Ruby trunk - Feature #13618

[PATCH] auto fiber schedule for rb_wait_for_single_fd and rb_waitpid

06/01/2017 12:14 AM - normalperson (Eric Wong)

Status:	Assigned
Priority:	Normal
Assignee:	normalperson (Eric Wong)
Target version:	

Description

```
auto fiber schedule for rb_wait_for_single_fd and rb_waitpid

Implement automatic Fiber yield and resume when running
rb_wait_for_single_fd and rb_waitpid.

The Ruby API changes for Fiber are named after existing Thread
methods.

main Ruby API:

    Fiber#start -> enable auto-scheduling and run Fiber until it
           automatically yields (due to EAGAIN/EWOULDBLOCK)

The following behave like their Thread counterparts:

    Fiber.start - Fiber.new + Fiber#start (prelude.rb)
    Fiber#join - run internal scheduler until Fiber is terminated
    Fiber#value - ditto
    Fiber#run - like Fiber#start (prelude.rb)

Right now, it takes over rb_wait_for_single_fd() and
rb_waitpid() function if the running Fiber is auto-enabled
(cont.c::rb_fiber_auto_sched_p)

Changes to existing functions are minimal.

New files (all new structs and relations should be documented):

    iom.h - internal API for the rest of RubyVM (incomplete?)
    iom_internal.h - internal header for iom_(select|epoll|kqueue).h
    iom_epoll.h - epoll-specific pieces
    iom_kqueue.h - kqueue-specific pieces
    iom_select.h - select-specific pieces
    iom_pingable_common.h - common code for iom_(epoll|kqueue).h
    iom_common.h - common footer for iom_(select|epoll|kqueue).h

Changes to existing data structures:

    rb_thread_t.afrunq   - list of fibers to auto-resume
    rb_vm_t.iom          - Ruby I/O Manager (rb_iom_t) :)

Besides rb_iom_t, all the new structs are stack-only and relies
extensively on ccan/list for branch-less, O(1) insert/delete.

As usual, understanding the data structures first should help
you understand the code.

Right now, I reuse some static functions in thread.c,
so thread.c includes iom_(select|epoll|kqueue).h

TODO:

    Hijack other blocking functions (IO.select, ...)
```

```
I am using "double" for timeout since it is more convenient for
arithmetic like parts of thread.c.   Most platforms have good FP,
I think.  Also, all "blocking" functions (rb_iom_wait*) will
have timeout support.

./configure gains a new --with-iom=(select|epoll|kqueue) switch

libkqueue:

  libkqueue support is incomplete; corner cases are not handled well:

    1) multiple fibers waiting on the same FD
    2) waiting for both read and write events on the same FD

  Bugfixes to libkqueue may be necessary to support all corner cases.
  Supporting these corner cases for native kqueue was challenging,
  even.  See comments on iom_kqueue.h and iom_epoll.h for
  nuances.

Limitations

Test script I used to download a file from my server:
----8<---
require 'net/http'
require 'uri'
require 'digest/sha1'
require 'fiber'

url = 'http://80x24.org/git-i-forgot-to-pack/objects/pack/pack-97b25a76c03b489d4cbbd85b12d0e1ad287
17e55.idx'

uri = URI(url)
use_ssl = "https" == uri.scheme
fibs = 10.times.map do
  Fiber.start do
    cur = Fiber.current.object_id
    # XXX getaddrinfo() and connect() are blocking
    # XXX resolv/replace + connect_nonblock
    Net::HTTP.start(uri.host, uri.port, use_ssl: use_ssl) do |http|
      req = Net::HTTP::Get.new(uri)
      http.request(req) do |res|
    dig = Digest::SHA1.new
    res.read_body do |buf|
      dig.update(buf)
      #warn "#{cur} #{buf.bytesize}\n"
    end
    warn "#{cur} #{dig.hexdigest}\n"
      end
    end
    warn "done\n"
    :done
  end
end

warn "joining #{Time.now}\n"
fibs[-1].join(4)
warn "joined #{Time.now}\n"
all = fibs.dup

warn "1 joined, wait for the rest\n"
until fibs.empty?
  fibs.each(&:join)
  fibs.keep_if(&:alive?)
  warn fibs.inspect
end

p all.map(&:value)
```

```
Fiber.new do
  puts 'HI'
end.run.join
```

History

#1 - 06/01/2017 12:41 AM - normalperson (Eric Wong)

Pull request below for non-"git am" users...

I tried my best to add many comments throughout the code.

I realize this is a lot of new code; and not a typical or common
usage of kqueue or epoll. The kqueue code ended up being very
complicated to support corner cases (see comments in iom_kqueue);
so perhaps the epoll implementation should be easiest-to-understand.

I suggest understanding data structures, first; everything else
will be easier. Please do not hesitate to ask here if you come
across questions or bugs or any comments.

Finally; libkqueue is broken (using epoll on Linux) for corner
cases. I am using a FreeBSD 11.0 VM for kqueue development; but
I may resume working with libkqueue upstream if I have time.
I expect real-world Linux users to be using native epoll,
of course; so no big problems, there.

The following changes since commit d0015e4ac6b812ea1681b1f5fa86fbab52a58960:

Improve performance of implicit type conversion (2017-05-31 12:30:57 +0000)

are available in the git repository at:

git://80x24.org/ruby iom

for you to fetch changes up to 8d6b09d46fcdf6362d6f875347c4790d5cf86401:

auto fiber schedule for rb_wait_for_single_fd and rb_waitpid (2017-06-01 00:07:18 +0000)

Eric Wong (1):
auto fiber schedule for rb_wait_for_single_fd and rb_waitpid

```
common.mk                                    |   7 +
configure.in                                 |  32 ++
cont.c                                       | 119 +++-
include/ruby/io.h                            |   2 +
iom.h                                        |  92 ++++
iom_common.h                                 | 198 +++++++
iom_epoll.h                                  | 423 ++++++++++++++++
iom_internal.h                               | 251 +++++++++
iom_kqueue.h                                 | 600 ++++++++++++++++++++++
iom_pingable_common.h                        |  46 ++
iom_select.h                                 | 306 ++++++++++
prelude.rb                                   |  12 +
process.c                                    |  14 +-
signal.c                                     |  40 +-
.../wait_for_single_fd/test_wait_for_single_fd.rb  |  44 ++
test/lib/leakchecker.rb                      |   9 +
test/ruby/test_fiber_auto.rb                 | 238 ++++++++
thread.c                                     |  42 ++
thread_pthread.c                             |   5 +
vm.c                                         |   9 +
vm_core.h                                    |   4 +
21 files changed, 2479 insertions(+), 14 deletions(-)
create mode 100644 iom.h
create mode 100644 iom_common.h
create mode 100644 iom_epoll.h
create mode 100644 iom_internal.h
create mode 100644 iom_kqueue.h
create mode 100644 iom_pingable_common.h
create mode 100644 iom_select.h
create mode 100644 test/ruby/test_fiber_auto.rb
```

#2 - 06/01/2017 02:15 AM - ko1 (Koichi Sasada)

Thank you for your great work.

summary of this comment

Recent days I'm thinking about this feature's "safety" or "dependability".
Because of this issue, I think it is difficult to employ this feature right now.

Non-auto-fibers

Without this feature, Fiber switching is explicit (Fiber.yield) and most of case, it is easy to write several operations in atomic.

Typical atomic operation is increment. Let's think about it with example: t = n; ...; n = t+1.

```
def some_method
  Fiber.yield
end

n = 0
f1 = Fiber.new{
  t = n
  some_method
  n = t + 1
}

f1.resume
n += 1
f1.resume

p n #=> 1 (although two increments are tried)
```

In this case, main fiber and fiber f1 try to increment n and some_method breaks atomicity because of Fiber.yield. Of course, nobody write such silly code and it is easy to check because Fiber.yield is strongly coupled with Fiber operations written by users (basically, libraries don't call Fiber.yield).

auto-fibers

However, auto-fiber switching introduce this kind of danger.

```
# assume all fibers are auto-scheduling fibers

n = 0
f1 = Fiber.new{
  n = log(t) + 1
}

f1.resume # auto-fibers should not call resume
          # but please allow me, this is pseudo-code to describe an issue.
n += 1
f1.resume

p n
```

If log() method tries to send a log message over network, Fiber will switch to other fibers.

Problems are:

- It is difficult to know which operations should be run in atomic (users write code without checking atomicity).
- It is difficult to find out which method can switch.
  - Not only user writing code, but also all library code can switch fibers.
  - This means that we need to check all of library code to know that they don't violate atomic assumptions.
- It introduced non-deterministic behavior (with Fiber.yield it will be deterministic behavior and it is easy to reproduce the problem).

This kind of difficulties are same as threading. The impact can be smaller than threading (because threading can switch anywhere and it is very hard to predict the behavior. Auto-fibers switch only at blocking operations especially on IO operations).

Consideration

To solve this behavior, we have several choice.

(1) Introduce synchronization mechanisms for auto-fibers

Like Mutex, Queue and so on.
On Ruby 1.8 era, we have Thread.exclusive to prohibit thread-switching.

I don't want to choice this option because it is what I want to avoid from Ruby.

(2) Introduce limitations

The problem "It is difficult to find out which method can switch" is because we need to check whole of code. If we can restrict the auto-fiber switching, this problem can be smaller.

(2-1) Introduce Fiber switching methods

Instead of implicit blocking (IO) operations, introduce explicit blocking operations can switch. We can check all of source code by grep.

(2-2) Check context

Permit fiber switching only at permitted places, by block, pragma, and so on.

```
# auto-fiber: true # <- this file can switch fibers automatically
Fiber.new(auto: true){
  ...
  io.read # can switch
  ...
  something_defined_in_gem # can't switch
  ...
}
```

I think other languages like Python, JavaScript employs this idea. I need to survey more on such languages.

(3) Something else cleaver

Introducing debugger is one choice (maybe it is easy than threading issues).
But we can't avoid troubles (and maybe the troubles should be not frequent, non-reproducible).

Other option is to introduce hooks to implement auto-fibers and provide auto-fibers by gems and advanced users know the above risk use this feature. But not good idea because we can't provide good way to write for many people.

thought?

#3 - 06/01/2017 05:48 AM - ko1 (Koichi Sasada)

Another idea is change this name from Fiber but thread-related name, but implementation is based on Fiber. It means resurrection of ruby 1.8 green thread without time based preemption (actually, implementation is similar).

Personally I want to avoid threading problems, but I show this idea as an option.

off topic: it is similar to CPU architecture about hardware multi-threading: simultaneous multi-threading (SMT, HT for x86) vs. virtical threading (Sparc, switching on cache miss).

#4 - 06/01/2017 09:21 AM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> Issue #13618 has been updated by ko1 (Koichi Sasada).
>
> Thank you for your great work.

You're welcome :)

> ## summary of this comment
>
> Recent days I'm thinking about this feature's "safety" or "dependability".
> Because of this issue, I think it is difficult to employ this feature right now.

I disagree. I do not recall Ruby 1.8 Threads being a big problem
for Rubyists. Modern Rubyists seem OK using native Threads
("OK", not "great" :)

We can improve APIs (maybe more Queue/SizedQueue, less Mutex).

What auto-Fiber provides is an option to reduce memory usage and
improve scalability without rewriting existing synchronous

codebases (e.g. Rack + middlewares).

In my experience, I think Ruby gained more users during 1.8-era when it memory usage was low for green threads; and lost users as 1.9/2.x memory usage increase (and I guess 3rd-party libs grew, too).

The safety difference between auto-Fiber and Thread is a minor point. Lowering memory usage while retaining compatibility with existing synchronous code is my reason for working on this.

- It is difficult to know which operations should be run in atomic (users write code without checking atomicity).
- It is difficult to find out which method can switch.
  - Not only user writing code, but also all library code can switch fibers.
  - This means that we need to check all of library code to know that they don't violate atomic assumptions.
- It introduced non-deterministic behavior (with Fiber.yield it will be deterministic behavior and it is easy to reproduce the problem).

Yes; we will document all switch points in RDoc and NEWS, of course (maybe write a separate doc/auto-fiber.rdoc)

> This kind of difficulties are same as threading. The impact can be smaller than threading (because threading can switch anywhere and it is very hard to predict the behavior. Auto-fibers switch only at blocking operations especially on IO operations).

Right, I think auto-fiber will have some of the same (probably minor) difficulties as threading. However, I do not believe it is a big problem since Rubyists should already be used to threading.

Consideration

> To solve this behavior, we have several choice.
>
> (1) Introduce synchronization mechanisms for auto-fibers
>
> Like Mutex, Queue and so on.

Yes, I think Queue/SizedQueue should be able to respect Fiber scheduling boundaries. Queue/SizedQueue are especially useful and I plan to implement auto-fiber support for that.

I am not sure about Mutex... (can we defer to Matz for decisions?)

> On Ruby 1.8 era, we have Thread.exclusive to prohibit thread-switching.
>
> I don't want to choice this option because it is what I want to avoid from Ruby.

Right.

Maybe Mutex#synchronize can prohibit auto-switch (or, it will show a warning or raise at auto-switch points).

> (2) Introduce limitations
>
> The problem "It is difficult to find out which method can switch" is because we need to check whole of code. If we can restrict the auto-fiber switching, this problem can be smaller.

Right now for IO, it is double opt-in:

It requires both Fiber#start and IO#nonblock=true.

Sidenote:

As a Rubyist who studies the Linux kernel; I consider it imperative to give Rubyists the choice to make real blocking syscalls (not the "fake blocking" with auto-fiber/green threads).

This is because Linux can optimize "wake-one" situations to:
a) give round-robin load distribution across independent processes
b) avoid thundering herd with multiple threads/processes
c) (I forget...)

(sorry I forgot to note this in my original ticket, but it will
be in the final docs)

> (2-1) Introduce Fiber switching methods
>
> Instead of implicit blocking (IO) operations, introduce explicit blocking operations can switch. We can check all of source code by grep.

I am against this. Instead, I want it to be easy to port
existing Thread-aware codebases over.

Notice my example test script used net/http from stdlib.

I would like to use existing stdlib (net/*, webrick, drb, ...)
as much as possible without modifications. That means many
existing Ruby libraries can work transparently.

> (2-2) Check context
>
> Permit fiber switching only at permitted places, by block, pragma, and so on.
>
> ```
> # auto-fiber: true # <- this file can switch fibers automatically
> Fiber.new(auto: true){
>   ...
>   io.read # can switch
>   ...
>   something_defined_in_gem # can't switch
>   ...
> }
> ```
>
> I think other languages like Python, JavaScript employs this idea. I need to survey more on such languages.

I do not like this, either. I admit I am not familiar with
those languages. I think we should strive to make existing
Thread-aware Ruby code work well, and as transparently as possible...

> (3) Something else cleaver
>
> Introducing debugger is one choice (maybe it is easy than threading issues).
> But we can't avoid troubles (and maybe the troubles should be not frequent, non-reproducible).

Adding Tracepoint to help track auto-switch should be done
(honestly I have never used this feature in ruby :x).

And yes, I think native threading bugs are trickier to track down
than auto-Fiber switching. Just remember, today we have native
threading and things are OK. And I think there were more happy
Rubyists in 1.8 days.

> Other option is to introduce hooks to implement auto-fibers and provide auto-fibers by gems and advanced users know the above risk use this feature. But not good idea because we can't provide good way to write for many people.
>
> thought?

Again, no. I am really in favor of making it easy to port
existing Thread-aware code to auto-Fiber.

Again; from my experience; I do not believe many Ruby
programmers had safety problems with 1.8 green threads.

Today we have Rubyists who are already used to 1.9/2.x native
Thread already.

The safety improvement is a minor point.

#5 - 06/01/2017 02:40 PM - ko1 (Koichi Sasada)

normalperson (Eric Wong) wrote:

> I disagree. I do not recall Ruby 1.8 Threads being a big problem
> for Rubyists. Modern Rubyists seem OK using native Threads
> ("OK", not "great" :)
> ...
> However, I do not believe it
> is a big problem since Rubyists should already be used to
> threading.
> ...
> And yes, I think native threading bugs are trickier to track down
> than auto-Fiber switching. Just remember, today we have native
> threading and things are OK. And I think there were more happy
> Rubyists in 1.8 days.
> ...
> Again; from my experience; I do not believe many Ruby
> programmers had safety problems with 1.8 green threads.
>
> Today we have Rubyists who are already used to 1.9/2.x native
> Thread already.
>
> The safety improvement is a minor point.

My opinion is opposite. I think "For human being using threading is too hard to use correctly" or "Rubyist shouldn't care about threading difficulties". I agree my opinion is extreme and many "advanced" programmers like Eric can write correct thread programs. But most (many? some? a few?) of ruby programmer (including me) can not write correct code I believe.

(In addition: I heard some advanced programmers say "people can write". I doubt because it is something survivor bias)

(recent days I fixed rubygems' threading problem it is difficult to reproduce)

I often use this metaphor: It is like GC strategy. If people can manage object lifetime, it is faster than using GC (at some case. Some case GC is more faster than manual memory management). However we choose GC because we want to concentrate on writing application code.

I agree auto-fibers is safer than threads. In my mind:

```
danger <-> safe (this is my opinion)
```

```
   parallel threads (JRuby, ...) > concurrent threads (MRI) >>
   auto-fibers (full-auto)       > auto-fiber (restricted) >>
   Guild                         > single thread
```

But auto-fiber can introduce accident and it should be not so frequent, and it is difficult to reproduce. This means it is difficult to debug.

Ruby has many pit falls to shoot our own legs (meta-programming features, open class and so on) but they are deterministic (at most of case).

I think this is how to evaluate the risk of such danger.

C/C++/Java/... (and many imperative languages) choose performance (people should write correct code).

Some languages try to avoid this kind of difficulties. Rust choose threading but introduce harness by type system. Clojure choose STM to prevent atomic violation.

I agree threading and auto-fiber is easy to use. Maybe most of case it is no problem (especially on auto-fiber). But it can includes accident in only few cases and it will be difficult to find out.

I hope Ruby is safe language because I don't want to bother of such difficulties. This is my wish. I agree there are another wish like Eric's and I respect it.

Other than this point, I agree of all of your opinions. If I can believe "All Rubyist can write correct thread programs", your points make sense for me.

(other points)

> Yes; we will document all switch points in RDoc and NEWS,
> of course (maybe write a separate doc/auto-fiber.rdoc)

My point is, if method "foo" is switching point, then any method can call "foo" (bar, and baz, the caller of bar, ...) should be noted. Maybe it is impossible to complete because of Ruby's dynamic nature.

> I would like to use existing stdlib (net/*, webrick, drb, ...)
> as much as possible without modifications. That means many
> existing Ruby libraries can work transparently.

I understand your point.

#6 - 06/01/2017 10:02 PM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> My opinion is opposite. I think "For human being using threading is too hard to use correctly" or "Rubyist shouldn't care about threading difficulties". I agree my opinion is extreme and many "advanced" programmers like Eric can write correct thread programs. But most (many? some? a few?) of ruby programmer (including me) can not write correct code I believe.

I do not believe I can write correct code of any type, actually.
Everything I write; even trivial single-threaded scripts has bugs.

On the other hand, my likelyhood of introducing bugs seems
nearly identical across any environment and programming models.
However, having less/simpler code (and less dependencies) seems
to result in fewer bugs, in my experience.

> (In addition: I heard some advanced programmers say "people can write". I doubt because it is something survivor bias)

Yes.

> (recent days I fixed rubygems' threading problem it is difficult to reproduce)
>
> I often use this metaphor: It is like GC jtrategy. If people can manage object lifetime, it is faster than using GC (at some case. Some case GC is more faster than manual memory management). However we choose GC because we want to concentrate on writing application code.

Right. However, it seems choosing "easier" strategies means
less focus on overall design, leading to more problems down the line.

Since around 2010; I believe unicorn caused major, irreparable
damage to Rack ecosystem by promoting single-threaded design and
having a SIGKILL timeout feature. unicorn made Rubyists stop
caring to fix concurrency bugs and do proper timeouts.

Nowadays Rack apps are both too buggy AND use too much memory :<

I know some people disagree with my assessment of unicorn;
but I prefer to hate everything I've done: it's easier to
find improvements that way :)

> I agree auto-fibers is safer than threads. In my mind:
>
> ```
> danger <-> safe (this is my opinion)
>
>   parallel threads (JRuby, ...) > concurrent threads (MRI) >>
>   auto-fibers (full-auto)       > auto-fiber (restricted) >>
>   Guild                         > single thread
> ```

Agree. So maybe we can design API for "auto-fiber (restricted)"?

> But auto-fiber can introduce accident and it should be not so frequent, and it is difficult to reproduce. This means it is difficult to debug.
>
> Ruby has many pit falls to shoot our own legs (meta-programming features, open class and so on) but they are deterministic (at most of case).

Yes. I think these (along too much code + dependencies) cause
more problems than concurrency bugs.

> normalperson (Eric Wong) wrote:
>
> > Yes; we will document all switch points in RDoc and NEWS,
> > of course (maybe write a separate doc/auto-fiber.rdoc)
>
> My point is, if method "foo" is switching point, then any method can call "foo" (bar, and baz, the caller of bar, ...) should be noted. Maybe it is impossible to complete because of Ruby's dynamic nature.

Right. Maybe that is a lot of documentation...

What if the API were the opposite of Thread.exclusive/Mutex#synchronize?
Perhaps:

```
Fiber.new do
Fiber.auto do
# enable auto-fiber inside this block
end
# disable auto-fiber again
end
```

Maybe Fiber.exclusive can disable Fiber.auto temporarily:

```
Fiber.new do
Fiber.auto do
# enable auto-fiber

Fiber.exclusive do
# temporarily disable auto-fiber
end
# enable auto-fiber again
...
end
end
```

Fiber.auto/Fiber.exclusive would be no-ops unless inside
a Fiber.new block...

But maybe that is too much code and nesting levels;
so I still like Fiber.start more.

> I would like to use existing stdlib (net/*, webrick, drb, ...)
> as much as possible without modifications. That means many
> existing Ruby libraries can work transparently.

I understand your point.

Thanks; that is my biggest wish for this feature.

Anyways, I will leave matz, you and others deal with final API
decisions.

#7 - 06/02/2017 06:05 PM - Eregon (Benoit Daloze)

This is interesting work, I am curious to see how it will work out.

This looks similar to what Crystal has [1].

Does Kernel#puts potentially yields to another auto-Fiber?
I think that would be very counter-intuitive, but it would be tempting if $stdout is a pipe or socket.

Will a read from a socket always yield to the next fiber,
or can it proceed immediately if the socket is ready?
If not, then scheduling is non-deterministic,
even when communicating with a deterministic server.

It seems that the Crystal approach has some issues for terminating correctly.
However, if I understand in your model there is an implicit wait for all auto-fibers until termination at the program end?
This makes more sense to me for cooperative threading.

The description from Crystal mentions:
"Crystal uses green threads, called fibers, to achieve concurrency.
Fibers communicate with each other using channels, as in Go or Clojure, without having to turn to shared memory or locks."
The part about shared memory and locks is a lie though, these fibers do share memory and
atomicity is broken at every possible call that could invoke some IO-like operation.

This is also true for auto-fibers, which is a form of shared-memory concurrency,
and every yielding point will effectively need to assume
any other auto-fiber could have run in between and modified some global state
(unless the yielding order is very clear such as in a small program,
but in larger programs it becomes extremely difficult to know the fiber schedule).

[1] https://crystal-lang.org/docs/guides/concurrency.html

#8 - 06/02/2017 11:21 PM - normalperson (Eric Wong)

eregontp@gmail.com wrote:

> This is interesting work, I am curious to see how it will work out.

Thanks for the interest.

> This looks similar to what Crystal has [1].

Right. But actually I would use MRI 1.8 green threads as
a reference point. The key difference between this and 1.8
is this is tickless (or timer-less); so more predictable.

