it's both a bottleneck, and a protection for users, to avoid having to deal with explicit parallelism, while getting many benefits of concurrency.

Files

port_scanner_threadlet.rb	925 Bytes	06/13/2018	normalperson (Eric Wong)