To me, there are only two types threads available to userland:

1) OS kernel knows about them (native thread)
2) OS kernel has no idea about them (fiber/green thread/goroutine)

> Does Kernel#puts potentially yields to another auto-Fiber?
> I think that would be very counter-intuitive, but it would be tempting if $stdout is a pipe or socket.

Yes, potentially. However, it requires setting IO#nonblock=true
on $stdout (or whatever $> points to), which is rare...

Non-blocking stdout is rare since likely causes headaches if using
system() to run other programs or having 3rd-party libs which
write to stdout.

> Will a read from a socket always yield to the next fiber,
> or can it proceed immediately if the socket is ready?

It only yields on EAGAIN/EWOULDBLOCK when rb_wait_for_single_fd
is called. It will never yield if there is always data.

AFAIK, Ruby io.c+ext/socket/* does not use rb_wait_for_single_fd
until it encounters EAGAIN/EWOULDBLOCK. (I would consider it a
performance bug if it did)

> If not, then scheduling is non-deterministic,
> even when communicating with a deterministic server.

(sorry, double negatives are confusing to me to parse and use).

If a socket can always read/write without encountering
EAGAIN/EWOULDBLOCK, the Fiber may run forever. This will starve
other Fibers, so it is up to the programmer to yield explicitly.

We should add Fiber.pass (like Thread.pass) to aid users with
this. This will protect HTTP/1.1 servers from DoS via request
pipelining.

So I guess scheduling is non-deterministic; but actual use
can be deterministic since the programmer should know when
to yield/pass explicitly?

> It seems that the Crystal approach has some issues for terminating correctly.
> However, if I understand in your model there is an implicit wait for all auto-fibers until termination at the program end?
>
> This makes more sense to me for cooperative threading.

No implicit waiting for termination. Fibers can be forgotten
and dropped at program end; just like threads. I think this is
a necessary condition for supporting fork or exec.

Users must use Fiber#join or Fiber#value to ensure termination;
(same as Thread#join / Thread#value)

> The description from Crystal mentions:

> "Crystal uses green threads, called fibers, to achieve concurrency.
> Fibers communicate with each other using channels, as in Go or Clojure, without having to turn to shared memory or locks."
> The part about shared memory and locks is a lie though, these fibers do share memory and
> atomicity is broken at every possible call that could invoke some IO-like operation.
>
> This is also true for auto-fibers, which is a form of shared-memory concurrency,
> and every yielding point will effectively need to assume
> any other auto-fiber could have run in between and modified some global state
> (unless the yielding order is very clear such as in a small program,
> but in larger programs it becomes extremely difficult to know the fiber schedule).
>
> [1] https://crystal-lang.org/docs/guides/concurrency.html

Yes. Programmers must be careful about shared memory; but
ruby-core can promote+improve APIs like Queue/SizedQueue to use
as communications channels. This should reduce the use of (and
dangers associated with) shared memory.

#9 - 06/09/2017 08:15 AM - ioquatix (Samuel Williams)

To a certain extent, things discussed here are already implemented in

https://github.com/socketry/async

and

https://github.com/socketry/async-io

What are the benefits of having this implemented in core Ruby as opposed to a gem which can be versioned independently and works with all Rubies 2.x, including JRuby and (in theory) Rubinius?

Why not focus on making core part of Ruby fast, and providing the appropriate hooks, rather than expanding her scope and complexity, in a way which has a proven track record for frustration (poorly designed stdlib which can't be fixed or improved due to breaking backwards compatibility).

#10 - 06/09/2017 08:41 PM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> To a certain extent, things discussed here are already implemented in
>
> https://github.com/socketry/async
>
> and
>
> https://github.com/socketry/async-io
>
> What are the benefits of having this implemented in core Ruby as opposed to a gem which can be versioned independently and works with all Rubies 2.x, including JRuby and (in theory) Rubinius?

Neverblock basically tried the same thing with EM and never took
off. I don't know much about getting software adopted or
popularized, but maybe being in core has a better chance of
gaining adoption and being sustainable.

Being in core provides greater compatibility with external
libraries which are not aware of existing event loops. So
3rd-party DB adapters (e.g. mysql2) will be able to take advantage
of these changes transparently if they use rb_wait_for_single_fd
(and I will add a hook for rb_thread_fd_select, too).

It will also be easily possible to get existing primitives like
Queue/SizedQueue to work with Fibers out-of-the-box. Maybe even
Mutex+ConditionVariable, if approved.

One current example is being able to hook rb_waitpid: any
existing code using trap(:CHLD) continues to work transparently
even if using auto-Fiber for I/O; but auto-Fiber users can also
rely on "blocking" Process.waitpid if they desire.

Anyways, accepting any of this into core is not my decision to
make. I will only provide implementation and advice/hints.

A small rant about existing event loops:

Most existing event loop implementations (libev, libevent, EM)
seem stuck in single-thread mentality from legacy select/poll
APIs. They handle MT by having one event loop per-thread;
instead of taking advantage of the fact that modern primitives
like kqueue and epoll are both MT-friendly queues which are
populated by threads running inside the kernel.

In a world where memory and CPU are your only constraints,
you can run one (native thread|process) per-core and thus one
event loop per-core. This is perfectly fine for things like
memcached which are only memory+CPU bound.

That falls down once you have other constraints, such as
physical disks to deal with. I maintain software which reads
and writes simultaneously to dozens, if not hundreds of
rotational disks (JBOD) in a single process. With current APIs
on GNU/Linux and FreeBSD, the only way I've found(*) to deal
with this effectively is to use >=1 pthread per disk.

(*) Various AIO implementations are lacking, too. They
pessimize the hot cache case, lack open/unlink/rename/stat
equivalents, and userland implementations tend to not be
mountpoint/device-aware. Native AIO requires O_DIRECT in
Linux, so no page cache at all :<

> Why not focus on making core part of Ruby fast, and providing the appropriate hooks, rather than expanding her scope and complexity, in a way which has a proven track record for frustration (poorly designed stdlib which can't be fixed or improved due to breaking backwards compatibility).

I think core and stdlib can evolve best if done together.

Fiber has been in production Ruby for nearly a decade now, with
only minor improvements, and seems largely ignored in the wider
scheme of things. I guess they're not that useful in practice.

And just because we're adding new features does not mean we're
not also finding places to optimize our code.
Mutex/Queue/SizedQueue/ConditionVariable are already faster in
trunk because of preparation work to make them auto-Fiber aware:

https://bugs.ruby-lang.org/issues/13517
https://bugs.ruby-lang.org/issues/13552

Why can't stdlib be fixed? Just because we need to support old
behaviors and APIs does not mean we cannot improve things.

Having a solid stdlib is a great way to improve core and
vice-versa, and helps us bridge the gap for end user code.

Finally, keep in mind there are Rubyists who are not
enthusiastic users willing to explore, they're the
"distro users". It'll be easier for them to pick up Ruby
and use Ruby apps if stdlib were better.

Despite using Perl more than Ruby, I'm a conservative "distro
user" myself with Perl. So I'm hesitant to use or depend on
stuff which isn't packaged by distros, especially when it comes
to end user convenience (some who do not even know or care about
what a programming language is).

So yes, I still write Perl 5.8-compatible code, and still
support legacy CentOS 5.x and 6.x systems.

#11 - 06/14/2017 02:13 AM - ioquatix (Samuel Williams)

I appreciate your detailed response it was interesting.

Does Ruby File.read and File.stat (and others) release the GVL? Otherwise, the performance benefit of multiple threads in this specific case is irrelevant. While I agree with you when writing high performance servers in C/C++, it might not be directly relevant to Ruby as it currently stands.

#12 - 06/14/2017 02:51 AM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> Does Ruby File.read and File.stat (and others) release the GVL? Otherwise, the performance benefit of multiple threads in this specific case is

> irrelevant. While I agree with you when writing high performance servers in C/C++, it might not be directly relevant to Ruby as it currently stands.

File.read does. File.stat does not, at the moment. I tried it a while back but the GVL is expensive to release for hot cache situations(*).

File.open, IO.copy_stream, IO#write, IO#read, readpartial, sysread, syswrite all release GVL, too.

In particular, IO.copy_stream is great for large, parallel transfers to/from high-latency storage.

(*) the cost of GVL for quick ops is a big reason I want to get rid of it

But yeah, maybe the small regression from releasing GVL is acceptable for now with File.stat. It's better than getting stufk on NFS or slow disks.

File.rename, File.unlink, most Dir methods all have the same problem with slow storage, too. We already pay the price for small regressions when releasing GVL in current cases, so maybe those can be GVL release points.

#13 - 06/15/2017 01:56 AM - ioquatix (Samuel Williams)

Thanks for your detailed reply. It's impressive and useful that you have such a good knowledge of these issues.

I spent some time just thinking about this issue, and how this feature tries to solve the problem in Ruby.

On the one hand, I'm fundamentally opposed to increasing the surface area of Ruby when it could be done by writing a gem. This has a massive upstream cost, affecting both JRuby and Rubinius. While I appreciate what you are saying w.r.t. maximising usage, I feel like building this into Ruby will cause stagnation of progress long term - one solution for all problems isn't always ideal. Seeing initiatives like stdgems.org only reinforces how I feel about this.

Generally speaking - I really appreciate the work that's been done here. I also feel like you've reinvented nio4r, async and a bunch of other stuff, at a very low level, without as much testing, compatibility, etc.

Ideally, we could move all socket related code into a gem - perhaps that's already on the cards e.g. stdgems. Once that's done, fixing issues like exceptions: false would be easier since it can be versioned.

I was thinking about how we could expose this to Ruby - and ideally, I think we should add two functions:

IO.wait_for_single_fd and IO.wait_for_pid. The C functions rb_wait_for_single_fd and rb_waitpid would invoke these functions, and these functions would implement the current logic of the current C functions. It probably makes sense to think in more detail how these functions should work - e.g. wait_for_multiple_fds (or select), or something more elaborate.

Then, we could allow things like async and auto-fibers to extend Ruby's IO system to provide a policy for blocking IO. auto-fibers could be implemented as a gem with a C extension.

What do you think?

#14 - 06/15/2017 08:31 PM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> Thanks for your detailed reply. It's impressive and useful that you have such a good knowledge of these issues.

No problem.

> I spent some time just thinking about this issue, and how this feature tries to solve the problem in Ruby.
>
> On the one hand, I'm fundamentally opposed to increasing the surface area of Ruby when it could be done by writing a gem. This has a massive upstream cost, affecting both JRuby and Rubinius. While I appreciate what you are saying w.r.t. maximising usage, I feel like building this into Ruby will cause stagnation of progress long term - one solution for all problems isn't always ideal. Seeing initiatives like stdgems.org only reinforces how I feel about this.

I understood something it was already decided by matz and ko1 to do something along the lines of auto-Fiber. Though I can't find ko1's original message in the archives, it's mostly quoted in in my reply to him:

http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/80531

I should note some languages like Go, Erlang, Haskell, and the afore-mentioned Crystal all have lightweight threading along these lines in the core language.

In their current state, Fibers are much less useful than the equivalents in those languages; while native Threads are too expensive. Something in between Fibers and Threads seems desirable; maybe we can give auto-Fiber another (short) name; but I'm not sure it's necessary.

I was also influenced to explore lightweight threading in a rack-devel thread and the responses James Tucker wrote to me:

Subject: big responses to slow clients: Rack vs PSGI

It's somewhere in https://groups.google.com/group/rack-devel but that requires JS; so I can't view or link to it using w3m :<

> Generally speaking - I really appreciate the work that's been done here. I also feel like you've reinvented nio4r, async and a bunch of other stuff, at a very low level, without as much testing, compatibility, etc.

That's a fair point about less testing and compatibility. But, I think there is more code using normal Ruby stdlib that can automatically take advantage of these changes so we'll be able to nail down any problems quickly.

On a technical level, I consider the design of libev (used by nio4r and async) too limited in that it does not take advantage of thread-safety baked into kqueue and epoll. Thinking in terms of "events vs. threads" too limiting. As I've said before; combining them is advantageous because both have their uses. kqueue is an thread-friendly queue, so is epoll.

This feels like the microkernel vs monolithic kernel debate, too. On one level, isolation and compartmentalization provided by micro-kernels is appealing; but the ease-of-development of a monolith allowed Linux to become the kernel for nearly everything, from tiny IoT devices to giant supercomputers.

And that doesn't preclude things like loadable modules and FUSE for userspace filesystems from being useful, despite core filesystem drivers being bundled with Linux. So I think async can still be supported as an alternative for Ruby; but the bundled implementation can benefit more from tighter integration into the core.

A more recent example might be git; which included high-level non-essential "porcelain" tools early on in addition to the core "plumbing". Initially, it was intended that separately maintained wrappers such as "cogito", would implement the porcelain UI bits and git would remain low-level plumbing. That ended up making both development and usage more complicated. Eventually git swallowed up most of the cogito functionality and cogito was abandoned.

git also ended up with bundled functionality that would've been separately packaged in other VCSes, including import/export tools for email, CVS, SVN, etc.

The most relevant example from git might be the bundling of libxdiff in git, allowing optimizations and tweaks not possible with an external diff. However, GIT_EXTERNAL_DIFF still remains supported for less-common use cases.

On a non-technical level:

Finally, this (ruby-core) is one of the few places I can still contribute to in the Ruby world. All other relevant Ruby projects requires running non-Free software (including JS) and having to abide accept Terms-of-Service set by a corporation.

Fwiw, I agree with Rubinius philosophy of implementing more of Ruby in Ruby and would rather contribute to that; but the above is a huge factor in why I went on to work on C Ruby, instead. (the other major factor is I strongly prefer C to C++).

> Ideally, we could move all socket related code into a gem - perhaps that's already on the cards e.g. stdgems. Once that's done, fixing issues like exceptions: false would be easier since it can be versioned.

Maybe that'll be done, too, but not my call.
But what about IO.pipe, backtick, and IO.popen?

> I was thinking about how we could expose this to Ruby - and ideally, I think we should add two functions:
>
> IO.wait_for_single_fd and IO.wait_for_pid. The C functions rb_wait_for_single_fd and rb_waitpid would invoke these functions, and these functions would implement the current logic of the current C functions. It probably makes sense to think in more detail how these functions should work - e.g. wait_for_multiple_fds (or select), or something more elaborate.

Maybe.. I guess we already have IO#wait_*able in io/wait; and Process.wait*/IO.select is already possible to override and that would have the same effect. We'd also have to expose the optional read/write buffering + encoding conversion and make that accessible to pure Ruby.

It would make C Ruby feel closer to Rubinius and that would be nice :) I'm not sure how feasable it would be; to introduce more Ruby-visible APIs to implement this.

And I think exposing more APIs to handle FDs directly is a mistake in the presence of native threads. My proposed C API prefers "int *fd" and "rb_io_t" to deal with close notification handling. Multithreaded programs recycle FDs frequently and internal APIs need to be prepared to deal with that.

The implementation I proposed also takes advantage of some C-only optimizations such as reading/writing to memory across Fiber stack boundaries: something which cannot be done with higher-level APIs . Similar optimizations already landed for thread_sync.c (Mutex/Queue) as well as IO#close in trunk.

Again, designing user-visible APIs is most difficult and ruby-core have to think most about long-term support and consequences.

So the difficulty of changing/adding APIs is:

```
1) internal C API (easiest)
2) public C API (difficult)
3) Ruby API (most difficult)
```

So, I've mainly done 1) and made minimal additions to 3).
Only changes to 2) are to internal behavior, so use from C extensions remains the same.

> Then, we could allow things like async and auto-fibers to extend Ruby's IO system to provide a policy for blocking IO. auto-fibers could be implemented as a gem with a C extension.
>
> What do you think?

I guess this is meant for matz and ko1.

We could actually have that today; and I guess you already have
that with async. All the IO methods are well-documented and
you can even ignore/override the existing IO buffering if you
override all the methods by monkey patching core classes.
Heck, you may even go as far as to never allocate rb_io_t
if you override IO.open/IO.pipe/*Socket.new/... and replace
them with your own class.

What I think is (or at least ought to be) irrelevant.

I only give matz and ko1 another option to choose from. We can wait
for matz and ko1 to decide what to do, maybe they'll discuss
this at: https://bugs.ruby-lang.org/projects/ruby/wiki/DevelopersMeeting20170616Japan
I certainly won't attend meetings or try to influence anybody
using anything besides plain-text messages, here.

#15 - 06/19/2017 01:17 PM - ioquatix (Samuel Williams)

Ruby Fibers as they currently stand are perfect and making them more complex is a mistake IMHO.

Let's be clear on this: auto-fibers are really just Fibers that yield when you call a blocking operation. It's as if you are rewriting the blocking function to call Fiber.yield.. and as you have implemented by overriding rb_wait_for_single_fd which invokes something to resume the fiber when the blocking function is done. This is exactly what async does, but it does it the only way currently possible - by wrapping around _nonblock methods. It's the reverse of what your proposed method does - by handling rb_wait_for_single_fd. Because I can't access that method from async without writing C, my choice is limited. But, if it was available, async could use it successfully.

I appreciate what you said about multi-thread multi-fiber execution using your proposed reactor design. I think it's good and it's probably better than libev. It's excellent that you have thought about how to solve these problems and I admire it. However, in my experience, libev is fast enough and n-m concurrency model is fast enough for Ruby. Until Ruby is several orders of magnitude faster, it won't make much difference, except perhaps a tiny bit of latency, but there are benefits to keeping a single request on a single thread or process - you can avoid having to deal with locking and other synchronisation primitives in some cases, e.g. caches. So, there are tangible benefits to using, say, m-process n-fibers vs n-fibers/m-threads model. Ruby has never really suited multi-threaded model unfortunately.

Just to be clear: I'm more interested in semantics than implementation. Get the semantics right and the correct implementation will follow. I see a lot of work done here on an implementation (which is awesome and it looks good), but I'm not completely clear that the semantics are really sound.

In contrast, Async is all about getting the right semantics and finding the implementation that suits.

#16 - 06/20/2017 07:11 PM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> I appreciate what you said about multi-thread multi-fiber
> execution using your proposed reactor design. I think it's
> good and it's probably better than libev. It's excellent that
> you have thought about how to solve these problems and I
> admire it. However, in my experience, libev is fast enough and
> n-m concurrency model is fast enough for Ruby. Until Ruby is
> several orders of magnitude faster, it won't make much
> difference, except perhaps a tiny bit of latency, but there
> are benefits to keeping a single request on a single thread or
> process - you can avoid having to deal with locking and other
> synchronisation primitives in some cases, e.g. caches. So,
> there are tangible benefits to using, say, m-process n-fibers
> vs n-fibers/m-threads model. Ruby has never really suited
> multi-threaded model unfortunately.

Just one correction; auto-Fiber does not migrate fibers or
migrate userspace(*) I/O operations across native threads at the
moment. You might be confusing this with my other
non-Fiber-using server designs which do migrate I/O operations
across threads.

For auto-fiber, there's minimal locking requirements even if we
get rid of GVL. It relies on locking already done by the
kernel; kqueue will require extra locking in the corner case
where read and write filters are both installed for an FD.

(*) Of course, Linux kernel soft IRQ handlers can migrate work
across cores in the background.

> Just to be clear: I'm more interested in semantics than

> implementation. Get the semantics right and the correct implementation will follow. I see a lot of work done here on an implementation (which is awesome and it looks good), but I'm not completely clear that the semantics are really sound.

Anyways, it looks like matz is inclined to accept it; but ko1 wants some semantic tweaks with the API (but I'm not sure what/how, exactly).

https://docs.google.com/document/d/1z19pKt8jlpiEUR3RnWWBCfs3OR_hbiAZMwpQ6ZTIlP0/pub

(I've only viewed it with w3m, no idea if I'm missing anything due to lack of JS)

#17 - 06/29/2017 06:11 AM - normalperson (Eric Wong)

Updated patch against r59201:
https://80x24.org/spew/20170629043509.14939-1-e@80x24.org/raw

matz/ko1: any idea on what changes to the Ruby API you guys want?

Anyways, I will make IO.select / rb_thread_fd_select sometime soonish...

#18 - 07/13/2017 08:37 AM - ko1 (Koichi Sasada)

sorry for long absent about this topic. it is hard task (hard to start writing up because of problem difficulties and my English skil ;p) to summarize about this topic.

I try to write step by step.

Discussion at last developers meeting

Thread/Fiber switch safety

Koichi: (repeat my opinion about difficulty of thread/fiber safety)

akr: providing better synchronize mechanism (such as go-lang has) and encouraging safe parallel computation seems better.

Koichi: It is one possible solution but my position is "if people can shoot their foot, people will shoot".

Matz: I don't like to force people to use lock and so on.

(the point is Matz doesn't reject "-safe" approach)

Introduce restriction

(The following idea is not available at last meeting (only part of idea I showed))

Koichi:
The problem of this feature is mind gap using auto-fiber user and script writer. This is same as thread-safety. Person A consider the code is auto-fiber safe, and other person B (can be same as A) write a code without auto-fiber safety, then it will be problem.

In general, most of existing libraries are not auto-fiber safe code (it doesn't mean most of libraries are not auto-fiber safe. Many code are auto-fiber safe without any care).

If we can know a code (and code called by this code) is auto-fiber safe, we can use auto-fiber in safe.

There are three type of code.

- (1) don't care about auto-fiber
- (2) auto-fiber aware code (assume switching is not allowed at the beginning)
- (3) auto-fiber aware code (don't care it is allowed or not allowed to switch)

There are three types of status.

- (a) can't switch
- (b) can enable to switch, but don't switch
- (c) can switch

in matrix

```
    can switch / can enable switch
(a) can't      / can't
(b) can't      / can
(c) can        / ??
```

matrix with (1-3) and (a-c)

```
     (a)      (b)      (c)
(1)   OK       NG       NG
(2)   OK       OK       NG
(3)   OK(*1)   OK(*1)   OK
```

(1)-(b) and (1)-(c) is not accepted because other method called from this code can switch the context.
(2)-(c) is also unacceptable because the beginning of code is not auto-fiber aware.

*1) Possible problem: (3) can introduce dead-lock problem because it can stop forever.

Normal threads start from (a).
Auto-fibers start from (b). They are written in (1), (2) and (3). Maybe (2) is written for auto fiber top-lelvel. This code will call some async methods which can change context.

My proposal is, to write down explicitly of (1) to (3) and (a) to (c) in program.

At the meeting, I proposed non-matured keywords(-like) to control them.
(and just now I don't have good syntax for it yet)

akira: If we introduce such keywords, we need to rewrite all of code if we want to use auto-fiber web application request handler (for example, we need to rewrite Rails to run on auto-fiber based rack server).

Matz: it is unacceptable to introduce huge rewriting for existing code.

(IMO (not appeared in last meeting) we need to rewrite all of code even if we don't introduce keywords to make sure the auto-fiber safety)

after this discussion

Matz and I discussed about this issue, and we conclude that it is too early to introduce this feature on Ruby 2.5.

I want to consider this issue further. auto-fiber based guild is one possibility, this mean we can introduce object isolation and context switching each other.

#19 - 07/13/2017 10:32 PM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> sorry for long absent about this topic. it is hard task (hard to start writing up because of problem difficulties and my English skil ;p) to summarize about this topic.

No problem, thank you for summarizing.

> I try to write step by step.

Discussion at last developers meeting

Thread/Fiber switch safety

> Koichi: (repeat my opinion about difficulty of thread/fiber safety)
>
> akr: providing better synchronize mechanism (such as go-lang has) and encouraging safe parallel computation seems better.
>
> Koichi: It is one possible solution but my position is "if people can shoot their foot, people will shoot".

I think your approach is too cautious.

We already have many dangerous things in Ruby, even in single-threaded code. For example: File.read, IO#read, IO#gets are all dangerous with no size limit: they can cause

out-of-memory or swapping on gigantic inputs, leading to DoS.

Fork and inadvertant sharing of open files/sockets can also cause problems. And there are also pathological Regexp which can cause unbound CPU usage.

> Matz: I don't like to force people to use lock and so on.
>
> (the point is Matz doesn't reject "-safe" approach)

Introduce restriction

> (The following idea is not available at last meeting (only part of idea I showed))
>
> Koichi:
>
> The problem of this feature is mind gap using auto-fiber user and script writer. This is same as thread-safety. Person A consider the code is auto-fiber safe, and other person B (can be same as A) write a code without auto-fiber safety, then it will be problem.
>
> In general, most of existing libraries are not auto-fiber safe code (it doesn't mean most of libraries are not auto-fiber safe. Many code are auto-fiber safe without any care).

Right; most code does not have to care; and all these dangers already exist with native Threads.

> If we can know a code (and code called by this code) is auto-fiber safe, we can use auto-fiber in safe.
>
> There are three type of code.
>
> - (1) don't care about auto-fiber
> - (2) auto-fiber aware code (assume switching is not allowed at the beginning)
> - (3) auto-fiber aware code (don't care it is allowed or not allowed to switch)
>
> There are three types of status.
>
> - (a) can't switch
> - (b) can enable to switch, but don't switch
> - (c) can switch
>
> in matrix
>
> ```
>    can switch / can enable switch
> (a) can't      / can't
> (b) can't      / can
> (c) can        / ??
> ```
>
> matrix with (1-3) and (a-c)
>
> ```
>      (a)      (b)      (c)
> (1)   OK       NG       NG
> (2)   OK       OK       NG
> (3)   OK(*1)   OK(*1)   OK
> ```
>
> (1)-(b) and (1)-(c) is not accepted because other method called from this code can switch the context.
> (2)-(c) is also unacceptable because the beginning of code is not auto-fiber aware.
>
> *1) Possible problem: (3) can introduce dead-lock problem because it can stop forever.

Perhaps holding Mutex lock should disable auto-fiber switching. This should prevent deadlocks, I think.

Existing code has Mutexes, so I'm not sure how they should interact with auto-Fiber. I agree with Matz that we should discourage locking, so I guess disabling auto-Fiber switch while Mutex is held is the most straightforward solution.

> Normal threads start from (a). Auto-fibers start from (b).

> They are written in (1), (2) and (3). Maybe (2) is written for
> auto fiber top-lelvel. This code will call some async methods
> which can change context.
>
> My proposal is, to write down explicitly of (1) to (3) and (a)
> to (c) in program.
>
> At the meeting, I proposed non-matured keywords(-like) to control them.
> (and just now I don't have good syntax for it yet)
>
> akira: If we introduce such keywords, we need to rewrite all
> of code if we want to use auto-fiber web application request
> handler (for example, we need to rewrite Rails to run on
> auto-fiber based rack server).
>
> Matz: it is unacceptable to introduce huge rewriting for existing code.

I agree completely with akira's observation and Matz's opinion
of this.

> (IMO (not appeared in last meeting) we need to rewrite all of
> code even if we don't introduce keywords to make sure the
> auto-fiber safety)

I don't agree with this. A lot of code is already auto-fiber
safe because they are written with GVL+Threads in mind.
(see my original Net::HTTP example); and we also have a lot
of code (webrick, net/*) which worked fine with green Threads
in 1.8

Worst case is we release GVL in a native Thread and forget to
yield to other Fibers in the same Thread. However, that is
already a problem with existing code when run inside Fibers
(e.g. getaddrinfo, IO operations on NFS/slow-disk, ...)

I am working on making rb_thread_fd_select auto-fiber aware,
too. (done for iom_select/iom_epoll, working on iom_kqueue)

> Matz and I discussed about this issue, and we conclude that it
> is too early to introduce this feature on Ruby 2.5.

OK, I will continue to work on implementation improvements
and keep patches rebased to trunk.

> I want to consider this issue further. auto-fiber based guild
> is one possibility, this mean we can introduce object
> isolation and context switching each other.

Do you think this is in the 2.5 timeline?

Thank you.

#20 - 07/31/2017 09:19 AM - ioquatix (Samuel Williams)

I am following this thread and I find it really fascinating.

Thanks everyone for thinking about these issues and Eric for your insightful work and ideas. Just as an aside, I feel like something is being lost in translation w.r.t. the response from Matz and other core Ruby developers. Perhaps we need to have a hangout to discuss these ideas.

I've just released async, async-io and async-dns 1.0.0, along with rubydns 2.0.0 - in addition to this there is also async-http (client and server library) and falcon, a rack compatible server, built on top of async. The http library lacks support for SSL so it's not 1.x yet - still working on that part.

It works on Ruby 2.0+, and most of it also works on JRuby, excepting JRuby's missing support for UDP sockets (https://github.com/jruby/jruby/pull/4684).

I would like to think async is a proof of concept of what is possible with Ruby, in terms of performance. I think it's a solid platform for making network clients and servers, and I've implemented both DNS client/server and HTTP client/server which provide useful test cases for both performance and design.

In terms of design, it's a very simple concept to use with an API that works as if it's sequential, but yields if the operation would block. The user almost cannot make any mistakes, and implementing complex network logic becomes trivial.

In terms of performance, there are few comparisons I can make. If you like more details, let me know. I'm going to be matter of fact, you can draw your own conclusions.

- RubyDNS is about as fast as Bind for a trivial benchmark resolving a fixed set of IP addresses.
- Falcon is as fast as Puma but scales significantly better especially if non-blocking IO is leveraged.
- Falcon and Puma both process requests significantly faster than typical Rack middleware can cope with them. An example would be, Falcon can easily handle 30,000 conn/s on my 8-core workstation, but as soon as I put any non-trivial rack application behind it, it would drop to < 3000 conn/s. Falcon can handle up to 100,000 req/s on the same hardware (e.g. using keep alive).
- I implemented a complete stack in C++ of the same concept, and it achieved roughly on 1 core what Ruby required 8 cores. That is, a single process/thread could handle 25,000 conn/s on 1 core, and about 90,000 req/s. So, Ruby is about 10x slower than similar C++ code.

Eric, my opinion at this point is that the work you've done here is awesome.

What I would personally like to see, is a backend, perhaps an alternative to nio4r, which, as an example, async could use to implement it's reactor. I think that when your selector is running for the current fiber, operations like wait_for_pid and wait_one_fd should be hijacked and go via reactor. I think it should be possible for nio4r to tap into this too some how. This would make things completely transparent for user.

I still believe this should be a gem - even if it's an official one distributed with Ruby, and that Ruby should expose the relevant hooks. Otherwise, it's going to make a lot of trouble for other implementations e.g. JRuby, MRI, etc. Ideally they can just expose the same low-level hooks at the VM level.

I would like to say at this point, with the release of async & (-*) 1.0, I believe that this concept has proven itself - e.g. that the implementation works, that it has good performance, and that it can be used to implement good composable libraries. Whatever form the final library takes, I hope that it is (a) modular (b) fast and (c) composable.

One final opinion that I've formed while working on this project, is that Ruby IO primitives are overly complex and fail to expose the right abstraction. *_nonblock methods never should have existed. If there is one thing I'd wish for, it's that once a decent asynchronous library is adopted, that these methods are not made part of it's public API. async does forward these methods, but it's only to make wrapping existing Net::HTTP work better, and essentially the x_nonblock variant is identical to the x method in async.

#21 - 07/31/2017 09:21 AM - ioquatix (Samuel Williams)

Just to add, Puma has a HTTP parser (and perhaps other bits) written in C, while Falcon is pure Ruby, yet Falcon has better/similar performance in my (hopefully unbiased) tests. Additionally, Falcon had significantly lower latency, and the C++ implementation even moreso.

#22 - 08/30/2017 03:16 AM - mame (Yusuke Endoh)

I comment in compliance with hsbt's request.

Basically I agree with ko1; Thread is considered harmful. Casual Rubyists (including I) had better not use it.

However, I'm not against introducing the feature in question as a professional feature for mature Rubyists.

One issue that I'm concerned about is, the name. (Sorry, but this is an important point to me!) Fiber is fiber because the programmer manages its control flow completely. "Auto-fiber" looks self-contradictory to me. For example, MSDN says:

> A *fiber* is a unit of execution that must be manually scheduled by the application.
> https://msdn.microsoft.com/ja-jp/library/windows/desktop/ms682661(v=vs.85).aspx

I believe that this feature should be introduced with another name. I have no counterproposal, though. Sorry.

#23 - 08/31/2017 06:04 AM - normalperson (Eric Wong)

mame@ruby-lang.org wrote:

> I believe that this feature should be introduced with another
> name. I have no counterproposal, though. Sorry.

What about Thriber? Or Fred?

"Fread" might be confused with fread(3) function, and I don't
know anybody named "Fred", so it is a safe name to choose :)

#24 - 09/12/2017 05:41 AM - normalperson (Eric Wong)

Eric Wrong normalperson@yhbt.net wrote:

> mame@ruby-lang.org wrote:
>
> > I believe that this feature should be introduced with another
> > name. I have no counterproposal, though. Sorry.
>
> What about Thriber? Or Fred?

> "Fread" might be confused with fread(3) function, and I don't
> know anybody named "Fred", so it is a safe name to choose :)

OK, "class Fred" occurs in object.c documentation already,
so maybe it is confusing. So I choose Thriber as a name:

https://80x24.org/spew/20170912053032.13622-1-e@80x24.org/raw

That patch contains the latest version of this feature rebased
against ko1's recent execution context changes in trunk (up to
r59844) along with some bugfixes (infinite wait fix).

It also adds rb_thread_fd_select as a scheduling point
(in addition to rb_wait_for_single_fd and rb_waitpid from
previously published patches). Only lightly tested,
more tests will need to be written...

Naming is hard :<

Pull request available below for git users:

The following changes since commit 65b11a04f10a2438f0d6ba263a78d16367c3aac0:

console.c: set winsize on Windows (2017-09-11 20:10:34 +0000)

are available in the git repository at:

git://80x24.org/ruby thriber

for you to fetch changes up to d9c0095537c3c01d2187e783910cdc92d6c545fc:

thriber: green threads implemented using fibers (2017-09-12 05:29:31 +0000)

Eric Wrong (1):
thriber: green threads implemented using fibers

```
common.mk                                  |   7 +
configure.in                               |  32 +
cont.c                                     | 123 ++-
include/ruby/io.h                          |   2 +
iom.h                                      |  95 +++
iom_common.h                               | 204 +++++
iom_epoll.h                                | 697 ++++++++++++++++
iom_internal.h                             | 280 +++++++
iom_kqueue.h                               | 899 ++++++++++++++++++++++
iom_pingable_common.h                      |  54 ++
iom_select.h                               | 448 ++++++++++
prelude.rb                                 |  12 +
process.c                                  |  15 +-
signal.c                                   |  39 +-
.../wait_for_single_fd/test_wait_for_single_fd.rb  |  62 ++
test/lib/leakchecker.rb                    |   9 +
test/ruby/test_thriber.rb                  | 274 +++++++
thread.c                                   |  76 +-
thread_pthread.c                           |   5 +
vm.c                                       |   9 +
vm_core.h                                  |   4 +
21 files changed, 3324 insertions(+), 22 deletions(-)
create mode 100644 iom.h
create mode 100644 iom_common.h
create mode 100644 iom_epoll.h
create mode 100644 iom_internal.h
create mode 100644 iom_kqueue.h
create mode 100644 iom_pingable_common.h
create mode 100644 iom_select.h
create mode 100644 test/ruby/test_thriber.rb
```

--
Mr. Wrong

#25 - 09/28/2017 01:08 AM - normalperson (Eric Wong)

I've updated the series to support FIBER_USE_NATIVE=0 (along
with the proposed fix for [Bug #13887]).

The primary change for FIBER_USE_NATIVE=0 platforms is to move
away from cross stack linked-list manipulation and use the
heap for allocations, instead. This involved some structure
modifications to make rb_thread_fd_select work on select(2)-based
implementations. Of course, this increases the dependency on
rb_ensure to release heap memory.

FIBER_USE_NATIVE=1 platforms are still more important and faster,
of course.

I've tested on Debian 8.x and FreeBSD 11.0. Test reports from
other platforms appricated, thank you

Patch mbox (gzipped):

https://80x24.org/spew/20170928004228.4538-1-e@80x24.org/t.mbox.gz

...or "git request-pull"-generated pull request:

The following changes since commit d21aab2d3e007372973f2b803d7d8d7f9547f0cc:

- 2017-09-28 (2017-09-27 21:55:33 +0000)

are available in the git repository at:

git://80x24.org/ruby thriber-copy

for you to fetch changes up to 20ea4d710d3d75d946f74346e6a6f3616dac682d:

thriber: non-native fiber support (2017-09-28 00:41:34 +0000)

Eric Wrong (3):
thriber: green threads implemented using fibers
thread_pthread: do not corrupt stack
thriber: non-native fiber support

```
common.mk                                  |   9 +
configure.in                               |  32 +
cont.c                                     | 173 ++--
fiber.h                                    |  54 ++
include/ruby/io.h                          |   2 +
iom.h                                      |  95 +++
iom_common.h                               | 228 ++++++
iom_epoll.h                                | 710 +++++++++++++++++
iom_internal.h                             | 372 +++++++++
iom_kqueue.h                               | 907 ++++++++++++++++++++++
iom_pingable_common.h                      |  49 ++
iom_select.h                               | 473 ++++++++++
prelude.rb                                 |  12 +
process.c                                  |  15 +-
signal.c                                   |  39 +-
.../wait_for_single_fd/test_wait_for_single_fd.rb  |  62 ++
test/lib/leakchecker.rb                    |   9 +
test/ruby/test_thriber.rb                  | 274 +++++++
thread.c                                   |  76 +-
thread_pthread.c                           |  10 +-
vm.c                                       |   9 +
vm_core.h                                  |   4 +
22 files changed, 3541 insertions(+), 73 deletions(-)
create mode 100644 fiber.h
create mode 100644 iom.h
create mode 100644 iom_common.h
create mode 100644 iom_epoll.h
create mode 100644 iom_internal.h
create mode 100644 iom_kqueue.h
create mode 100644 iom_pingable_common.h
create mode 100644 iom_select.h
create mode 100644 test/ruby/test_thriber.rb
```

#26 - 12/07/2017 04:32 AM - normalperson (Eric Wong)

Too late for 2.5, but I'll maintain and periodically rebase this
in hope it can be accepted for 2.6. I've updated patches for
Thriber support against latest trunk (r61067)

https://80x24.org/spew/20171207041831.29005-2-e@80x24.org/raw
https://80x24.org/spew/20171207041831.29005-3-e@80x24.org/raw

Also available at the "thriber-r61067" branch on git://80x24.org/ruby

#27 - 12/10/2017 10:30 PM - ioquatix (Samuel Williams)

I think that the work being done here is great. However I feel that this PR requires far more scrutiny than it's receiving.

It's worth considering that nio4r and friends took several years to stabilise and there is a huge amount of hard earned knowledge embedded in those gems, e.g.

> I am using "double" for timeout since it is more convenient for arithmetic like parts of thread.c. Most platforms have good FP, I think.

e.g. https://github.com/socketry/nio4r/issues/140

I think it's a great idea to have non-blocking evented IO. However, it's not as simple as making read/write non-blocking. How about DNS lookups? Filesystem access? The benefit of a library based approach as I proposed is that these limitations can be clearly part of the contract of a specific library, and people can make different libraries to suit their needs, but making it part of core Ruby is a slippery slope. If anything, it would be better to depend on an established solution for this, so that cases like using the system DNS resolver are handled correctly (e.g. libuv). Otherwise, this is a HUGE addition to the surface area of the ruby interpreter.

#28 - 12/11/2017 01:41 AM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> I think that the work being done here is great. However I feel that this PR requires far more scrutiny than it's receiving.

Of course, which is why you don't see me pushing for it's
inclusion in 2.5. I only present and update it so people can
test it if they're bored. And I only started working on it
because ko1 seemed interested in it at the time.

I'd be surprised if this gets into 2.6 or any release in the
future. Nobody besides you and I seems interested in discussing
this anymore; so likely it'll sit here quietly for a few more
years.

Again, I don't make API decisions, I only present options.

> > I am using "double" for timeout since it is more convenient for arithmetic like parts of thread.c. Most platforms have good FP, I think.
>
> e.g. https://github.com/socketry/nio4r/issues/140

Right. We already have plenty of threading internals using FP
for timing, as well as the public Ruby APIs for IO.select and
IO#wait_*able. Internally, at least it's a minor thing to
change all the internal APIs to use "struct timespec" all around
for maximum precision.

> I think it's a great idea to have non-blocking evented IO. However, it's not as simple as making read/write non-blocking. How about DNS lookups? Filesystem access? The benefit of a library based approach as I proposed is that these limitations can be clearly part of the contract of a specific library, and people can make different libraries to suit their needs, but making it part of core Ruby is a slippery slope. If anything, it would be better to depend on an established solution for this, so that cases like using the system DNS resolver are handled correctly (e.g. libuv). Otherwise, this is a HUGE addition to the surface area of the ruby interpreter.

We have resolv.rb in stdlib; which was at least popular in 1.8
days. It's implemented entirely in Ruby, so it automatically
takes advantage of these Thriber changes, and has seen a fair
amount of use back in the 1.8 days (not that DNS has changed
drastically).

So really, the network I/O part is not a big, or even complex
change, it's 1.8 Threads being made an option again for

Ruby 2.x. I miss 1.8 Threads, but I also like native threads in 1.9/2.x; they each have their place. And the lightweight threading for network I/O is what people seem to care about in other languages (Go, Erlang). nio4r/libuv and async can still be an option and I have no intention of breaking compatibility.

Filesystem access: out-of-scope for this...

I definitely do NOT want to try and make this use callbacks and threadpools behind users' backs, even internally. It pessimizes the common hot cache case (which doesn't require waiting); and more importantly, and I do not want Ruby or any library to interfere with mountpoint-aware code.

Mountpoint awareness is 100% necessary for me so there's no queue blocking when one native thread is doing IO on a fast FS while another native thread is doing IO on a slow FS. I end up with dozens or hundreds of threads, because I have dozens or hundreds of mountpoints of different speeds. This is an uncommon use case, I know, but some people need it and the VM must not get in the way.

So I think anything to deal with FS access specifically is out-of-scope for this issue. We already have native Thread support, and I use it to implement mountpoint-awareness. Some of the GVL-release changes to File and Dir for 2.5 will help with this (which reminds me, I still need to document some of that in NEWS :x).

#29 - 01/23/2018 11:35 AM - hsbt (Hiroshi SHIBATA)

- Assignee set to normalperson (Eric Wong)

- Status changed from Open to Assigned

Hi, Eric.

We discussed your proposal at last developer meeting (Dec 26, 2017)

- Name this "Thread", or something Thread-ish word than Fiber-ish
- Matz doesn't have a strong opinion on the name but prefers 2 words (auto-fiber) than a coined word "Thriber."

Next actions:

- Give a thread-ish name
- Lock and queue should work with auto-fiber?
- Is explicit context switching onto auto-fiber possible?

#30 - 01/23/2018 05:32 PM - normalperson (Eric Wong)

hsbt@ruby-lang.org wrote:

> We discussed your proposal at last developer meeting (Dec 26, 2017)

Awesome news.

> - Name this "Thread", or something Thread-ish word than Fiber-ish

So if we just use "Thread", then existing Thread becomes M:N?
I will think about that... I have many use cases for native threads, too; but maybe they can be satisfied transparently.

> - Matz doesn't have a strong opinion on the name but prefers 2 words (auto-fiber) than a coined word "Thriber."
>
> Next actions:
>
> - Give a thread-ish name

OK, naming is hard :<

LightThread? Maybe too long...

Threadlet?

Not Thread-ish, but "Task"(*) or "Tasklet" may be a candidate.

This might take a while....

- Lock and queue should work with auto-fiber?

I can definitely make Queues work. I think ko1 was mildly against increasing use of Mutex.

One safety feature I was thinking about was disabling auto-switching of Fibers while a Mutex is locked, even.

- Is explicit context switching onto auto-fiber possible?

Yes, right now it's a subclass of Fiber so inherits transfer/resume/yield

(*) Linux kernel uses "task" as generic term for threads, processes, and everything in-between (different flags describe level of sharing for clone(2))

#31 - 01/25/2018 01:15 AM - dsferreira (Daniel Ferreira)

Hi Eric,

I've been reading this issue and I'm finding it fascinating.
Let me play here the role of the ruby developer that is seeking to understand better the asynchronous ruby capabilities.
Every time I read threads(conversations) like this one about the pros and cons of Fibers vs Threads I tend to think: stay away from it.

When people like Kochi write comments like this:

> "But most (many? some? a few?) of ruby programmer (including me) can not write correct code I believe."

or Yusuke Endoh:

> "Thread is considered harmful. Casual Rubyists (including I) had better not use it."

what these comments make us mere mortals feel?

I will speak about me. When I read such a line I tend to step away.
So yes, this situation makes me develop single threaded code as much as possible.
I rely on libraries to handle asynchronous behaviour for me and specially I rely extensively on the actor model.

I doubt I will change my mind unless I start to read that Thread is good to be used or Fiber is good to be used.

When I read all this conversation and you mention corner cases that still have problems that is a NO GO for me.

IMHO to add yet another Thread like feature it should be "The Killer Feature".

The one that we can say to the all community: Hey people use this thing because async is a paradise in ruby land at last.
If we don't have this it will be just another Thread, Fiber nightmare for the very few who accept the overhead of dealing with all the "buts".

And for the record, I use async libraries but I don't feel confident about them either knowing that ruby core is not reliable in itself.
Production code in the enterprise world it is not something to mess around.
For me ruby core needs desperately to change this situation so I really hope your work will be the answer for all of this I'm talking about.
So yes, if it is it fits in ruby core like a glove IMO. If it is not then we will be much worst because instead of 2 walking deads we will

have 3.
A 50% increase is a lot in this domain. Turns things into a joke.

So, can you please explain us what peace of mind will we gain with this new "light thread" in our everyday work?

Thank you very much and keep up the excellent work.
I appreciate specially the care you have in passing across your knowledge on the subject.
Really helpful and insightful.

Note:

Your last two messages are not part of the issue in redmine. I hope my message will be there!
It seems mine did came in as well. I'm copy pasting it.

#32 - 01/26/2018 10:16 AM - ioquatix (Samuel Williams)

In async, I called it Async::Task. I think task is a good name for this kind of thing. In your case, you might want to consider Thread::Task. Since, the lexicographic nesting is similar to the logical nesting.

Regarding kqueue bugs. macOS kqueue implementation is horrendous. So, nio4r doesn't use it AFAIK.

Do you have explicit reactor, or is it implicit per-thread or per-process?

#33 - 01/27/2018 12:56 AM - normalperson (Eric Wong)

Eric Wong normalperson@yhbt.net wrote:

> hsbt@ruby-lang.org wrote:
>
> - Name this "Thread", or something Thread-ish word than Fiber-ish
>
> So if we just use "Thread", then existing Thread becomes M:N?
> I will think about that... I have many use cases for native threads, too; but maybe they can be satisfied transparently.

Thinking about this even more; I don't think it's possible to preserve round-robin recv_io/accept behavior I want from blocking on native threads when sharing descriptors between multiple processes.

So a new class it is...

- Matz doesn't have a strong opinion on the name but prefers 2 words (auto-fiber) than a coined word "Thriber."

Next actions:

- Give a thread-ish name

> OK, naming is hard :<
>
> LightThread? Maybe too long...
>
> Threadlet?

OK, I am liking "threadlet", and it looks like a real word:

https://www.merriam-webster.com/dictionary/threadlet
": a small thread : a delicate filament"

- Lock and queue should work with auto-fiber?

> I can definitely make Queues work. I think ko1 was mildly against increasing use of Mutex.

How about we use Threadlet to discourage things we don't like about normal Threads (such as Mutex, ConditionVariable, ...).

> One safety feature I was thinking about was disabling auto-switching of Fibers while a Mutex is locked, even.

s/Fibers/Threadlets/; but yes, I think it should be possible to have something like Threadlet.exclusive { ... } to prevent auto-switch surprises (like Thread.exclusive in 1.8)

#34 - 01/27/2018 12:56 AM - normalperson (Eric Wong)

> Thinking about this even more; I don't think it's possible to preserve round-robin recv_io/accept behavior I want from blocking on native threads when sharing descriptors between multiple processes.

```
The following example hopefully clarifies why I care about
maintaining blocking I/O behavior in some places despite relying
on non-blocking I/O for light-weight threading.

# With non-blocking accept; PIDs do not share fairly:
$ NONBLOCK=1 ruby fairness_test.rb
PID    accept count
5240   55
5220   42
5216   36
5242   109
5230   57
5208   26
5227   53
5212   26
5223   46
5236   43
total: 493

# With blocking accept on Linux; each process gets a fair share:
$ NONBLOCK=0 ruby fairness_test.rb
PID    accept count
5271   50
5278   50
5275   50
5282   49
5286   49
5290   49
5295   49
5298   49
5303   49
5306   49
total: 493

For servers which only handle one client-per-process (e.g.
Apache prefork), unfairness is preferable because the busiest
process will be hottest in CPU cache.

For everything else that serves multiple clients in a single
process, fair sharing is preferable.  This will apply to Guilds
in the future, too.

More information about this behavior I rely on is here:
http://www.citi.umich.edu/projects/linux-scalability/reports/accept.html


require 'socket'
require 'thread'
require 'io/nonblock'
Thread.abort_on_exception = STDOUT.sync = true
host = '127.0.0.1'
srv = TCPServer.new(host, 0)
srv.nonblock = true if ENV['NONBLOCK'].to_i != 0
port = srv.addr[1]
pipe = IO.pipe
nr = 10
running = true
```

```
trap(:INT) { running = false }
pids = nr.times.map do
fork do
pipe[0].close
q = Queue.new # per-process Queue
Thread.new do # dedicated accept thread
q.push(srv.accept) while running
q.push(nil)
end
while accepted = q.pop
 # n.b. a real server would do processing, here, maybe spawning
 # a new Thread/Fiber/Threadlet
pipe[1].write("#$$ #{accepted.fileno}\n")
accepted.close
end
end
end
pipe[1].close

sleep(1) # wait for children to start
cleanup = SizedQueue.new(1024)
Thread.new do
cleanup.pop.close while true
end

Thread.new do
loop do
cleanup.push(TCPSocket.new(host, port))
sleep(0.01)
rescue => e
break
end
end
Thread.new { sleep(5); running = false }

counts = Hash.new(0)
at_exit do
tot = 0
puts "PID\taccept count"
counts.each { |pid, n| puts "#{pid}\t#{n}"; tot += n }
puts "total: #{tot}"
end
case line = pipe[0].gets
when /\A(\d+) /
counts[$1] += 1
else
running = false
Process.waitall
end while running
```

#35 - 01/27/2018 12:57 AM - subtileos (Daniel Ferreira)

Hi Eric,

I've been reading this issue and I'm finding it fascinating.
Let me play here the role of the ruby developer that is seeking to
understand better the asynchronous ruby capabilities.
Every time I read threads(conversations) like this one about the pros
and cons of Fibers vs Threads I tend to think: stay away from it.

When people like Kochi write comments like this:

> "But most (many? some? a few?) of ruby programmer (including me) can not write correct code I believe."

or Yusuke Endoh:

> "Thread is considered harmful. Casual Rubyists (including I) had better not use it."

what these comments make us mere mortals feel?

I will speak about me. When I read such a line I tend to step away.
So yes, this situation makes me develop single threaded code as much
as possible.

I rely on libraries to handle asynchronous behaviour for me and
specially I rely extensively on the actor model.

I doubt I will change my mind unless I start to read that Thread is
good to be used or Fiber is good to be used.

When I read all this conversation and you mention corner cases that
still have problems that is a NO GO for me.

IMHO to add yet another Thread like feature it should be "The Killer Feature".

The one that we can say to the all community: Hey people use this
thing because async is a paradise in ruby land at last.
If we don't have this it will be just another Thread, Fiber nightmare
for the very few who accept the overhead of dealing with all the
"buts".

And for the record, I use async libraries but I don't feel confident
about them either knowing that ruby core is not reliable in itself.
Production code in the enterprise world it is not something to mess around.
For me ruby core needs desperately to change this situation so I
really hope your work will be the answer for all of this I'm talking
about.
So yes, if it is it fits in ruby core like a glove IMO. If it is not
then we will be much worst because instead of 2 walking deads we will
have 3.
A 50% increase is a lot in this domain. Turns things into a joke.

So, can you please explain us what peace of mind will we gain with
this new "light thread" in our everyday work?

Thank you very much and keep up the excellent work.
I appreciate specially the care you have in passing across your
knowledge on the subject.
Really helpful and insightful.

Note:

Your last two messages are not part of the issue in redmine. I hope my
message will be there!

On Wed, Jan 24, 2018 at 10:01 PM, Eric Wong normalperson@yhbt.net wrote:

> Thinking about this even more; I don't think it's possible to
> preserve round-robin recv_io/accept behavior I want from
> blocking on native threads when sharing descriptors between
> multiple processes.

```
The following example hopefully clarifies why I care about
maintaining blocking I/O behavior in some places despite relying
on non-blocking I/O for light-weight threading.

# With non-blocking accept; PIDs do not share fairly:
$ NONBLOCK=1 ruby fairness_test.rb
PID     accept count
5240    55
5220    42
5216    36
5242    109
5230    57
5208    26
5227    53
5212    26
5223    46
5236    43
total: 493

# With blocking accept on Linux; each process gets a fair share:
$ NONBLOCK=0 ruby fairness_test.rb
PID     accept count
5271    50
5278    50
5275    50
5282    49
```

```
5286    49
5290    49
5295    49
5298    49
5303    49
5306    49
total: 493
```

For servers which only handle one client-per-process (e.g.
Apache prefork), unfairness is preferable because the busiest
process will be hottest in CPU cache.

For everything else that serves multiple clients in a single
process, fair sharing is preferable. This will apply to Guilds
in the future, too.

More information about this behavior I rely on is here:
http://www.citi.umich.edu/projects/linux-scalability/reports/accept.html

```
require 'socket'
require 'thread'
require 'io/nonblock'
Thread.abort_on_exception = STDOUT.sync = true
host = '127.0.0.1'
srv = TCPServer.new(host, 0)
srv.nonblock = true if ENV['NONBLOCK'].to_i != 0
port = srv.addr[1]
pipe = IO.pipe
nr = 10
running = true
trap(:INT) { running = false }
pids = nr.times.map do
  fork do
    pipe[0].close
    q = Queue.new # per-process Queue
    Thread.new do # dedicated accept thread
      q.push(srv.accept) while running
      q.push(nil)
    end
    while accepted = q.pop
      # n.b. a real server would do processing, here, maybe spawning
      # a new Thread/Fiber/Threadlet
      pipe[1].write("#$$ #{accepted.fileno}\n")
      accepted.close
    end
  end
end
pipe[1].close

sleep(1) # wait for children to start
cleanup = SizedQueue.new(1024)
Thread.new do
  cleanup.pop.close while true
end

Thread.new do
  loop do
    cleanup.push(TCPSocket.new(host, port))
    sleep(0.01)
  rescue => e
    break
  end
end
Thread.new { sleep(5); running = false }

counts = Hash.new(0)
at_exit do
  tot = 0
  puts "PID\taccept count"
  counts.each { |pid, n| puts "#{pid}\t#{n}"; tot += n }
  puts "total: #{tot}"
end
case line = pipe[0].gets
when /\A(\d+) /
```

```
  counts[$1] += 1
else
  running = false
  Process.waitall
end while running
```

#36 - 01/27/2018 12:58 AM - normalperson (Eric Wong)

danieldasilvaferreira@gmail.com wrote:

> Hi Eric,
>
> I've been reading this issue and I'm finding it fascinating.
> Let me play here the role of the ruby developer that is seeking to
> understand better the asynchronous ruby capabilities.
> Every time I read threads(conversations) like this one about the pros
> and cons of Fibers vs Threads I tend to think: stay away from it.
>
> When people like Kochi write comments like this:
>
> > "But most (many? some? a few?) of ruby programmer (including me) can not write correct code I believe."
>
> or Yusuke Endoh:
>
> > "Thread is considered harmful. Casual Rubyists (including I) had better not use it."
>
> what these comments make us mere mortals feel?

Often, you will not have to think about things like Threads or
Fibers; and you may use them every day without knowing it.
Fwiw, every project screws up threading (and many other things)
sometimes; even scanning LKML from the past month I see several
subjects with "race condition" in them.

> I will speak about me. When I read such a line I tend to step away.
> So yes, this situation makes me develop single threaded code as much
> as possible.
> I rely on libraries to handle asynchronous behaviour for me and
> specially I rely extensively on the actor model.

Threadlets/Fibers/Threads can all support the actor model. This
is why I lean towards supporting Queue/SizedQueue but am not as
enthusiastic about increasing scope of Mutex/ConditionVariable.

Threadlet can easily become Actors if matz or ko1 decides to
make such an API. The implementation details which exist today
would barely change.

I'm not a computer language person; to me it's all just bytes in
memory. The key difference is "native thread" has support from
an external layer, the kernel, whereas "userspace" Fiber/Threadlet
are invisible to the kernel.

Any actor API can be either "native" or not, or hybrid (M:N threading).
I believe M:N is too unpredictable/controllable to the
programmer (but I could be wrong).

> I doubt I will change my mind unless I start to read that Thread is
> good to be used or Fiber is good to be used.
>
> When I read all this conversation and you mention corner cases that
> still have problems that is a NO GO for me.

I think the only corner case I mentioned was for libkqueue;
which only affects Linux developers who want to support
some *BSD-specific code without installing FreeBSD.

Normal users won't be expected to use libkqueue.

> IMHO to add yet another Thread like feature it should be "The Killer Feature".

No, what I work towards are incremental improvements and
regression fixes. So I consider Threadlet a regression fix for
the lightweight Thread we lost in the MRI 1.8 -> YARV (1.9) change.
It is also an opportunity to improve on what 1.8 had with better
scalability and more predictable (safer) behavior.

> The one that we can say to the all community: Hey people use this
> thing because async is a paradise in ruby land at last.

I would never say anything that optimistic :P

> If we don't have this it will be just another Thread, Fiber nightmare
> for the very few who accept the overhead of dealing with all the
> "buts".

Huh? If you don't like something, you can ignore them and let
others use/try them. There's plenty of things I don't care for
in Ruby, too. Sometimes we can deprioritize/deprecate them,
make them less intrusive and move on (see 'callcc', $SAFE, taint).

> And for the record, I use async libraries but I don't feel confident
> about them either knowing that ruby core is not reliable in itself.

I'm not sure what you're talking about. I suppose nothing is
reliable :P For example, see how often "stable" Linux kernel
releases come out with GregKH saying "all users must upgrade".
Yet Linux is trusted with countless mission critical systems.

Best we can do is fix bugs and learn lessons from them to avoid
repeating history.

And life goes on...

> Production code in the enterprise world it is not something to mess around.
> For me ruby core needs desperately to change this situation so I
> really hope your work will be the answer for all of this I'm talking
> about.
> So yes, if it is it fits in ruby core like a glove IMO. If it is not
> then we will be much worst because instead of 2 walking deads we will
> have 3.
> A 50% increase is a lot in this domain. Turns things into a joke.

Did you see my other post about blocking accept? I have every
intent to continue using Thread as-is; and I also use Fiber
as-is in places where it is the perfect tool for the job.

They each have their uses.

And I also look forward to Guilds, too; which I expect to be
implemented using native threads but with less sharing visible
to the Ruby layer.

> So, can you please explain us what peace of mind will we gain with
> this new "light thread" in our everyday work?

But often I want something in-between what Thread and Fiber are,
and that's where Threadlet comes in.

> Thank you very much and keep up the excellent work.
> I appreciate specially the care you have in passing across your
> knowledge on the subject.
> Really helpful and insightful.

You're welcome.

> Note:

> Your last two messages are not part of the issue in redmine. I hope my message will be there!

These two?
http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/85081
http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/85082

Maybe it's a bug in Redmine mailing list integration plugin,
Will try to get with hsbt (Hiroshi SHIBATA) to track it down...

#37 - 01/27/2018 12:58 AM - subtileos (Daniel Ferreira)

Eric Wong normalperson@yhbt.net wrote:

> These two?

Yes Eric. And the last one as well. And I guess this here that I will send will happen the same.
I believe it will be better to not reply to way while this is broken.
Which is a petty since I have some things to say but I believe it will be better to wait. :-|

#38 - 01/27/2018 01:02 AM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> In async, I called it Async::Task. I think task is a good name for this kind of thing. In your case, you might want to consider Thread::Task. Since, the lexicographic nesting is similar to the logical nesting.

I prefer shorter names; and I'm not sure if Thread::Task makes sense since it's an alternative to Thread (in some situations); not a helper to Thread (unlike Mutex/Queue/etc).

> Regarding kqueue bugs. macOS kqueue implementation is horrendous. So, nio4r doesn't use it AFAIK.

Yes, there's also a select() implementation which should be a safe fallback for everybody (not scalable, of course). I'm not sure if OpenBSD/NetBSD/Dragonfly have acceptable kqueue implementations, nowadays, either (FreeBSD seems fine).

I will add notes to guide porters into disabling kqueue support, either broadly or fine-grained (per-type), or better, eventually fixing their native kqueue implementations.

I also intend to try aio-poll support in future Linux versions (currently under development).

> Do you have explicit reactor, or is it implicit per-thread or per-process?

Implicit per-process, and lazily created. kqueue and epoll persistent data structures in the kernel are completely safe to use across multiple threads. select needs no persistent structure in the kernel. Userspace structures are of course done in a thread-safe way and will be adjusted for guilds or GVL removal.

If guilds end up being what I expect them to be (implemented via native threads), reactor will likely remain per-process since FDs are still per-process. Some structures and locking will be adjusted for guilds, of course.

#39 - 01/27/2018 01:02 AM - subtileos (Daniel Ferreira)

Hi Eric,

It is really a shame that your replies in this thread are not being

added to the issue tracker.
Samuel's reply is there but your reply once again didn't get in.

Please try to do something about it because the conversation will be
lost in the future if nothing is done on that respect.

On my side, I will continue to wait that the problem can be corrected
in order to continue to give my contribution to it.

Many Thanks,

On Fri, Jan 26, 2018 at 7:13 PM, Eric Wong normalperson@yhbt.net wrote:

> samuel@oriontransfer.org wrote:

#40 - 01/27/2018 01:02 AM - normalperson (Eric Wong)

Daniel Ferreira subtileos@gmail.com wrote:

> Please try to do something about it because the conversation will be
> lost in the future if nothing is done on that respect.

I've contacted hsbt (Hiroshi SHIBATA) about it, be patient as he is busy.

#41 - 01/27/2018 01:08 AM - hsbt (Hiroshi SHIBATA)

normalperson (Eric Wong) wrote:

> Daniel Ferreira subtileos@gmail.com wrote:
>
> > Please try to do something about it because the conversation will be
> > lost in the future if nothing is done on that respect.
>
> I've contacted hsbt (Hiroshi SHIBATA) about it, be patient as he is busy.

Hi, I've restored missing comments on redmine from our mailing list.
It's affected by server maintenance and has some issues with server configuration.

#42 - 01/27/2018 01:17 AM - dsferreira (Daniel Ferreira)

hsbt (Hiroshi SHIBATA) wrote:

> I've restored missing comments on redmine

Thank you very much Hiroshi.
Feels much better now.

#43 - 01/27/2018 03:45 AM - dsferreira (Daniel Ferreira)

normalperson (Eric Wong) wrote:

> I'm not sure what you're talking about. I suppose nothing is reliable

Let me try to explain what I think about the async subject in ruby land using a different story:

For me there is ruby core and there is ruby.
I'm a ruby kind of guy like most of ruby developers.
I like to use ruby and I like to use it as it is given to us by ruby core.
I prefer to build my own tools in top of ruby core rather than using external libraries/gems.
That with the assumption that ruby core will not break backwards compatibility.
If there is something I really dislike is to fix broken code due to dependency issues.

Ruby core is the rock solid foundation I rely upon for the developments I design and implement.

I started in ruby land with rails like most of us.
As years passed by I went more and more to other territories.
So I believe my story it is a very common story:
The ruby developer that starts at the very high level with rails.

With time becomes progressively more and more familiar with the low level concepts of programming.
Gets to understand the underlying concepts behind the frameworks and starts to grasp at last the ruby essence.
And here I am now speaking with you guys.
Ruby core. The lower level by excellence.
It is fascinating to go through discussions like this one.
The craft of ruby landscape for the future.
Technically speaking I'm learning a lot but I'm not prepared yet to give my contribution at that level.
The contribution I believe I can give is this view I'm speaking about.
The daily user that sometimes struggle to find the right paths for the problems in hands.

Ruby developers like my self (I imagine there will be more that feel this way) are very much impacted by the opinions of ruby core team members.
Specially top team members like Koichi.
We can call it the teacher - student dichotomy.

When Koichi referring to threads functionality in ruby land writes and says:

> "But most (many? some? a few?) of ruby programmer (including me) can not write correct code I believe."

I do listen. People listen.

(Koichi sentence here is just an handy reference example (sorry Koichi), from the many I have read throughout this many years and many of those comments are here, embedded in redmine issues).

These sentences have a very big impact.

I as a programmer aim to write and develop correct code.
If there is an area that I do not feel comfortable with then I study it, play with it but that is it.
I will not put my job and my company in jeopardy just to show some cool stuff to the team.
Ruby programmer not ruby core hacker remember?

How many ruby developers develop a http server or know the internals of at least one?
(Just as an example of different levels of developer seniority.)
Unicorn or passenger or thin or puma... are black boxes for the most of us.
And yes there are bugs and our applications are impacted by them.
That is the ecosystem and it is good like that. It will not change.

Somehow there are people that feel happy playing in dangerous zones like threads and fibers
(See previous Koichi reference. We know you have a "slightly" different opinion).
Us, mere mortals, just would like to be able to do our daily work at least without compromising.
Although I would like to use libraries to play with my actors without worrying to much I can't.
Knowing that they are dangerous zones tells me I must worry still.

So, why not use Akka and live happy ever after?
In Akka land everyone happily uses actors.
I never heard any reference telling people to be careful about a given issue.
Maybe the issues exist but what you read is that Akka is the solution for all your problems in async world.

I don't want to use Akka but I know that ruby is losing developers every day because of situations like this one I'm referring here.
Ruby desperately needs to resolve once and for all this situation.

The key word for me here is a clear message that could say with confidence:

"Ruby is rock solid for async because..."

If we don't succeed to pass this message to the world of programming ruby will slowly be replaced by other languages.
Parallelism and concurrency and async will be everywhere in the future.

I took the decision to express this thoughts in this conversation because I love ruby and I want to help ruby become better.

In my opinion:

We need to create the foundations for a post ruby 3 future in ruby land where async is the standard for the many and not the exception for the few.

That is my vision.

Many Thanks,

Daniel

#44 - 01/27/2018 11:34 PM - jeremyevans0 (Jeremy Evans)

dsferreira (Daniel Ferreira) wrote:

> We need to create the foundations for a post ruby 3 future in ruby land where async is the standard for the many and not the exception for the few.
>
> That is my vision.

According to the tagline on the homepage, ruby is "A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write." Asynchronous code tends to negatively affect simplicity and productivity in order to gain performance, and in general is more difficult to read, more difficult to write, and more difficult to test than synchronous code. To the extent that ruby can improve its support for asynchronous code without compromising its other values, I would probably support it (this feature is one of those cases). However, we should be careful to never sacrifice ruby's core values just to improve support for whatever programming paradigm is currently popular in some other programming subcultures.

In the future, for general philosophical discussion of ruby's goals and future direction, it is probably best to email ruby-core@ruby-lang.org directly. If you have a specific new feature or change in mind, then add it a new feature request. I think we should try to avoid adding tangentially-related philosophical discussion posts as notes on existing features/bugs.

#45 - 01/28/2018 12:02 AM - dsferreira (Daniel Ferreira)

jeremyevans0 (Jeremy Evans) wrote:

> we should be careful to never sacrifice ruby's core values

I couldn't agree more.

> If you have a specific new feature or change in mind, then add it a new feature request.

Yes I do Jeremy.
Eric's light thread it was a good starting point for this discussion but I will present something more concrete in a new issue and will link it to the different issues I believe are related to it.

#46 - 01/28/2018 12:33 AM - dsferreira (Daniel Ferreira)

normalperson (Eric Wong) wrote:

> How about we use Threadlet

IMO the name we will chose will be more important then the functionality in itself.
It needs to stand out and create a clear picture in our mind.
Thread, Fiber, Guild? (not so sure about this name either), ?
We will have four entities on our async family.
Each name should be clearly sound.

Light Thread maps well in my mind.
Matz said he preferes two words. I would prefer a single word that could draw the same picture as "LightThread".
No mixtures between Thread and Fiber. That would be saying that the feature is linked to them.
IMO the message should be that this feature can be used by its own independently.
A clear distinction that will put aside any links to Threads and Fibers dos and don'ts.
It would be a good first step towards a smoother async ecosystem.

For all these reasons I would like to propose for the "Light Thread" feature the name:

"Strand"

- Strand definition: a thin thread of something, often one of a few, twisted around each other to make a string or rope.
- Strand gem is not used (only 0.1.0) so we can claim the name. See: https://rubygems.org/gems/strand.

#47 - 01/28/2018 05:20 AM - sam.saffron (Sam Saffron)

Hmmm, what about just bringing in the IO Manager APIs including Ruby helpers prior to re-introducing the green threads?

As it stands kqueue/epoll abstractions always require another fat dependency and there is no official API to consume them.

Even just solving this problem is enough of a hornets nest prior to introduction of other complications.

epoll is notoriously monstrous, http://cvs.schmorp.de/libev/ev_epoll.c?view=markup so having an officially supported abstraction would be a great start.

Wouldn't having these abstractions allow building this by hand using existing Fiber?

#48 - 01/28/2018 10:51 AM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> Hmmm, what about just bringing in the IO Manager APIs including Ruby helpers prior to re-introducing the green threads?

One big problem I notice with existing IO manager APIs (libev/libevent/EventMachine) is multi-threading was as an afterthought to them. As in, throw a lock around a single-threaded event loop and call it a day.

Ruby was this way, too; but want to work towards changing that and embracing the multi-thread friendliness baked into APIs provided by kqueue and epoll.

Btw, some of the discussion/planning around this started in: https://public-inbox.org/ruby-core/20170402023514.GB30476@dcvr/t/

> As it stands kqueue/epoll abstractions always require another fat dependency and there is no official API to consume them.

I don't know if exposing a new API around them is desirable. For human-friendliness, it seems desirable to keep the Ruby API synchronous even if internal bits become async.

I think it's also desirable to be able to change some/most existing Thread uses to auto-Fiber/Threadlet/Thriber without having to re-design things, just changing "Thread.new" to something else.

> Even just solving this problem is enough of a hornets nest prior to introduction of other complications.
>
> epoll is notoriously monstrous, http://cvs.schmorp.de/libev/ev_epoll.c?view=markup so having an officially supported abstraction would be a great start.

I disagree. IMHO, Lehman's notes and complaints against epoll are either out-of-date or his mental model went wrong somewhere. Fwiw, fs/eventpoll.c is straightforward and easy-to-understand in git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

> Wouldn't having these abstractions allow building this by hand using existing Fiber?

One question is, how painful will it be in Ruby?

I've kinda soured on _nonblock APIs in Ruby over the years. For example, in https://bugs.ruby-lang.org/issues/14404 I don't think there's a non-painful Ruby way to resume a partial writev. Doable, of course, but it requires extra allocations and copies. Resuming a partial write_nonblock today without writev isn't great, either...

With a synchronous interface (IO#write), dealing with partial writev in C is only a few adds/subracts; and we wont expose pointer arithmetic in Ruby :)

And then there's also stuff like IO.copy_stream not having a _nonblock analogue...

#49 - 01/28/2018 10:51 AM - normalperson (Eric Wong)

danieldasilvaferreira@gmail.com wrote:

> When Koichi referring to threads functionality in ruby land writes and says:
>
> > "But most (many? some? a few?) of ruby programmer (including me) can not write correct code I believe."
>
> These sentences have a very big impact.

They should not have a big impact. Really make up your
own mind on these things instead just believing somebody;
even if they are a leader of this project.

I suspect if you look at any development archives for any major
projects; you will see similar statements from major contributors.

> The key word for me here is a clear message that could say with confidence:
>
> "Ruby is rock solid for async because..."

Saying something like that would open us up to lawsuits.
The following (or similar) disclaimer is in every project
I work on:

1. THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

So, no, I'm never going to say anything I work on is "rock solid".

> If we don't succeed to pass this message to the world of
> programming ruby will slowly be replaced by other languages.

In my experience, people left Ruby because incompatibilities got
painful and memory usage was too high.

Ruby 1.8 green Threads were a middle ground. Since 1.9+, Fibers
went in one direction (harder-to-use), while native Threads
went in another direction (too heavy); and there's nothing left
in the middle.

All I aim to do with this feature is fill the void in the middle.

> Parallelism and concurrency and async will be everywhere in the future.

They already are, and have been for a while.

> We need to create the foundations for a post ruby 3 future in
> ruby land where async is the standard for the many and not the
> exception for the few.

Internal implementation can be async, but the public API will
likely remain and favor synchronous (because redesiging existing
libs is expensive).

New features should always be opt-in, never a requirement.
That said, it should still be easy to port code over to take
advantage of new features; so I want to minimize publically
visible changes.

#50 - 01/28/2018 11:08 AM - normalperson (Eric Wong)

danieldasilvaferreira@gmail.com wrote:

> For all these reasons I would like to propose for the "Light Thread" feature the name:
>
> "Strand"

No, I don't want to introduce a non-obvious term nobody has seen
before in concurrency. I expect "Strand" will be mistaken for some
thing String-related. (String and Thread are also interchangeable
in English).

Anyways, I think Threadlet is an acceptable name.

#51 - 01/28/2018 11:08 AM - normalperson (Eric Wong)

Eric Wong normalperson@yhbt.net wrote:

- Matz doesn't have a strong opinion on the name but prefers 2 words (auto-fiber) than a coined word "Thriber."

Next actions:

- Give a thread-ish name

Threadlet?

OK, I changed to Threadlet for now.

- Lock and queue should work with auto-fiber?

I can definitely make Queues work. I think ko1 was mildly
against increasing use of Mutex.

One safety feature I was thinking about was disabling
auto-switching of Fibers while a Mutex is locked, even.

Still TODO; I don't expect much time for more development
until March; but maybe I'll find pockets of time here and
there (much of the other work I do here is while procrastinating)

Anyways, rebased against r62077:

The following changes since commit 46bfa65fccf58cee280bf552193f93388b00d16d:

internal.h: add BITFIELD macro to aid C99 users (2018-01-27 21:04:42 +0000)

are available in the Git repository at:

git://80x24.org/ruby threadlet-r62077

for you to fetch changes up to 6b5c8ba6cbfd33d557748cad6ef4928332893083:

threadlet: non-native fiber support (2018-01-28 10:31:48 +0000)

Raw patches here:
https://80x24.org/spew/20180128103907.12069-2-e@80x24.org/raw
https://80x24.org/spew/20180128103907.12069-3-e@80x24.org/raw

#52 - 01/28/2018 12:29 PM - dsferreira (Daniel Ferreira)

normalperson (Eric Wong) wrote:

They should not have a big impact.

Playing the ruby developer role here remember? Do you think most ruby developers don't care about those statements?
What about good and straight forward async guides for regular ruby developer users?
Documentation and evangelism is a very important part of a language.
If we have the features but they are not very well explained to the main public the features become a knowledge to the few.
We need to put outside to the public a better message.
I'm planning to work on that as well and help fix it by the way.

So, no, I'm never going to say anything I work on is "rock solid".

That is called an hyperbole if it was not obvious. Sometimes I use them to emphasise a certain point.

In my experience, people left Ruby because incompatibilities got painful and memory usage was too high.

Fair enough. It would be interesting to research more about the subject. Is there any discussion on the subject in ruby core?
Stoically we are holding our position but losing to Python big time.
If it was me I would target Python as a reference.

Parallelism and concurrency and async will be everywhere in the future.

They already are, and have been for a while.

Yeah. Now I was being defensive. :-)
The "everywhere" might put some people arguing.
We can not fail with ruby 3. That was the main point of my alert.
But I agree with you that ruby is already behind the point where it should be.

> Internal implementation can be async, but the public API will likely remain and favor synchronous

That is inline with what I have in my mind.
The issue I'm planning to open is about that.

> New features should always be opt-in, never a requirement.

Totally agree.

#53 - 01/28/2018 02:08 PM - ko1 (Koichi Sasada)

On 2018/01/25 6:51, Eric Wong wrote:

> Threadlet?
> OK, I am liking "threadlet", and it looks like a real word:
>
> https://www.merriam-webster.com/dictionary/threadlet
> ": a small thread : a delicate filament"

Another idea is to pass option to Thread.new() like
Thread.new(preemption: false).

Note: we can't pass thread options because of args/keywords spec.
It was pseudo code.
Note2: I agree it can be confusing with mixing normal Threads and
Threadlet. It likes proc and lambda.

--
// SASADA Koichi at atdot dot net

#54 - 01/28/2018 02:12 PM - ko1 (Koichi Sasada)

On 2018/01/25 7:01, Eric Wong wrote:

> For everything else that serves multiple clients in a single
> process, fair sharing is preferable.

Could you elaborate more? Generally, fairness is preferable. But I think
we can document "we don't guarantee fairness scheduling on this
feature", because our motivation is to provide a way to process multiple
connections. Thoughts?

Or dose it cause live-lock? (no-problem on server-client apps, but
multi-agents programs seems to cause live locking)

--
// SASADA Koichi at atdot dot net

#55 - 01/28/2018 02:41 PM - ko1 (Koichi Sasada)

On 2018/01/24 2:31, Eric Wong wrote:

> - Lock and queue should work with auto-fiber? I can definitely make Queues work. I think ko1 was mildly against increasing use of Mutex.
>
> One safety feature I was thinking about was disabling
> auto-switching of Fibers while a Mutex is locked, even.

If we name it as Thread-like (Threadlet), we can use all synchronization
tools with Threads (I feel it is natural). I'm not sure we should limit
to use them on Threadlet or not.

1. Threads and Threadlets can share same synchronization tools
   -> Good: no learning efforts
   -> Bad: People can cause sync issues with mis-using or missing syncs

2. Introduce Threadlets special synchronization tools and introduce
   special rules communicate with other threads
   -> Good: people can only use good tools (such as Queues)
   -> Bad: we need to learn new tools and rules

If we think Threadlet is a special Thread (and the name indicates it),
then (1) seems nice for me.

With both options, we can enjoy advantages of Threadlet:
(a) lightweight creation
(b) predictable (than preemptive threads) switching

--
// SASADA Koichi at atdot dot net

#56 - 01/28/2018 05:50 PM - dsferreira (Daniel Ferreira)

ko1 (Koichi Sasada) wrote:

> I'm not sure we should limit to use them on Threadlet or not.
>
> 1. Threads and Threadlets can share same synchronization tools
>   -> Good: no learning efforts
>   -> Bad: People can cause sync issues with mis-using or missing syncs
>
> 2. Introduce Threadlets special synchronization tools and introduce special rules communicate with other threads
>   -> Good: people can only use good tools (such as Queues)
>   -> Bad: we need to learn new tools and rules

I'm all for (2) for the reasons I already mentioned:

- Specially the big minus that we have in (1): "People can cause sync issues"
- Using only good tools is a big +.
- Not causing sync issues is a big ++.
- The fact that people will be forced to learn new tools and rules is also a big + for me.
  - It draws the border between the old async scenario and the new one we are trying to implement.

> If we think Threadlet is a special Thread (and the name indicates it),
> then (1) seems nice for me.

I agree Threadlet has that implication.

Since we prefer to use names already in use in the async world what about call it:

Lane

- Lua is always a source of inspiration to me.
- Lanes is a lightweight, native, lazy evaluating multithreading library for Lua.
- Lane meaning: a narrow road or division of a road
- Lane gem (v0.1.0). 247 downloads. https://rubygems.org/gems/lane.

The sense of speed and direction pleases me a lot.

Note:

About Threads vs Lanes in Lua

```
LuaThread provides thread creation...and need therefore to be guarded against multithreading conflicts.

Whether this is exactly what you want, or whether a more loosely implemented
multithreading (s.a. Lanes) would be better, is up to you. One can argue that
a loose implementation is easier for the developer, since no application level
lockings need to be considered.
```

#57 - 01/28/2018 08:03 PM - normalperson (Eric Wong)

Koichi Sasada ko1@atdot.net wrote:

> On 2018/01/25 7:01, Eric Wong wrote:
>
> > For everything else that serves multiple clients in a single
> > process, fair sharing is preferable.
>
> Could you elaborate more? Generally, fairness is preferable. But I think we
> can document "we don't guarantee fairness scheduling on this feature",
> because our motivation is to provide a way to process multiple connections.
> Thoughts?

If I write a multi-process server with many long-lived
connections, it's best to balance those connections to mitigate
bottlenecks/problems which exist in each process. That way, any
slowdown or crash which affects one process only affects
its fair subset of connections.

This is fair sharing across different *nix processes...
Within each process, Threadlets are also round-robin scheduled,
but run until they cannot proceed.

> Or dose it cause live-lock? (no-problem on server-client apps, but
> multi-agents programs seems to cause live locking)

It should not, Threadlet is FIFO for "ready" Fibers;
epoll and kqueue are readiness queues are FIFO internally, too.

Blocking accept() mitigates live-lock/thundering herd across
different processes. For non-blocking accept(), I will add
EPOLLEXCLUSIVE support.

#58 - 01/28/2018 08:21 PM - normalperson (Eric Wong)

danieldasilvaferreira@gmail.com wrote:

> ko1 (Koichi Sasada) wrote:
>
> > I'm not sure we should limit to use them on Threadlet or not.
> >
> > 1. Threads and Threadlets can share same synchronization tools
> > -> Good: no learning efforts
> > -> Bad: People can cause sync issues with mis-using or missing syncs
> >
> > 2. Introduce Threadlets special synchronization tools and introduce special rules communicate with other threads
> > -> Good: people can only use good tools (such as Queues)
> > -> Bad: we need to learn new tools and rules
>
> I'm all for (2) for the reasons I already mentioned:
>
> - Specially the big minus that we have in (1): "People can cause sync issues"
> - Using only good tools is a big +.
> - Not causing sync issues is a big ++.
> - The fact that people will be forced to learn new tools and rules is also a big + for me.
>   - It draws the border between the old async scenario and the new one we are trying to implement.

No, I'm against making major changes. For 2, I mean we limit
usage to queues for now, which is a a subset of 1; but I'm also
OK implementing mutex/condvar support for 1.

Having less things to learn is better for adoption and improving
usefulness

> > If we think Threadlet is a special Thread (and the name indicates it),
> > then (1) seems nice for me.
>
> I agree Threadlet has that implication.
>
> Since we prefer to use names already in use in the async world
> what about call it:

Lane

Too obscure and not obvious for me; do non-Lua people know about it?

Terms such as process, thread, task, actor are already in wide use
across several different languages; so it should be obvious.

- Lane meaning: a narrow road or division of a road

When comparing to physical objects, it seems more appropriate for
something like a channel or pipe.

#59 - 01/28/2018 08:43 PM - dsferreira (Daniel Ferreira)

normalperson (Eric Wong) wrote:

> No, I'm against making major changes. For 2, I mean we limit
> usage to queues for now, which is a a subset of 1; but I'm also
> OK implementing mutex/condvar support for 1.
>
> Having less things to learn is better for adoption and improving
> usefulness

I would agree with that comment if the "less" doesn't imply in itself an overlap of confusions.
How will be the documentation? We need to think very careful about that.

> Too obscure and not obvious for me; do non-Lua people know about it?

Do we have Threadlets in other languages?
It seems Lua has got something very similar (how similar?) and calls it Lanes.
Am I wrong with this assumption?

> When comparing to physical objects, it seems more appropriate for
> something like a channel or pipe.

In a dedicated Lane I see "vehicles" moving steady and fast in between the traffic chaos.
I consider it a fortunate choice from Lua people.
The notion of async for me is management of traffic in between the chaos.
Why thread? Because it is a kind of channel or pipe as well, isn't it?

#60 - 01/28/2018 09:21 PM - normalperson (Eric Wong)

danieldasilvaferreira@gmail.com wrote:

> normalperson (Eric Wong) wrote:
>
> > No, I'm against making major changes. For 2, I mean we limit
> > usage to queues for now, which is a a subset of 1; but I'm also
> > OK implementing mutex/condvar support for 1.
> >
> > Having less things to learn is better for adoption and improving
> > usefulness
>
> I would agree with that comment if the "less" doesn't imply in itself an overlap of confusions.
> How will be the documentation? We need to think very careful about that.

I prefer minimal documentation and having it do obvious/predictable
things which are already familiar to existing users of Thread.

In my experience, too much documentation overhwhelms users and
they ignore it.

And about the comments you see from developers here: the vast
majority of Ruby users will never read or see them even.
There's too much to read for most people.

> > Too obscure and not obvious for me; do non-Lua people know about it?

> Do we have Threadlets in other languages?
> It seems Lua has got something very similar (how similar?) and calls it Lanes.
> Am I wrong with this assumption?

The "let" suffix is commonly associated with a smaller version
of something; and the "Thread" prefix already exists; so it
should be immediately familiar (at least to English speakers)

> > When comparing to physical objects, it seems more appropriate for
> > something like a channel or pipe.

> In a dedicated Lane I see "vehicles" moving steady and fast in between the traffic chaos.
> I consider it a fortunate choice from Lua people.
> The notion of async for me is management of traffic in between the chaos.
> Why thread? Because it is a kind of channel or pipe as well, isn't it?

Not exactly. Pipes are a type of queue (ring buffer), it is
something which data passes through. Threads/Processes/Fibers
are execution contexts which can use pipes/queues to pass data along.

#61 - 01/29/2018 12:39 AM - sam.saffron (Sam Saffron)

I am not a huge fan of the name threadlet, it just does not sound right.

What if a new construct is introduced:

```
pool = ThreadPool.new(concurrency: 100, max_workers: 5 # optional)

thread = pool.run do
  sleep # thread pool should be aware, this would preempt a context switch to another fiber
end
```

Using this construct one could manage many pools of fibers.

That can simplify all sorts of stuff, like creating a proxy that can only download 3 streams concurrently

```
DOWNLOAD_POOL = ThreadPool.new(concurrency: 3)
def proxy_url(url)

   if DOWNLOAD_POOL.queued > 5
     raise "too many things queued"
   end

   done = Queue.new
   t = DOWNLOAD_POOL.run do
     done << download(url)
   end
   render body: done.pop
end
```

#62 - 01/29/2018 04:51 AM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> I am not a huge fan of the name threadlet, it just does not sound right.

Is "Task" better? Or "CoThread" (like "coroutine").
Actually I don't like "CoThread" much, but "Task" is
short and a somewhat popular name:

```
https://en.wikipedia.org/wiki/Task_(computing)
```

> What if a new construct is introduced:
>
> pool = ThreadPool.new(concurrency: 100, max_workers: 5 # optional)

I really don't like that. It's too much up-front cost to having
to declare a pool ahead-of-time. One thing I love about
Fiber/Thread/fork is they can be used anywhere, even when deep

inside libraries.

That said, glibc has internal caching of thread stacks, and Ruby
also caches Fiber stacks internally, but they're completely
transparent to the user. There's also code for an internal
Thread cache for Ruby, but it's broken with fork and disabled, atm

#63 - 01/29/2018 05:06 AM - sam.saffron (Sam Saffron)

I like Task a lot, it is short and makes much sense.

So conceptually a kernel thread will be allowed to schedule N Tasks.

How would you manage scheduling tasks that are potentially blocking. Should Ruby opt for a goroutine type implementation where core just handles spawning "enough" underlying threads to handle the work, or would the management be at a higher level and you would spawn N threads and then tasks from said threads.

I think it probably makes sense to always have Tasks coupled tightly with threads initially cause debugging will be much simpler.

If this is coupled to Thread does this make sense?

```
t = Thread.new do
   sleep
end

t.add_task do
   # my task
end

Thread.current.add_task do
   # some task
end
```

> even when deep inside libraries.

Note, you only get 1500 or so frames these days on Fiber and over 10k or so on Thread, this will be limited to a degree by Fiber design. This should be plenty for Rails apps that love deep stacks cause I don't think we usually pass 400 or so frames deep these days.

#64 - 01/29/2018 05:21 AM - ko1 (Koichi Sasada)

On 2018/01/29 14:06, sam.saffron@gmail.com wrote:

> I like Task a lot, it is short and makes much sense.

I strongly oppose the name Task because it is ambiguous, many language
(and OSs) uses this word as many purpose.

--
// SASADA Koichi at atdot dot net

#65 - 01/29/2018 05:38 AM - sam.saffron (Sam Saffron)

What about Job?

```
job = Thread.current.queue do
  sleep 100
end

job.cancel
```

#66 - 01/29/2018 09:51 AM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> I like Task a lot, it is short and makes much sense.

I guess there's a risk of namespace conflict with existing
code with such a generic name like "Task" or "Job". But,
maybe the class name should not matter as much as adding
new ones can always cause conflict with existing code.

So, based on your add_task proposal; maybe the name of the

class wouldn't even matter, and we can use whatever name,
(I just chose "async") to create it:

```
foo = Thread.current.async do
   # some task
end

foo.class => RubyVM::ThingWeCannotDecideANameFor

# (Or Thread.async, because only current is supported atm)
foo = Thread.async {}

foo.class => RubyVM::ThingWeCannotDecideANameFor
```

In other words, API for usage and class name can be orthogonal.

> So conceptually a kernel thread will be allowed to schedule N Tasks.

Right.

> How would you manage scheduling tasks that are potentially
> blocking. Should Ruby opt for a goroutine type implementation
> where core just handles spawning "enough" underlying threads
> to handle the work, or would the management be at a higher
> level and you would spawn N threads and then tasks from said
> threads.

That would be M:N threading which I am uncertain about.

Mainly, I want to still be able to do real blocking operations
even when non-blocking operations are supported for sockets:

http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/85082
https://public-inbox.org/ruby-core/20180124220143.GA5600@80x24.org/

(likewise with recv_io or small IO#sysread on IO.pipe)

So "enough" is difficult to determine (not just CPU count).
I have use cases which involve multiple mount points which
I'd like to be able to optimize for with Ruby.

> I think it probably makes sense to always have Tasks coupled
> tightly with threads initially cause debugging will be much
> simpler.

Yes, it's a requirement at the moment since migrating Fibers
across Threads is not possible.

I think we'd have to give up fast native Fiber switching
(ucontext_t) if we want to migrate Fibers across Threads (maybe
ko1 can confirm).

So that's why "rb_thread_t.afrunq" came to be:

> Changes to existing data structures:
>
> ```
> rb_thread_t.afrunq   - list of fibers to auto-resume
> ```

#67 - 01/29/2018 08:56 PM - shan (Shannon Skipper)

Looking at naming in few languages that implement a similar feature, there seems to be no consensus:

- Goroutine (Go)
- Lane (Lua)
- Spark on a Haskell Thread (Haskell)
- Task (Elixir - though there's more is going on here, and Process is really closer)
- Process (Erlang)
- Fiber (Crystal)

Process and Fiber obviously won't work because Ruby already uses these terms. Of the remaining options, Goroutine is the most widely known. I think RubyRoutine is overly verbose, but it'd be easy to explain that a Routine in Ruby is like a Goroutine.

I think Threadlet, Task, Routine, Lane and Spark are all viable options. Most folk won't know what a Lane or Spark are, so I'm not sure there's a big advantage in using one of those names for the sake of consistency across languages.

#68 - 01/29/2018 09:28 PM - sam.saffron (Sam Saffron)

I think Routine is a bit tricky to spell so I would recommend avoiding it. In Go people talk about Goroutines but never actually write it in code. That said, this is pretty hidden.

> In other words, API for usage and class name can be orthogonal.

I agree with this and do not think we can afford some level of extra verbosity here:

I like ThreadTask a lot since these things are coupled with threads. I think ThreadJob works as well.

API wise we can even avoid this altogether with

Thread.current << lambda { }

So we don't even need to think about async vs add_task vs add_job

> Yes, it's a requirement at the moment since migrating Fibers
> across Threads is not possible.

I would like to hear a bit more about this, could there be an "expensive" thread transfer operator added perhaps that only moves when the fiber is suspended?

```
j = Thread.current << lambda { sleep }

Thread.new do
   sleep
end.transfer(j)
```

#69 - 01/29/2018 10:32 PM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> I like ThreadTask a lot since these things are coupled with
> threads. I think ThreadJob works as well.

Maybe we can call it what it is: Thread::Green

I suspect using top-level namespace is unnecessary and may
introduce conflicts.

> API wise we can even avoid this altogether with
>
> Thread.current << lambda { }
>
> So we don't even need to think about async vs add_task vs add_job

I like that.

One question, is how will Thread#[]/#[]= be handled inside
the lambda?

> > Yes, it's a requirement at the moment since migrating Fibers
> > across Threads is not possible.
>
> I would like to hear a bit more about this, could there be an
> "expensive" thread transfer operator added perhaps that only
> moves when the fiber once suspended?

One problem is the act of suspending it (Fiber.yield)
will need to change. Maybe it could default to fast suspend,
and the migrate operation would:

```
1. set a flag to indicate migration in progress
2. resume
3. see + clear migration flag
4. suspend again immediately, but slowly for migration
```

But it's a totally orthogonal issue to auto-fiber

/me goes back to working on non-Ruby stuff...

#70 - 01/31/2018 02:48 AM - ioquatix (Samuel Williams)

> Wouldn't having these abstractions allow building this by hand using existing Fiber?

Yes, it's feasible and already implemented here https://github.com/socketry/async and it's backwards compatible with older Rubies.

> Even just solving this problem is enough of a hornets nest prior to introduction of other complications.

I agree with this, but not for the reason stated. I think modern epoll/kqueue/select basically just work. Yes, there are some odd issues you have to deal with, but for the most part things work well and it's as efficient as it's going to get in a general sense.

What I think is a bigger issue is blocking system calls. An example of this would be system name lookup (e.g. DNS). The two main mitigations are using a threadpool (libuv and neverblock do this AFAIK), or having multiple reactors and migrating other Fibers if the reactor is blocked. Even just having a tight loop can cause problems, and even in the case where you have non-blocking IO, if it never actually blocks and yields back to the reactor.

> pool = ThreadPool.new(concurrency: 100, max_workers: 5 # optional)

It's a bit surprising to see this, but your example is almost exactly the same as using Async::Reactor, simply replace ThreadPool with Async::Reactor and the code will almost work. Semantically it's about the same as what I think is the ideal solution.

I think that abstracting around the Reactor pattern is a good idea. It provides strong guarantees about the state of the program.

Here is the main entry point for an Async::DNS::Server instance:
https://github.com/socketry/async-dns/blob/5ec883c0dd3d69b766668e4e6811561aba847ac6/lib/async/dns/server.rb#L106-L120

Async::Reactor#run handles nesting:
https://github.com/socketry/async/blob/4f695ed6e340031f27f6db5100ab86ba139ae3d9/lib/async/reactor.rb#L38-L61

If you call the run method inside an existing reactor, it returns an async task which you can use to stop the server and all async tasks started within the server. If you call it outside of a reactor, it will create a reactor and block forever. In both cases the life cycle is managed correctly.

Simply making a per-thread reactor and making read/write calls non-blocking only solves about 10% of the problem IMHO.

To compare some of the pseudo examples with real code, take a look at the C10k implemented here:
https://github.com/socketry/async-io/blob/master/spec/async/io/c10k_spec.rb

#71 - 02/02/2018 05:46 AM - sam.saffron (Sam Saffron)

Having discussed this with Koichi I think he is wanting to merge this into core but the big blocker here is naming and some small details.

Koichi is not particularly fond of Thread.current << lambda {} cause he feels it is a bit confusing. Especially since we have Thread.current["x"].

I think this works (albeit with some multithreading concerns):

```
Thread.current.scheduler << lambda {}
Thread.current.scheduler.resume
Thread.current.scheduler.current
Thread.current.scheduler.current&.yield
```

> One question, is how will Thread#[]/#[]= be handled inside the lambda?

I think it should be simply treated as a Thread global so it is shared between the lambdas.

If you need lambda specific storage we could implement something else. Otherwise it complicates stuff.

Regarding:

> Simply making a per-thread reactor and making read/write calls non-blocking only solves about 10% of the problem IMHO.

I am not sure if I agree with this. This change will give us a single threaded reactor and allow us to continue using the exact same API we use elsewhere. It drops in to existing Ruby code much cleaner than introducing new APIs, File#read yields, PG::Connection#exec yields and so on. This is something we have wanted for your years. We basically get EventMachine without needing to adhere to the EventMachine API. It would be a great first step.

One big question I have though is how rb_thread_call_with_gvl and rb_thread_call_without_gvl will be handled, cause without magic handling there we don't get free PG / MiniRacer support and many others which is a huge shame.

#72 - 02/02/2018 06:23 AM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> Having discussed this with Koichi I think he is wanting to
> merge this into core but the big blocker here is naming and
> some small details.

I'm leaning towards Thread::Green, so existing users can do
s/Thread.new/Thread::Green.new/ in many cases.

But, it would be easier if somebody good at API design (matz)
chimed in :>

Meanwhile, I think get rid of floating point timeouts:
https://bugs.ruby-lang.org/issues/14431
Then it might be easier to work on Queue/Mutex/... support.

> > One question, is how will Thread#[]/#[]= be handled inside the lambda?
>
> I think it should be simply treated as a Thread global so it is shared between the lambdas.
>
> If you need lambda specific storage we could implement something else. Otherwise it complicates stuff.

That's probably too incompatible; I think the current Fiber#[]/#[]=
behavior is fine (Thread::Green implemented as subclass of Fiber)

> One big question I have though is how rb_thread_call_with_gvl
> and rb_thread_call_without_gvl will be handled, cause without
> magic handling there we don't get free PG / MiniRacer support
> and many others which is a huge shame.

I expect PG to be able to benefit from rb_wait_for_single_fd when
using sockets. I know mysql2 uses rb_wait_for_single_fd, at least.

rb_thread_call_* is meant for CPU (or FS/memory)-bound tasks,
and wouldn't MiniRacer be CPU-bound? Dunno much about it...

#73 - 02/03/2018 01:36 AM - sam.saffron (Sam Saffron)

> I'm leaning towards Thread::Green, so existing users can do
> s/Thread.new/Thread::Green.new/ in many cases.

Yes I think this works the problem though is that people will
expect this to work like green threads, meaning they also
should auto-yield regularly. You should be allowed to have
two green threads doing expensive computations. One tight loop
in a reactor now and you blow up everything (unlike normal threads)

This would mean you would have to pull in the 1.8 scheduler
or something. But then this stops being a proper reactor :(.

I guess this is the underlying reason you just wanted to call this auto
yielding fibers instead of threads to start with.

A question for Matz and Koichi is if they expect the scheduler from 1.8
to be brought back, if this is "safe" by default and "opt-in" for unsafe.

> I expect PG to be able to benefit from rb_wait_for_single_fd when

> using sockets. I know mysql2 uses rb_wait_for_single_fd, at least.

I am not sure about this, libpq abstracts all of this stuff away from you
this is why Sean G wrote a complete binary protocol implementation in rust,
to gain control. pg gem does not use rb_wait_for_single_fd it just releases gvl.

We have to make sure there is some sort of path forward with Postgres here
it is a huge issue.

MiniRacer is CPU bound its basically packaging libv8 into Ruby. I am on the fence here
On one hand it would be nice to auto yield so we feel reduced GVL pain and Ruby code
can run while v8 does it's thing. On the other hand the semantics of one thread at 100%
suddenly becomes 2 threads at 100% is not ideal. Hard to decide.

#74 - 02/03/2018 09:41 AM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> Issue #13618 has been updated by sam.saffron (Sam Saffron).
>
> > I'm leaning towards Thread::Green, so existing users can do
> > s/Thread.new/Thread::Green.new/ in many cases.
>
> Yes I think this works the problem though is that people will
> expect this to work like green threads, meaning they also
> should auto-yield regularly. You should be allowed to have
> two green threads doing expensive computations. One tight loop
> in a reactor now and you blow up everything (unlike normal threads)

Good point. rb_thread_call_without_gvl could be used to migrate
work to a thread pool (so we end up with M:N threads),
and maybe that's not horrible as a default behavior.
Data migration across native threads would hurt locality-wise
for short-lived tasks (e.g. rb_stat), though...

Fwiw, I was planning on adding a hinting mechanism to
rb_thread_call_without_gvl anyways later on (for GC, maybe);
but hints could be added to prevent/encourage migration based
on the expected duration/bottleneck of the function.

> This would mean you would have to pull in the 1.8 scheduler
> or something. But then this stops being a proper reactor :(.
>
> I guess this is the underlying reason you just wanted to call this auto
> yielding fibers instead of threads to start with.

Right, the predictability of not having a timer switch threads
automatically is appealing, sometimes.

Having rb_thread_call_without_gvl become a scheduling point of
some sort for green threads would be fine, however, since all
callers already assume a context switch will happen.

> A question for Matz and Koichi is if they expect the scheduler from 1.8
> to be brought back, if this is "safe" by default and "opt-in" for unsafe.
>
> > I expect PG to be able to benefit from rb_wait_for_single_fd when
> > using sockets. I know mysql2 uses rb_wait_for_single_fd, at least.
>
> I am not sure about this, libpq abstracts all of this stuff away from you
> this is why Sean G wrote a complete binary protocol implementation in rust,
> to gain control. pg gem does not use rb_wait_for_single_fd it just releases gvl.
>
> We have to make sure there is some sort of path forward with Postgres here
> it is a huge issue.

I don't know how expensive it is to parse the Pg protocol;
but I remember in the 1.8 days pg was one of the few gems to
use rb_thread_select and it played nicely with 1.8 green threads.

Can't say I know Pg well these days, it's been over a decade since I used it with Ruby.

> MiniRacer is CPU bound its basically packaging libv8 into Ruby. I am on the fence here On one hand it would be nice to auto yield so we feel reduced GVL pain and Ruby code can run while v8 does it's thing. On the other hand the semantics of one thread at 100% suddenly becomes 2 threads at 100% is not ideal. Hard to decide.

How long does it release the GVL for? The thread pool / workqueue idea I mentioned above might be a good fit for this if the communications overhead can masked by the length of the task. Nothing wrong with 2 threads at 100% if they're getting work done faster than 1 thread at 100%.

I wouldn't want a native pool to be used for something like getaddrinfo, however, that's hugely inefficient (but exactly what getaddrinfo_a does internally in glibc).

#75 - 02/04/2018 06:14 AM - jjyr (Jinyang Jiang)

Excited to see this awesome feature! I'm implemented fiber-auto-schedule at ruby userland(light) few month ago(using monkey patch). Due to ruby complexity IO API (like: getc, getbyte, put,c, putbyte), it's hard to implement these methods without C, the built-in Threadlet or Thread::Green is all I want as a ruby user. (bad news for me is my library have no meaning to exists).

Two opinions:

- The name Threadlet or Thread::Green both is easy to understand and to guess it behaviour, so as a application level user I think both is fine.
- I think Mutex, ConditonVariable needed to be Thread::Green aware, cause if I write a thread-safe library using mutex, it's not make sense if it can't work under Thread::Green.

#76 - 02/05/2018 09:51 PM - normalperson (Eric Wong)

jjyruby@gmail.com wrote:

> Excited to see this awesome feature! I'm implemented fiber-auto-schedule at ruby userland(light) few month ago(using monkey patch). Due to ruby complexity IO API (like: getc, getbyte, put,c, putbyte), it's hard to implement these methods without C, the built-in Threadlet or Thread::Green is all I want as a ruby user. (bad news for me is my library have no meaning to exists).

Thank you for your response.

I agree a lot of the current IO stuff is difficult or costly to implement outside of C. I hope some dependencies on C can eventually be reduced; but stuff like supporting writev in IO#write_nonblock https://bugs.ruby-lang.org/issues/14404 remind me some things are perhaps best done in C.

Anyways lightio can be counted as another reason to implement this feature natively in core (along with previous efforts dating back to Neverblock), so perhaps lightio already served a great purpose :)

> Two opinions:
>
> The name Threadlet or Thread::Green both is easy to understand and to guess it behaviour, so as a application level user I think both is fine. I think the Mutex, ConditonVariable needed to be Thread::Green aware, cause if I write a thread-safe library using mutex, it's not make sense if it can't work under Thread::Green.

Yes, I am strongly leaning towards making mutex, cv and queues green-thread aware and I'm working on improving time representations in core to that end:
https://bugs.ruby-lang.org/issues/14431
https://bugs.ruby-lang.org/issues/14452

#77 - 02/08/2018 12:25 AM - sam.saffron (Sam Saffron)

> How long does it release the GVL for?

I would see it heavily depends on workload, but usually for our loads it is milliseconds for v8 work, in PGs case shortest duration is probably 0.5ms with a median more around 4-5ms

I would like to expand on the auto scheduler question here with a code example:

```
t1 = Thread::Green.new do
   while true
   end
end

t2 = Thread::Green.new do
   puts "hi"
end

t1.stop
```

I think the general expectation here is for this to output "hi" just like standard threads do.

I think we should probably support a ninja mode

Thread::Green.automatic_scheduling = false

Or something like that if we just want the fiber auto yield and nothing else, but the default should be safe.

Clearly safety is going to have to be somewhat limited until Fibers can move between threads cause you can be lost in C land.

Wondering what Matz and Koichi are thinking here?

Totally support mutex, cv and queue being green thread aware. Also would like to see that native timer which is green thread aware.

#78 - 02/13/2018 10:41 PM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> > How long does it release the GVL for?
>
> I would see it heavily depends on workload, but usually for our loads it is milliseconds for v8 work, in PGs case shortest duration is probably 0.5ms with a median more around 4-5ms

It looks like currently pg is in the same boat as filesystem access (which sucks): the GVL release overhead for file.c and dir.c operations in 2.5 is painful to stomach on fast SSDs; but they make dealing with HDDs, network FSes and USB/MMC devices tolerable...

But yeah, synchronously waiting on read/write from the Pg sockets is a total waste of native thread resources.

(to that end, I still want to get rid of the GVL because it slows down those operations in single-threaded mode)

> I would like to expand on the auto scheduler question here with a code example:
>
> ```
> t1 = Thread::Green.new do
>    while true
>    end
> end
>
> t2 = Thread::Green.new do
>    puts "hi"
> end
>
> t1.stop
> ```
>
> I think the general expectation here is for this to output "hi" just like standard threads do.

Earlier messages from ko1 indicated he favors fewer
opportunities where scheduling happens:

http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/81495
http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/81507

I definitely do not like switching at unpredictable points;
I would only want to switch when the current execution context
cannot proceed immediately.

> I think we should probably support a ninja mode
>
> Thread::Green.automatic_scheduling = false

Global switches like that probably lead to unpredictable
code across libraries. Maybe per-thread or per-block
options would be better; but even then libraries might
get confused or thrown off by it. However, if existing code
all assumes timeslice-based scheduling; maybe
per-block/per-thread isn't so bad.

> Or something like that if we just want the fiber auto yield
> and nothing else, but the default should be safe.

"safe" is a relative term :) Working on
https://bugs.ruby-lang.org/issues/14357 was yet another reminder
of why I don't like switching execution contexts at unpredictable
points.

> Clearly safety is going to have to be somewhat limited until
> Fibers can move between threads cause you can be lost in C
> land.

Not sure what you mean by that.

> Wondering what Matz and Koichi are thinking here?

ko1 has given some hints on this thread; and I remember reading
a developer's meeting summary where matz didn't want people to
massively rewrite their code to take advantage of this new
feature.

> Totally support mutex, cv and queue being green thread aware.
> Also would like to see that native timer which is green thread
> aware.

1. Thread::Green will have timeslice scheduling, 100% compatible
   API-wise with built-in mutex/cv/queue/etc and pure-Ruby code

2. pipe and sockets become O_NONBLOCK by default (as in 1.8)
   when created inside green threads.

3. rb_thread_blocking_region - uses a native thread pool transparently
   inside green threads.

This pool can auto-grow/shrink but the bound is the total
number of green threads in the system. It's safe to use a
big upper bound for existing applications since they already
expect heavyweight native threads from 1.9+.
We will fix+reuse USE_THREAD_CACHE hidden in current source
to manage this thread pool.

1. introduce Thread options to give users ability to:
2. force rb_thread_blocking_region to run in the current native thread
3. disable timeslice-based switching

Disabling timeslice-based scheduling should become an option
with native threads, too.

While writing this email, I considered making Thread
green-by-default while doing the items 2-4 above; but
C extensions relying on pthread_{get,set}specific would
be broken by the transparent thread pool.

#79 - 02/15/2018 03:22 AM - ioquatix (Samuel Williams)

How does Process.wait behave in Thread::Green?

#80 - 02/15/2018 04:11 AM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> How does Process.wait behave in Thread::Green?

Process.wait* methods use rb_waitpid internally, so it's
always been a scheduling point which lets other
auto-fibers/green-threads/whatever-we-call-them-this-week run.

#81 - 02/15/2018 01:13 PM - ioquatix (Samuel Williams)

The PG gem which uses libpq provides both synchronous and asynchronous APIs, and it is up to the client code to select one or the other. You can already use poll/select with the PG gem, that is not the issue here. Making it transparently async is simply not possible if a client uses the sync APIs.

#82 - 02/20/2018 06:42 AM - matz (Yukihiro Matsumoto)

May I add two new candidates (IoThread and Thread::Coop)?

Matz.

#83 - 02/20/2018 09:12 AM - normalperson (Eric Wong)

matz@ruby-lang.org wrote:

> May I add two new candidates (IoThread and Thread::Coop)?

Thanks for the ideas. I like Thread::Coop; maybe "Thread::Coro"
is good, too...

I don't think IoThread is good if we support Queue/Mutex/...

Also, what do you think about having a way to pass options to
thread creation like pthread_attr_t?

```
    attr = Thread::Attr.new

    # similar to pthread_attr_setscope.  I'm used to NPTL and
    # LinuxThreads only having PTHREAD_SCOPE_SYSTEM and ignored
    # of pthread_attr_setscope until I saw it again with mjit code:
    attr.type = :coop # (default is :native)

    # for advanced users only :)
    attr.vm_stack_size = ...
    attr.machine_stack_size = ...

    # This API gives room for future expansion

    Thread::Attr#create becomes a substitute for Thread.new
    attr.create(*args) # => Thread or Thread::Coop
```

And maybe we unify Thread::Coop and existing "Thread" as
single public "Thread" class to Ruby API; similar to what
we do with "Integer" (vs Bignum/Fixnum)

#84 - 02/21/2018 02:12 AM - ko1 (Koichi Sasada)

On 2018/02/20 18:06, Eric Wong wrote:

> Also, what do you think about having a way to pass options to
> thread creation like pthread_attr_t?

I like this idea because we don't need to invent new abstraction name.

Or Thread.create(keyword_args...) ? (.create is an example name).

Maybe Eric will be afraid keyword args cost :)

--
// SASADA Koichi at atdot dot net

#85 - 02/21/2018 08:12 AM - normalperson (Eric Wong)

Koichi Sasada ko1@atdot.net wrote:

> On 2018/02/20 18:06, Eric Wong wrote:
>
> > Also, what do you think about having a way to pass options to
> > thread creation like pthread_attr_t?
>
> I like this idea because we don't need to invent new abstraction name.

Me too.

> Or Thread.create(keyword_args...) ? (.create is an example name).

One problem with a singleton method in "Thread" is separating args
intended for Thread creation vs args for block. We have the
same problem for Thread.new, too.

Maybe Thread::Attr can be used like Struct:

```
Thread::Attr.new(type: :coop, ...).new(_a) do |a|
...
end
```

```
class MyActor < Thread::Attr.new(type: :coop)
...
end
MyActor.new(_a) { |a| ... }
```

> Maybe Eric will be afraid keyword args cost :)

Yes, garbage from kwargs still annoys me; but that's a separate
problem which I hope can be fixed sooner.

#86 - 02/21/2018 08:42 AM - ko1 (Koichi Sasada)

On 2018/02/21 17:07, Eric Wong wrote:

> > Or Thread.create(keyword_args...) ? (.create is an example name).
>
> One problem with a singleton method in "Thread" is separating args
> intended for Thread creation vs args for block. We have the
> same problem for Thread.new, too.
>
> Maybe Thread::Attr can be used like Struct:
>
> ```
> Thread::Attr.new(type: :coop, ...).new(_a) do |a|
>   ...
> end
> ```
>
> ```
> class MyActor < Thread::Attr.new(type: :coop)
>   ...
> end
> MyActor.new(_a) { |a| ... }
> ```

We have discussed similar ideas.

https://bugs.ruby-lang.org/issues/3187#note-8
https://bugs.ruby-lang.org/issues/6694

(and maybe we can find other related tickets)

Attr is new naming idea and we need to discuss about it.

> Yes, garbage from kwargs still annoys me; but that's a separate
> problem which I hope can be fixed sooner.

Just now I'm working on it (to present at OkinawaRubyKaigi02, Japan regional Ruby conference. This is Event-Driven-Development).

--
// SASADA Koichi at atdot dot net

#87 - 02/21/2018 02:55 PM - jjyr (Jinyang Jiang)

Pass a new arg to create GreenThread maybe introduce more problems.
For example:

```
g_thr = Thread.create(green_thread: true)
g_thr.is_a?(Thread) # => false or ture?
```

If Thread.create return a non Thread object, it's weird.
But if g_thr is a Thread object, we need another API to tell whether thread is native or not

#88 - 02/28/2018 07:25 PM - baweaver (Brandon Weaver)

I would argue strongly for keeping simpler one-word naming conventions where possible. Ruby is built upon a simple elegance, and having to type out things such as Thread::Attr.new(type: :coop) feels very unintuitive.

It also stands that coop would have interesting english implications as it potentially relates to chicken coops and would not be read as co-op.

Highlighting Shan's earlier point, we should consider established names in other languages:

```
Goroutine (Go)
Lane (Lua)
Spark on a Haskell Thread (Haskell)
Task (Elixir - though there's more is going on here, and Process is really closer)
Process (Erlang)
Fiber (Crystal)
```

Simple is best when naming things.

#89 - 03/13/2018 02:57 AM - ioquatix (Samuel Williams)

I'm still not clear why a new name needs to be introduced. Fiber should be sufficient IMHO, if you want to enable/disable auto yield on blocking operations, perhaps make it an option to Fiber, e.g. Fiber.new(non_blocking: true). That way, it would be a minimally invasive drop-in to existing code, and it would be easy to switch between blocking and non-blocking behaviour by client code. That, and you wouldn't need to invent a name which relates to chicken farming :p

It would be also be pretty awesome if you could actually supply a reactor to use, e.g. Fiber.new(io_reactor: my_reactor). In this case, blocking operations would call my_reactor.wait_readable(io) and my_reactor.wait_writable(io). Something like this allows for the IO policy to be more flexible than global per-process reactor or other similar implementation. I'm still personally against a global IO reactor as it's an unnecessary point of thread contention and complexity (i.e. how does it work with fork? does every IO operation require locking a mutex?)

So, as mentioned earlier, libpq and the associated pq gems won't suddenly become asynchronous because of this patch (seems like there is a bit of a misunderstanding how this works under the hood). In fact, we can already achieve massive concurrency improvements using async, and I've tested this using puma and falcon. The difference was pretty big! I wrote up a summary here: https://github.com/socketry/async-postgres#performance - feel free to come and chat in https://gitter.im/socketry/async as there are a quite a few people there who are interested in the direction of asynchronous IO in Ruby.

So, again, I think this patch is simply does too much. It should be split into 1/ a standard IO reactor for Ruby with a standard interface that others can implement (could easily be a gem) and 2/ Expand Fiber to support non-blocking operations by way of a supplied IO reactor (very minimal surface area/names required). Which, for the most part, is close how https://github.com/socketry/async works and if you take this approach async could build on top of it. I don't think it's a stupid idea to allow things like EventMachine, async, and other IO reactors to work together.

Just FYI, I'm not sure what visibility you have on other projects, but there are at least two I know of implementing similar concepts:

https://github.com/chuckremes/ruby-io

https://github.com/socketry/lightio

Even as the author of async, I don't feel it's a one size fits all solution and I don't even want async to become a standard solution. I think it's great we have options like the above, and I think if we design the Fiber API correctly, it should absolutely be possible to a/ work across different implementations of Ruby efficiently and b/ compose existing and new IO reactors/models without hurting backwards compatibility.

I think that if people want to implement their own IO scheduling policies, on a per-fiber basis, that would be pretty awesome.

#90 - 04/21/2018 11:33 AM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> It would be also be pretty awesome if you could actually supply a reactor to use, e.g. Fiber.new(io_reactor: my_reactor). In this case, blocking operations would call my_reactor.wait_readable(io) and my_reactor.wait_writable(io). Something like this allows for the IO policy to be more flexible than global per-process reactor or other similar implementation. I'm still personally against a global IO reactor as it's an unnecessary point of thread contention and complexity (i.e. how does it work with fork? does every IO operation require locking a mutex?)

Global epoll FD is not anywhere close to being a point of thread
contention in real-world usage, especially with current GVL.

If epoll contention ever comes close to being a problem, I'll
fix it in the Linux kernel. I worked on reducing that
contention in the kernel a few years back but couldn't measure
an improvement outside of synthetic benchmarks with the workload
I had at the time (which wasn't ruby and had no GVL).

Of course unbuffered IO operations will not require mutexes and
fork is taken into consideration. For example, it accounts for
native kqueue having unique close-on-fork behavior which no
other FD type has.

Btw, your use the word "reactor" is a bit lost on me.
My view of epoll (and kqueue) is the sum of two data structures:

a. map structure (rbtree, hash, etc...)

b. readiness queue (why else does "kqueue" have the word "queue" in it)?

And what happens is:

1. green thread gets stuck on IO
2. native thread epoll_ctl/kevent(changelist) to places items into a.
3. kernel puts items in a. into b. when they are ready
4. threads take items off b. via epoll_wait/kevent(eventlist)

I don't think "reactor" describes that, because reactor pattern
is rooted in a single thread mentality. epoll/kqueue invite
parallelism.

Now, I'm hungry, maybe it's the "restaurant kitchen pattern",
and the analogy would be:

1. patrons order food from a waiter
2. waiters puts in orders into the kitchen
3. cooks work on orders, prepared plates are placed on the counter
4. waiters takes plates from counter and serves to patrons

There can be any number of waiters and cooks working, and in no
way are their quantities tied together.

Each waiter can handle multiple patrons, but sequentially.
Waiters may also put in orders for themselves.

Some plates take a long time to prepare, some plates are quick;
some plates are ready immediately.
Many orders may come in at once, or they can trickle in.

Many plates can be ready at once, or they can trickle out.

cooks = native threads inside the kernel
waiters = native threads seen by userspace
patrons = green threads

As with green threads and the kernel, cooks never see
the patrons and don't know how many there are.

Waiters don't care or know which cook prepares the plate they
serve. Cooks don't care which waiter the plate goes to, either.

> So, as mentioned earlier, libpq and the associated pq gems won't suddenly become asynchronous because of this patch (seems like there is a bit of a misunderstanding how this works under the hood). In fact, we can already achieve massive concurrency improvements using async, and I've tested this using puma and falcon. The difference was pretty big! I wrote up a summary here: https://github.com/socketry/async-postgres#performance

Of course things don't become asynchronous automatically.
Again, I've seen this all before with Revactor, NeverBlock, etc.
Problem is, if it's not in built-in, it'll likely end up
unused or causing more ecosystem fragmentation. Lets not forget
many languages ruby has lost users to (Go, Erlang) has similar
lightweight threading primitives built-in.

And again, I consider this work to be fixing a regression when
we made the 1.8 -> 1.9 transition to native Thread.

> feel free to come and chat in https://gitter.im/socketry/async as there are a quite a few people there who are interested in the direction of asynchronous IO in Ruby.

Sorry, it's not reasonable to expect Free Software developers to
rely on proprietary messaging platform like GitHub (which gitter
depends on).

> So, again, I think this patch is simply does too much. It should be split into 1/ a standard IO reactor for Ruby with a standard interface that others can implement (could easily be a gem)

Maybe exposing some parts of the mapping + queue API might
be possible. Again, I don't think "reactor" is even the right
pattern or word to describe what's going on, here.

#91 - 04/26/2018 04:57 AM - ioquatix (Samuel Williams)

If you are unsure of a good definition for the reactor pattern, I think this is a good one: https://en.wikipedia.org/wiki/Reactor_pattern except the assumption that you need to invert flow control which is not necessary using fibers.

In my experience, experimenting with implementations that use shared epoll/kqueue on a background thread, the thread contention is a pretty big overhead, I think somewhere between 5x and 10x overhead but I'd prefer to back that up with real numbers. Not only that, the practical implementation is more complicated since you need to implement IPC, locking etc. It's simpler to have a per-thread reactor, and the practical realities of Ruby as it stands with the GVL mean that using more than one thread is a terrible idea if you actually care about throughput/performance.

I'm writing up a summary of this but it's not finished yet, it will include actual numbers to back up the above assertions, once I'm done I will post it here.

#92 - 04/26/2018 06:04 AM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> If you are unsure of a good definition for the reactor pattern, I think this is a good one: https://en.wikipedia.org/wiki/Reactor_pattern except the assumption that you need to invert flow control which is not necessary using fibers.

Right, I was reading the Wikipedia page and that description does not resemble the implementation I have for this feature.

> In my experience, experimenting with implementations that use shared epoll/kqueue on a background thread

Using a background thread is your mistake. Multiple foreground threads safely use epoll_wait or kevent on the SAME epoll or kqueue fd. It's perfectly safe to do that.

Typical reactor is not designed to handle that :P

If we eventually encounter contention, we can add more epoll or kqueue descriptors; but I doubt it'll ever come to that.

Back to the diner analogy: multiple restaurant waiter can sit at the counter to wait if the cooks are slow and there's no diners placing new orders.

> , the thread contention is a pretty big overhead, I think somewhere between 5x and 10x overhead but I'd prefer to back that up with real numbers. Not only that, the practical implementation is more complicated since you need to implement IPC, locking etc.

IPC? Interprocess communication? What? There's no processes, here. No extra locking, either. The kernel already does locking, no point in doing it in luserspace.

#93 - 04/30/2018 01:24 AM - ioquatix (Samuel Williams)

> Using a background thread is your mistake.

Don't assume I made this design. It was made by other people. I merely tested it because I was interested in the performance overhead. And yes, there is significant overhead. And let's be generous, people who invested their time and effort to make such a thing for Ruby deserve our appreciation. Knowing that the path they chose to explore was not good is equally important.

> Multiple foreground threads safely use epoll_wait or kevent on the SAME epoll or kqueue fd. It's perfectly safe to do that.

Sure, that's reasonable. If you want to share those data structures across threads, you can dispatch your work in different threads too. I liked what you did with https://yhbt.net/yahns/yahns.txt and it's an interesting design.

The biggest single benefit of this design is that blocking operations in an individual "task" or "worker" won't block any other "task" or "worker", up to the limit of the thread pool you allocate, at which point things WILL start causing blocking. So you can't avoid blocking even with this design.

The major downside of such a design is that workers have to assume they could be running on different threads, so shared data structure needs locking/will cause contention. In addition the current state of the Ruby GIL means that any such design will generally have poor performance.

Here is almost identical code path running, one with 8 forked processes, and one with 8 threads, running on Ruby 2.5:

```
> falcon serve --threaded
> wrk -t8 -c8 -d10 http://localhost:9292
Running 10s test @ http://localhost:9292
  8 threads and 8 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    54.67ms   25.39ms 189.02ms   72.29%
    Req/Sec    18.50      7.18    40.00     53.38%
  1483 requests in 10.04s, 174.88MB read
Requests/sec:    147.74
Transfer/sec:     17.42MB

> falcon serve --forked
> wrk -t8 -c8 -d10 http://localhost:9292
Running 10s test @ http://localhost:9292
  8 threads and 8 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    29.77ms   66.90ms 571.70ms   93.71%
    Req/Sec    71.50     19.54   128.00     83.42%
  5442 requests in 10.10s, 641.61MB read
```

```
Requests/sec:    538.90
Transfer/sec:     63.54MB
```

This test is actually on a fresh Rails website (Rails performance isn't great to begin with), on macOS which has pretty bad IO performance. Running the same thing on Linux gives:

```
% falcon serve --threaded
% wrk -t8 -c8 -d10 http://localhost:9292
Running 10s test @ http://localhost:9292
  8 threads and 8 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    26.41ms   13.74ms 123.01ms   69.85%
    Req/Sec    38.53     11.26    80.00     63.38%
  3082 requests in 10.01s, 363.36MB read
Requests/sec:    307.99
Transfer/sec:     36.31MB

% falcon serve --forked
% wrk -t8 -c8 -d10 http://localhost:9292
Running 10s test @ http://localhost:9292
  8 threads and 8 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     9.78ms   24.91ms 309.70ms   97.59%
    Req/Sec   168.68     49.75   262.00     63.89%
  13203 requests in 10.02s, 1.52GB read
Requests/sec:   1318.05
Transfer/sec:    155.39MB
```

So, I think it's safe to say, that in an end to end test, the GIL is a MAJOR performance issue. Feel free to correct me if you think I'm wrong. I'm sure this story is more complicated than the above benchmarks, but I felt like it was a useful comparison.

Therefore, right now, for highly concurrent IO with Ruby, what you actually want is the following:

- One process per CPU core.
- One IO thread per process.
- Multiple fibers, one per worker.

Blocking operations that are causing performance issues should use a thread pool. For things like launching an external process or syscall, and waiting for it to finish, threads are ideal.

The major benefit of such a design is that individual fibers all run on the same thread. You ultimately have similar issues w.r.t. blocking as yahns. However, because all workers run concurrently on the same thread, you don't have any locking/concurrency/mutability issues. To me, this is a massive benefit as it makes writing code with this model super easy.

> Typical reactor is not designed to handle that :P

Yes, but it's by design, not by accident. If you need to scale up, just fork more reactors. On the linux desktop above, async-http can handle 100,000+ requests per second using 8 cores for trivial benchmarks. So, performance is something which can scale. The next question then, is design.

There is some elegance in the design you propose. Your proposal requires some kind of "Task" or "Worker" which is a fiber which will yield when IO would block, and resume when IO is ready. Based on what you've said, do you mind explaining whether the "Task" or "Worker" is resumed on the same thread or a different one? Do you maintain a thread pool?

If it's always resumed on the same thread, how do you manage that? e.g. perhaps you can show me how the following would work:

```
Thread.new do
    Worker.new do
        # .. blocking IO
    end

    Worker.new do
        # .. blocking IO
    end

    # implicitly waits for all workers to complete?
end
```

If you following this model, the thread must be calling into epoll or kqueue in order to resume work. But based on what you've said, if you have several of the above threads running, and the thread itself is invoking epoll_wait, then it receives events for a different thread, how does that work? Do you send the events to the different thread? If you do that, what is the overhead? If you don't do that, do you move workers between threads?

Then, why not consider the similar model to async which uses per-thread reactors. The workers do not move around threads, and the reactor does not need to send events to other threads.

Thanks for your continued time and patience discussing these interesting issues.

#94 - 04/30/2018 01:37 AM - ioquatix (Samuel Williams)

> IPC? Interprocess communication? What? There's no processes, here.

Sorry, my terminology wasn't so clear. However, what I meant in this case is IPC between user process and kernel. i.e. any syscall.

e.g. locking on a mutex - as long as there is no contention, there is no system call, otherwise calling thread will sleep (IPC).

If you use any kind of thread-safe shared mutable state (e.g. a queue), you will invoke some kind of IPC (syscall) overhead.

If you don't need to communicate between threads, you can avoid all IPC, e.g. https://github.com/socketry/async/blob/master/lib/async/queue.rb doesn't directly invoke any kind of IPC/syscall in order to function. It's preferable because you can make stronger guarantees about the order of operations and performance.

#95 - 04/30/2018 10:32 AM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> > Using a background thread is your mistake.
>
> Don't assume I made this design. It was made by other people. I merely tested it because I was interested in the performance overhead. And yes, there is significant overhead. And let's be generous, people who invested their time and effort to make such a thing for Ruby deserve our appreciation. Knowing that the path they chose to explore was not good is equally important.

The problem I have with existing reactor patterns is threads are an afterthought. They should not be.

> > Multiple foreground threads safely use epoll_wait or kevent on the SAME epoll or kqueue fd. It's perfectly safe to do that.
>
> Sure, that's reasonable. If you want to share those data structures across threads, you can dispatch your work in different threads too. I liked what you did with https://yhbt.net/yahns/yahns.txt and it's an interesting design.
>
> The biggest single benefit of this design is that blocking operations in an individual "task" or "worker" won't block any other "task" or "worker", up to the limit of the thread pool you allocate, at which point things WILL start causing blocking. So you can't avoid blocking even with this design.

Of course everything blocks at some point when things get overloaded. The difference is there's no head-of-line blocking in yahns because sockets can migrate to an idle thread.

Auto-fiber can't avoid head-of-line blocking right now, because Ruby Fiber can't migrate across threads (that's a separate problem).

> The major downside of such a design is that workers have to assume they could be running on different threads, so shared data structure needs locking/will cause contention. In addition the current state of the Ruby GIL means that any such design will generally have poor performance.

No, you don't need locking for read/write ops if you use EV_ONESHOT/EPOLLONESHOT. libev and typical reactor pattern designs are not built with one-shot in mind, so they're stuck using Level-triggering and rely on locking.

Only FD allocation/deallocation requires locking (the kernel needs locking, there, too).

> So, I think it's safe to say, that in an end to end test, the GIL is a MAJOR performance issue. Feel free to correct me if you think I'm wrong. I'm sure this story is more complicated than the above benchmarks, but I felt like it was a useful comparison.

GVL is a major performance issue if your bottleneck is the CPU. It is not a major problem when my bottleneck is network I/O or high-latency disks (I have systems with dozens or hundreds).

> Blocking operations that are causing performance issues should use a thread pool. For things like launching an external process or syscall, and waiting for it to finish, threads are ideal.

Launching external process and waitpid does not benefit from native threads.

Again, native_thread_count >= disk_count is a huge thing I rely on with Ruby for years now, so using one native thread is totally wrong for my use case when I have dozens/hundreds of slow disks.

> There is some elegance in the design you propose. Your proposal requires some kind of "Task" or "Worker" which is a fiber which will yield when IO would block, and resume when IO is ready. Based on what you've said, do you mind explaining whether the "Task" or "Worker" is resumed on the same thread or a different one? Do you maintain a thread pool?

The use of threads or thread pool remains up to the Ruby user. There's no extra fibers or native threads created behind users' back; that's a waste of memory. It uses "idle time" of any available threads (including main thread) to do scheduling work.

(Current Ruby has provisions for an internal thread cache for Thread.new, but it's orthogonal to this issue and has been around for a decade in a buggy, never-enabled state).

> If it's always resumed on the same thread, how do you manage that? e.g. perhaps you can show me how the following would work:

Every thread has a FIFO run-queue (th->afrunq or th->runq depending on which version you look at)....

> If you following this model, the thread must be calling into epoll or kqueue in order to resume work. But based on what you've said, if you have several of the above threads running, and the thread itself is invoking epoll_wait, then it receives events for a different thread, how does that work? Do you send the events to the different thread? If you do that, what is the overhead? If you don't do that, do you move workers between threads?

When a thread receives work for a fiber for a different thread, it inserts into the runqueue of the other thread.

Right now it's ccan/list for branchless insert/delete (relies on GVL)

If/when we get rid of GVL, we will likely use wfcqueue for wait-free insert and mass dequeue. Wait-free is better than lock-free, even, but there'd still be memory barriers, of course.

Again, we can't move fibers across threads in Ruby atm.

One-shot notifications ensures we don't get unintended events.

> Then, why not consider the similar model to async which uses per-thread reactors. The workers do not move around threads, and the reactor does not need to send events to other threads.

I know all that sounds like an unnecessary serialization and overhead, but the same stuff is being serialized in the kernel and hardware, even.

For (typical) servers with a single active NIC, interrupts tend to be handled by a single CPU and inserting into epoll readylist has the same serialization overhead. So partitioning across multiple epoll/kqueue descriptions inside the kernel is a waste of time unless you're getting enough traffic to max out a CPU with interrupt handling.

There's nothing about the design which prevents the use of parallel schedulers (they are not "reactors" to me).

So if I was getting enough network traffic to saturate multiple NICs and peg a CPU from network traffic alone, yes, as a last resort I'd have extra epoll/kqueue-based schedulers inside a process.

That's a last resort. I know we can eek more performance out of the epoll readylist inside the Linux kernel, first. But that's not even worth the effort atm.

Until then, I"d rather save unswappable kernel memory and FDs with a single epoll/kqueue per-process.

#96 - 04/30/2018 10:52 AM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> If you use any kind of thread-safe shared mutable state (e.g. a queue), you will invoke some kind of IPC (syscall) overhead.

There's unavoidable locking costs we're always paying inside the kernel for every epoll_ctl/epoll_wait/kevent syscall.

> If you don't need to communicate between threads, you can avoid all IPC, e.g. https://github.com/socketry/async/blob/master/lib/async/queue.rb doesn't directly invoke any kind of IPC/syscall in order to function.

Since we're paying the locking cost inside the kernel, we won't need to pay that cost in userspace (with one-shot). So we might as well take advantage of the fact we're getting "free" thread-safety from the kernel...

#97 - 05/02/2018 05:20 AM - ioquatix (Samuel Williams)

Thanks for the detailed information.

So, it seems like your design has unavoidable contention (and therefore latency) because you need to send events between threads, which is what I expected. However, you argue this overhead should be small. I'd like to see actual numbers TBH.

And as you state, it's not possible (nor desirable IMHO) to move fibers between threads. Yes, head-of-line blocking might be an issue. Moving stacks between CPU cores is not without it's own set of overheads. If you have serious issues with head-of-line blocking it's more likely to be a problem with your code (I've directly experienced this and the result was: https://github.com/socketry/async-http/blob/ca655aa190ed7a89b601e267906359793271ec8a/lib/async/http/protocol/http11.rb#L93).

It would be interesting to see exactly how much overhead is incurred using a shared epoll. I know from my testing that

I remember in my tests, the latency of yahns was a lot higher than async-http:

```
async-http

koyoko% wrk -c 16 -t 16 -d 10 http://localhost:9292/wiki/index
```

```
Running 10s test @ http://localhost:9292/wiki/index
  16 threads and 16 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     9.18ms    3.23ms  86.46ms   98.51%
    Req/Sec   109.47     17.70   121.00     95.49%
  8954 requests in 10.02s, 29.99MB read
  Socket errors: connect 0, read 0, write 0, timeout 4
Requests/sec:    893.68
Transfer/sec:      2.99MB
```

```
yahns
```

```
koyoko% wrk -c 16 -t 16 -d 10 http://localhost:9292/wiki/index
Running 10s test @ http://localhost:9292/wiki/index
  16 threads and 16 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    20.51ms   16.04ms 190.57ms   85.79%
    Req/Sec    54.43     32.59   191.00     65.56%
  8702 requests in 10.10s, 29.68MB read
Requests/sec:    861.61
Transfer/sec:      2.94MB
```

This was a long time ago, async-http performance has also improved and the issue regarding timeouts was resolved. When I have some time I can repeat these tests.

Tangentially related:

In my own IO scheduler/reactor, I choose to use EPOLLET and manually add/remove the handlers. The basic implementation looks something like this:

Implementation of Readable which is what manages adding events to epoll/kqueue:
https://github.com/kurocha/async/blob/master/source/Async/Readable.cpp

It's prepared but not added to the reactor here (it's lazy):
https://github.com/kurocha/async-http/blob/eff77f61f7a85a3ac21f7a8f51ba07f069063cbe/source/Async/HTTP/V1/Protocol.cpp#L34

By calling wait, the fd is inserted into the reactor/selector:
https://github.com/kurocha/async/blob/2edef4d6990259cc60cc307b6de2ab35b97560f1/source/Async/Protocol/Buffer.cpp#L254

The cost of adding/removing FDs is effectively constant time given an arbitrary number of reads or writes. We shouldn't preclude implementing this model in Ruby if it makes sense. As you say, the overhead of the system call is pretty minimal.

Now that you mention it, I'd like to compare EPOLLET vs EPOLLONESHOT. It's an interesting design choice and it makes a lot of sense if you are doing only one read in the context of a blocking operation.

#98 - 05/02/2018 08:04 AM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> So, it seems like your design has unavoidable contention (and therefore latency) because you need to send events between threads, which is what I expected. However, you argue this overhead should be small. I'd like to see actual numbers TBH.

Any contention is completely masked by GVL at the moment. When we get Guilds or get rid of GVL, of course I'll try per-core schedulers. As it stands, per-Thread schedulers will be a disaster for systems with hundreds of native Threads (because those threads are needed for servicing hundreds of slow disks)

> And as you state, it's not possible (nor desirable IMHO) to move fibers between threads. Yes, head-of-line blocking might be an issue. Moving stacks between CPU cores is not without it's own set of overheads. If you have serious issues with head-of-line blocking it's more likely to be a problem with your code (I've directly experienced this and the result was: https://github.com/socketry/async-http/blob/ca655aa190ed7a89b601e267906359793271ec8a/lib/async/http/protocol/http11.rb#L93).

Fwiw, yahns makes large performance sacrifices(*) to avoid HOL blocking.

According to Go user reports, being able to move goroutines between native threads is a big feature to them. But I don't

think it's possible with current Ruby C API, anyways :<

> It would be interesting to see exactly how much overhead is incurred using a shared epoll. I know from my testing that

It's high for yahns because the default max_events for epoll_wait is only 1. (*) Throughput should increase with something reasonable like 64, but you will lose HOL blocking resistance.

Auto-fiber starts at max_events 8 and auto-increases if needed. Fwiw, HOL blocking considerations for auto-fiber and yahns are completely different so I don't see a downside to increasing max_events with auto-fiber aside from memory use.

With yahns Transfer-Encoding:chunked responses, yahns prioritizes latency of each individual chunk over overall throughput, so it loses in throughput performance, too. Maybe there can be an option to change that *shrug*

> It's prepared but not added to the reactor here (it's lazy):
> https://github.com/kurocha/async-http/blob/eff77f61f7a85a3ac21f7a8f51ba07f069063cbe/source/Async/HTTP/V1/Protocol.cpp#L34
>
> By calling wait, the fd is inserted into the reactor/selector:
> https://github.com/kurocha/async/blob/2edef4d6990259cc60cc307b6de2ab35b97560f1/source/Async/Protocol/Buffer.cpp#L254
>
> The cost of adding/removing FDs is effectively constant time given an arbitrary number of reads or writes. We shouldn't preclude implementing this model in Ruby if it makes sense. As you say, the overhead of the system call is pretty minimal.

Right, for auto-fiber in Ruby is lazy add (and the scheduler is lazily created).

> Now that you mention it, I'd like to compare EPOLLET vs EPOLLONESHOT. It's an interesting design choice and it makes a lot of sense if you are doing only one read in the context of a blocking operation.

To me, they're drastically different in terms of programming style.

ET isn't too different than level-triggered (LT), but I'd say the trickiest of the bunch. I would say ET is trickier than LT to avoid head-of-line blocking because you need to keep track of undrained buffers (from clients which pipeline aggressively) and not lose them if you want to temporarily yield to other clients.

I suppose Edge and Level trigger invite a "reactive" design and inverted control flow.

The main thing which bothers me about both ET and LT is you have to remember to disable/reenable events (to avoid unfairness or DoS).

Under ideal conditions (clients not trying to DoS or be unfair to other clients), ET can probably be fastest. Just totally unrealistic to expect ideal conditions.

So I strongly prefer one-shot because you don't have to deal with disabling events. This is especially useful when there's aggressive pipelining going on (e.g. client sending you requests quickly, yet reading responses slowly to fill up your output buffers). one-shot makes it a queue, so that also invites sharing across threads.

The way auto-fiber uses one-shot doesn't invert the control flow at all. (Though you could, yahns does that). Instead, auto-fiber feels like the Linux kernel scheduler API:

```
    /* get stuck on EAGAIN or similar, can't proceed */
    add_wait_queue_and_register(&foo) /* list_add + epoll_ctl */
    while (!foo->ready) {
```

```
        /*
         * run epoll_wait and let other threads/fibers run
         * Since we registered foo with the scheduler, it can become
         * ready at any time while schedule() is running;
         */
        schedule();
    }
    remove_wait_queue(&foo) /* list_del via rb_ensure */
    /* resume whatever this fiber was doing */
```

#99 - 05/02/2018 08:38 AM - ioquatix (Samuel Williams)

I found an interesting summary of EPOLLET, which I think explains it better than I did: https://stackoverflow.com/a/46634185/29381 Basically, it minimise OS IPC.

> According to Go user reports, being able to move goroutines
> between native threads is a big feature to them. But I don't
> think it's possible with current Ruby C API, anyways :<

By definition Fibers shouldn't move between threads. If you can move the coroutine between threads, it's a green thread (user-scheduled thread). Writing code that use multiple fibers in a single thread provides strong guarantees about asynchronous execution. You can avoid things like mutex, condition variable, etc, and minimise (avoid) deadlocks and other problems of multiple threads. And as you say, GVL is a big problem so there is little reason to use it anyway.

> Fwiw, yahns makes large performance sacrifices(*) to avoid HOL
> blocking.

And yet it has 2x the latency of async-http. Can you tell me how to test it in more favourable configuration?

> The main thing which bothers me about both ET and LT is you have
> to remember to disable/reenable events (to avoid unfairness or DoS).

Fortunately C++ RAII takes care of this.

> Under ideal conditions (clients not trying to DoS or be unfair
> to other clients), ET can probably be fastest. Just totally
> unrealistic to expect ideal conditions.

We've used it in production systems and it's been great, serving millions of requests with no issues.

I can see that we can discuss these things for a long time, and while I find it really interesting, we do need to move forward for Ruby's sake.

I think the work you've done here is really great.

I just think it needs to be slightly more modular; but not in a way that detracts from becoming a ubiquitous solution for non-blocking IO.

It needs to be possible for concurrency library authors to process blocking operations with their own selector/reactor design.

I really think there would be value in being able to write something like:

```
selector = NIO::Selector.new # or EventMachine, etc

Fiber.new(selector: selector) do
  io.read # invokes selector.wait_readable(io) if EWOULDBLOCK

  # nested fibers can inherit parent selector.
end.resume

selector.run
```

Your selector implementation could fit into that, along with NIO4R, EventMachine, etc.

I would REALLY like to see something like this. So, we can explore different models of concurrency. Sometimes we would like to choose different selector implementation for pragmatic reasons: On macOS, kqueue doesn't work with tty devices. But select does work fine, with lower performance.

```
# If program needs to block on TTY:
selector = Thread::Selector.new(:select)
# Otherwise
selector = Thread::Selector.new(:kqueue)
```

Such a design let's you easily tune parameters (like size of event queue, other details of the implementation that can significantly affect performance).

In addition, I recently implemented a debug wrapper for NIO::Selector. It detects undesirable conditions but it's typically only enabled when running tests: https://github.com/socketry/async/blob/master/lib/async/debug/selector.rb

With such a design as proposed above, such a feature becomes trivial to implement.

We can have a sane defaults. I don't mind how the API works, just that I can supply my own selector/reactor on a per-fiber basis.

#100 - 05/02/2018 11:03 AM - normalperson (Eric Wong)

samuel@oriontransfer.org wrote:

> I found an interesting summary of EPOLLET, which I think explains it better than I did: https://stackoverflow.com/a/46634185/29381 Basically, it minimise OS IPC.

Minimize syscalls, you mean. I completely agree EPOLLET results in the fewest syscalls. But again, that falls down when you have aggressive clients which are pipelining requests and reading large responses slowly.

> > According to Go user reports, being able to move goroutines between native threads is a big feature to them. But I don't think it's possible with current Ruby C API, anyways :<
>
> By definition Fibers shouldn't move between threads. If you can move the coroutine between threads, it's a green thread (user-scheduled thread).

I don't care for those rigid definitions. They're all just bytes that's scheduled in userland and not the kernel. "Auto-fiber" and green thread are the same to me so this feature might become "green thread".

> deadlocks and other problems of multiple threads. And as you say, GVL is a big problem so there is little reason to use it anyway.

Again, native threads are still useful despite GVL.

> > Fwiw, yahns makes large performance sacrifices(*) to avoid HOL blocking.
>
> And yet it has 2x the latency of async-http. Can you tell me how to test it in more favourable configuration?

yahns is designed to deal with apps with both slow and fast endpoints simultaneously. Given N threads running, (N-1) may be stuck servicing slow endpoints, while the Nth one remains free to service ANY other client.

Again, having max_events>1 as I mentioned in my previous email might be worth a shot for benchmarking. But I would never use that for apps where different requests can have different response times.

> > The main thing which bothers me about both ET and LT is you have to remember to disable/reenable events (to avoid unfairness or DoS).
>
> Fortunately C++ RAII takes care of this.

I'm not familiar with C++, but it looks like you're using EPOLL_CTL_ADD/DEL, but no EPOLL_CTL_MOD. Using MOD to disable events instead of ADD/DEL will save you some allocations and possibly extra locking+checks inside Linux.

No need to use EPOLL_CTL_MOD to disable with oneshot, only rearm (this is what makes oneshot more expensive than ET in ideal conditions).

> I just think it needs to be slightly more modular; but not in a way that detracts from becoming a ubiquitous solution for non-blocking IO.
>
> It needs to be possible for concurrency library authors to process blocking operations with their own selector/reactor design.

Really, I think it's a waste of time and resources to support these things. As I described earlier, the one-shot scheduler design is far too different to be worth shoehorning into dealing with a reactor with inverted control flow.

I also don't want to make the Ruby API too big; we can barely come up with this API and semantics as-is...

> I would REALLY like to see something like this. So, we can explore different models of concurrency. Sometimes we would like to choose different selector implementation for pragmatic reasons: On macOS, kqueue doesn't work with tty devices. But select does work fine, with lower performance.

The correct thing to do in that case is to get somebody to fix macOS :)

Since that's likely impossible, we'll likely support more quirks within the kqueue implementation and be transparent to the user. There's already a one quirk for dealing with the lack of POLLPRI/exceptfds support in kevent and I always expected more...

Curious if you know this: if select works for ttys on macOS, does poll?

In Linux, select/poll/ppoll/epoll all share the same notification internals (->poll callback); but from cursory reading of FreeBSD source; the kern_events stuff is separate and huge compared to epoll.

> In addition, such a design let's you easily tune parameters (like size of event queue, other details of the implementation that can significantly affect performance).

There's no need to tune anything. maxevents retrieved from epoll_wait/kevent is the only parameter and that grows as needed. Everything else (including maximum queue size) is tied to number of fibers/FDs which is already controlled by the application code.

#101 - 05/02/2018 11:36 PM - ioquatix (Samuel Williams)

> I don't care for those rigid definitions. They're all just bytes that's scheduled in userland and not the kernel. "Auto-fiber" and green thread are the same to me so this feature might become "green thread".

Personally, I find the definitions very useful, and there are good rigid definitions for those terms.

> yahns is designed to deal with apps with both slow and fast endpoints simultaneously. Given N threads running, (N-1) may be stuck servicing slow endpoints, while the Nth one remains free to service ANY other client.

I'm surprised by this, how would slow clients be a problem? Don't you just call nonblocking write? async-http handles these case just fine by streaming the response body in 16KB chunks. If it can't write the chunk, it yields back to the reactor. What do you mean by "slow clients"? If an IO is "slow", it just waits in the reactor until something can be done.

> Curious if you know this: if select works for ttys on macOS, does poll?

My understanding was that poll was implemented on top of kqueue in macOS. But I guess no one actually knows. My main reasoning behind this was

poll was added later but exhibited some of the same bugs as kqueue. You can work around TTY problems using something like http://code.saghul.net/index.php/2016/05/24/libuv-internals-the-osx-select2-trick/

> Really, I think it's a waste of time and resources to support
> these things.

If you think this, why add this proposed feature to Ruby at all. As originally suggested, just allow user code to intercept rb_wait_for_single_fd and rb_waitpid and write a gem to solve the problem of non-blocking IO. The surface area is much smaller than what you propose here. The API can simply be those two functions but in Ruby land. Allow users to set it on a per-thread basis. As in:

```
Thread.new do
    Thread.current.selector = NIO::Selector.new
    # selector responds to :wait_for_single_fd

    # Thread at exit calls selector.run
end
```

async-io does it pretty well, and it does it using wrappers which are a pain to inject into existing code. All that's needed for async-io to work in general, is a way to intercept these calls. Then, it will do everything possible here, and more.

As has been demonstrated, there are lots of trade-offs. Personally, I'd rather use libev or libuv which is actively maintained than what you've proposed here. They cover a much larger chunk of functionality, and they are maintained and updated independently of Ruby. Why repeat all that work? Are you going to maintain this feature for the next 20 years?

If you really believe in minimal surface area, the above proposal is about as minimal as it gets. Not only that, it's easier for JRuby and others to implement, and it might even work for MRuby. You don't need to add all this new C code to Ruby itself, either make a gem with your proposed selector design, or use an existing one (nio4r) or we can experiment with libuv/libev/libevent. We can already see that the general trend for Ruby is to minimise the standard library and add more code to gems (https://stdgems.org).

If you want some kind of default:

```
class Thriber
    def initialize
        self.selector = $selector
    end
end
```

It's an elegant and simple design with sane defaults and flexibility for the future.

#102 - 05/05/2018 01:06 PM - ioquatix (Samuel Williams)

I've been thinking about these issues for the past few days, and one thing I found was this video https://www.youtube.com/watch?v=KXuZi9aeGTw#t=519 where they specifically talk about the latencies incurred by the threading model you propose. I just thought you might find it interesting.

#103 - 05/06/2018 03:04 AM - normalperson (Eric Wong)

samuel@oriontransfer.net wrote:

> I've been thinking about these issues for the past few days, and one thing I found was this video
> https://www.youtube.com/watch?v=KXuZi9aeGTw#t=519 where they specifically talk about the latencies incurred by the threading model you
> propose. I just thought you might find it interesting.

Can you please summarize and perhaps link to code they talk about?

I don't have video now and I don't want it to become an expectation
that Ruby contributors be able to deal with video
(either through hardware, software or human physical limitation)

No rush, either, barely have any computer access for a bit.

Again, Mio (Glasgow Haskell Compiler) is most similar to what
I'm working on, here, and that scales to some ridiculous number
of cores:

http://haskell.cs.yale.edu/wp-content/uploads/2013/08/hask035-voellmy.pdf

I don't use a dedicated epoll thread like they do;
but I also know epoll_wait uses exclusive wakeup in Linux,
so there's no thundering herd when a single event arrives
when multiple threads are sleeping in epoll_wait.

#104 - 05/07/2018 11:39 AM - ioquatix (Samuel Williams)

> Can you please summarize and perhaps link to code they talk about?

The simplest summary I can give is that OS schedulers don't know what threads to wake up for optimal IO latency. So, Google implemented an API for cooperatively scheduling user threads within the OS time slice. They could have, say, a thread receiving events from the kernel via epoll, and then "yield" to the thread that was waiting for that event without requiring the OS to schedule it.

> Again, Mio (Glasgow Haskell Compiler) is most similar to what
> I'm working on, here, and that scales to some ridiculous number
> of cores

Thanks for that I will read it.

I will assume you've implemented something similar, but I have a question, when processing events, when handling events that don't belong to the current thread, you put them in a queue. How does the other thread know to wake up?

#105 - 05/08/2018 05:25 AM - ioquatix (Samuel Williams)

I studied that Mio paper.

It was very interesting.

I made a similar server in C++ using my async toolkit (the version for C++). It handled almost 400k recv/send IOs on a 4 core CPU. It was a similar implementation to Mio, but it uses the async design of one selector per thread.

My CPU is about the age of that research paper. In that paper, it looks like they could only get around 200-300K IOs with the same configuration. In fact, in their sample server (C implementation), they didn't have any kind of shared connection state, it was purely event driven.

Also, after checking, I realised they are using one epoll per worker in their C implementation:
https://github.com/AndreasVoellmy/epollbug/blob/master/SimpleServerC.c

I will continue to study the research paper, but I believe one selector per thread is a good design. It's simple, easy to implement, and scales very well.

#106 - 05/08/2018 06:40 AM - ioquatix (Samuel Williams)

Sorry, one more question.

I'm interested in the peformance of EPOLLONESHOT.

Is it correct that EPOLL_CTL_(ADD/DEL) create more contention in the kernel locks than EPOLL_CTL_MOD? So, we should prefer CTL_MOD because it should be more efficient in the kernel?

#107 - 05/08/2018 07:01 AM - ioquatix (Samuel Williams)

I hacked in EPOLLONESHOT semantics into my runloop. IT was about the same performance. But when I leveraged it correctly (calling EPOLL_CTL_ADD when accepting IO once, and EPOLL_CTL_DEL when closing IO, then EPOLL_CTL_MOD when waiting for event), I saw a 25% improvement in throughput. It was just a very rough test case but interesting none the less.

I then reverted that change but retained the "calling EPOLL_CTL_ADD when accepting IO once, and EPOLL_CTL_DEL when closing IO", and saw another... ~10% increase in throughput. So, it's of course, just related to making syscalls and the general performance issues they introduce. The less system calls the better.

However, the % time is so low, that any kind of higher level parsing/processing is going to fully dwarf a single system call.

#108 - 05/10/2018 08:12 PM - normalperson (Eric Wong)

samuel@oriontransfer.net wrote:

> > Again, Mio (Glasgow Haskell Compiler) is most similar to what
> > I'm working on, here, and that scales to some ridiculous number
> > of cores
>
> Thanks for that I will read it.

Fwiw, I've been citing Mio since before this feature was implemented...

> I will assume you've implemented something similar, but I have
> a question, when processing events, when handling events that
> don't belong to the current thread, you put them in a queue.
> How does the other thread know to wake up?

It enqueues an interrupt for the target thread, same thing as
"normal" thread switching. Ruby uses a 100ms timeslice with
pthreads.

#109 - 05/10/2018 09:12 PM - normalperson (Eric Wong)

samuel@oriontransfer.net wrote:

> I hacked in EPOLLONESHOT semantics into my runloop. IT was about the same performance. But when I leveraged it correctly (calling EPOLL_CTL_ADD when accepting IO once, and EPOLL_CTL_DEL when closing IO, then EPOLL_CTL_MOD when waiting for event), I saw a 25% improvement in throughput. It was just a very rough test case but interesting none the less.

I would not expect one-shot to improve things unless you design
your application around it. It also won't help if you only
expect to deal with well-behaved clients and your application
processing times are uniform.

One-shot helps with application design and allows in resource
migration when sharing the queue across threads. Again, this
design may harm overall throughput and performance under IDEAL
conditions. That's because there is a single queue and assumes
all requests can be processed at roughly the same speed.

However in NON-IDEAL conditions, some endpoints are handled more
slowly than others. They are slow because the application needs
to do more work, like an expensive calculation or FS access,
NOT because of a "slow client".

One-shot also makes application design easier when an evil
client which is aggressively pipelining requests to request
large responses, yet reading slowly. Thus, the evil client is
fast at writing requests, but slow at reading responses.

Server and reactor designers sometimes don't consider this case:
I haven't checked in years, but EventMachine was a huge offender
here since it didn't allow disabling read callbacks at all.
What happened was evil clients could keep sending requests,
and the server would keep processing them and writing responses
to a userspace buffer which the evil client was never draining.
So, eventually, it would trigger OOM on the server.

Non-oneshot reactor designs need to consider this attack vector.
One-shot designs don't even need to think about it, because it's
not "reacting" with callbacks. One-shot uses EPOLL_CTL_MOD (or
EV_ADD) only when the reader/writer hits EAGAIN. With one-shot
you won't have to deal with disabling callbacks which blindly
"react" to whatever evil clients send you.

So in my experience, one-shot saves me a lot of time since I
don't have to keep track of as much state in userspace and
remember to disable callbacks from firing if an evil client is
sending requests faster than they're reading them.

#110 - 05/11/2018 02:08 AM - ioquatix (Samuel Williams)

> Server and reactor designers sometimes don't consider this case:
> I haven't checked in years, but EventMachine was a huge offender
> here since it didn't allow disabling read callbacks at all.

I see. That makes sense.

Well, async-io doesn't have this problem AFAIK, because it doesn't have callbacks like this. It only read when you ask it to. That doesn't stop you putting read into a loop though.

#111 - 06/13/2018 01:22 AM - normalperson (Eric Wong)

I don't have a name for this yet, but I hope to work on Queue/SizedQueue

support soonish.

Anyways, rebased against r63641 and it shows reasonable
performance: [ruby-core:87483] [Feature #14736]

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

Or via "git fetch":

The following changes since commit c603e5c9ab09896ed269b90a637177673914a9a5:

- 2018-06-13 (2018-06-12 20:22:44 +0000)

are available in the Git repository at:

git://80x24.org/ruby.git threadlet-r63641

for you to fetch changes up to 8ac308cde533b84256fb9043e9e2360315aa8727:

Threadlet: green threads implemented using fibers (2018-06-13 00:34:32 +0000)

#112 - 06/18/2018 01:04 AM - normalperson (Eric Wong)

Also, I will extract timeout support into separate feature.

The basic idea is that (regardless of how efficient(*) Timeout is):

```
Timeout.timeout { io.read(...) }
Timeout.timeout { io.write(...) }
Timeout.timeout { io.read(...) }
Timeout.timeout { io.write(...) }
...
```

Will always be less efficient than:

```
Timeout.timeout do
  io.read(...)
  io.write(...)
  io.read(...)
  io.write(...)
  ...
end
```

So we should encourage the latter, not the former (as this patch does).
I will try to make both as fast as possible, but the former will
always be faster because registering the timeout (either in
userspace in our VM or the OS kernel) always has a highish cost.

#113 - 07/04/2018 07:37 AM - funny_falcon (Yura Sokolov)

Considering implementation, Fiber.transfer should be used, not Fiber.resume+Fiber.yield,
ie exclusively use fiber_switch(..., ..., ..., 0); and never use fiber_yield(..., ...)+fuber_resume(...,...,...).
Reason: to allow nested fiber calls to be scheduled as well.
Example:

Lets have following class

```
class MyPipe
  def initialize(host, port)
    @con = TCPSocket.new(host, port)
  end
  def pipe
    while s = @con.gets
      yield s
    end
  end
end
mypipe = MyPipe.new
```

Now lets iterate:

```
mypipe.pipe{|s| puts s}
```

So far so good.
But now lets use it as enumerator:

```
mypipe.to_enum(:pipe).each{|s| puts s}
```

BAAHHHMMMM!!!! Either we have to fallback to blocking io at @conn.gets,
or fiber_yield inside of scheduler will yield our puts s block, instead of returning to scheduler.

Both Enumerator and io scheduler could not use same fiber_yield.
There should be scheduler fiber as a property of native thread (probably, its main fiber),
and scheduled "Threadlets" should transfer control to it, not yield to.

Second remark about design: please, do not create new beast!
Ruby has Thread, Fiber, Continuation... It doesn't need new Thread::Light, Fred or any thing else!
Let it be just Thread!!!
In my opinion, Thread should be either "green" by default, or by constructor parameter.
Looks like, for backward compatibility, it is better to have native threads still default:

- first, all disk io heavy application will not magically turn into turtles after Ruby's version upgrade (so, Eric's application will work unmodified)
- aside of disk io and network io, there are heavy computations, that are also wrapped with "release GVL".

I think, it could be good to have possibility to specify different scheduler variants:

- Thread.new(scheduler: :current) - create thread in a scheduler of current native thread.
- Thread.new(scheduler: :main) - create thread in a scheduler of main native thread.
- Thread.new(scheduler: :new) - create new native thread, and schedule new thread there.
- Thread.new(scheduler: other_thread.scheduler) - create new thread in a scheduler of other thread.
- Thread.new(scheduler: :exclusive) - create new native thread without scheduler, and run Thread on a native thread stack.

Default Thread.new should be synonym for Thread.new(scheduler: :new) or Thread.new(scheduler: :exclusive).

Mutex and Queue should be aware of new scheduling model, because there will be only one Thread class.

To be honest, I'd prefer Thread to be green by default.
But given, it is desirable to accept this change before 3.0, it is better to keep compatible behavior, i think.

#114 - 07/04/2018 08:45 AM - ioquatix (Samuel Williams)

funny_falcon (Yura Sokolov) I don't think it's easy to change Thread.new(...) because arguments are passed to the thread block.

However, if you want an API which doesn't introduce any new class, please check out https://bugs.ruby-lang.org/issues/14736 and give me feedback.

#115 - 07/04/2018 04:40 PM - funny_falcon (Yura Sokolov)

ioquatix (Samuel Williams) nothing prevents from adding new method:
Thread.create(scheduler: :current, args: [block_arg1, block_arg2]){|arg1, arg2| ... }

Just remark: make test example to use Fiber.transfer for the reason above.

#116 - 07/05/2018 07:19 AM - funny_falcon (Yura Sokolov)

Looks like I was not exactly right: typical iteration over Enumerator doesn't use Fiber.yield.
But usage of Enumerator as external iterator does. And zip method does use 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]]
```

#117 - 07/05/2018 08:43 AM - funny_falcon (Yura Sokolov)

ioquatix (Samuel Williams) About "extendable api" vs "implicit behavior" (ie between your proposal
for "Thread.scheduler=' and replacing "Thread" with green implementation),
I'd prefer "implicit behavior".
Because, there were already EM::Synchrony, there were ... ah I've already forgot what there were.
If it is not part of internal behavior, there always will be non-compatible libraries.

Note, that changing Thread to green in CRuby in fact more compatible with other implementations,
because other implementation could still have only native threads, and they still will work.
Ie JRuby may safely not change their Thread implementation to be green, and all libraries, that
uses threads, will still work on JRuby, because api will not change. JRuby will need just declare
create method that will ignore scheduler argument.

#118 - 07/05/2018 09:35 AM - ioquatix (Samuel Williams)

Thanks for your feedback.

> About "extendable api" vs "implicit behavior" (ie between your proposal
> for "Thread.scheduler=' and replacing "Thread" with green implementation),
> I'd prefer "implicit behavior".
> Because, there were already EM::Synchrony, there were ... ah I've already forgot what there were.
> If it is not part of internal behavior, there always will be non-compatible libraries.

I don't see how that comparison applies to Thread.scheduler PR. Because it's completely transparent to higher level code.

In fact, if anything, the Thread.scheduler PR is more implicit than using Thread.create since all user code would need to be modified to use this new API, but Thread.scheduler can work without any changes to user code.

It's also going to be more useful for existing code bases like ActionCable, Puma, Async, etc which use their own IO scheduler.

There is no way you can not be compatible with Thread.scheduler if you use standard Ruby IO. Can you give me an example where this isn't true?

> Note, that changing Thread to green in CRuby in fact more compatible with other implementations,
> because other implementation could still have only native threads, and they still will work.
> Ie JRuby may safely not change their Thread implementation to be green, and all libraries, that
> uses threads, will still work on JRuby, because api will not change. JRuby will need just declare
> create method that will ignore scheduler argument.

Yes, but you pay all the overhead of "it might be a thread", and gain none of the benefits of green threads. It has non trivial costs both in performance and programmer sanity. Programmer sanity is much more important to me than performance, but performance is very important too.

In my experience, most people don't care about lower level. They just want a nice high level library.

#119 - 07/05/2018 06:12 PM - funny_falcon (Yura Sokolov)

> It's also going to be more useful for existing code bases like ActionCable, Puma, Async, etc which use their own IO scheduler.

They have their own IO scheduler because ruby had just native threads, which are bad as IO scheduler.
Ok, I'm not totally right: no scheduler will be good enough for everyone.
But I believe, single green scheduler will be good enough for most of things.

> There is no way you can not be compatible with Thread.scheduler if you use standard Ruby IO. Can you give me an example where this isn't true?

No, looks like I'm not confident to answer :-(
Edit: after writing rest of this, I've recognized, that standard Ruby Mutex and Queue will not be compatible with Thread.scheduler.

> Yes, but you pay all the overhead of "it might be a thread", and gain none of the benefits of green threads.

It has benefits both performance and uniformity benefits.
It will be fast, because scheduler still can switch "threads" sitting on a same native thread as fast as Fibers.

Uniformity, because there will be just single set of tools for synchronization: Mutex, ConditionVariable, Queue.
All these tools are needed regardless of "native" vs "green" scheduler.
If these utils will be universal, they will be easily composed together.
Otherwise mix of "native"/"green" threading will become nightmare.
How they will be composed with Thread.scheduler?

> Programmer sanity is much more important to me than performance.

That is my stand point too. I believe, less things programmer need to teach, is better.

So there should be:

- use Threads, Mutex, Queue. If you want performance of eventloop, pass scheduler parameter to Thread.create. That is all.

With Thread.scheduler= it becomes:

- you may use Threads, Mutex, Queue.
- but if you want performance of eventloop, you need to choose library, that provides scheduler, Mutex, Queue, use that library's primitives thorough your code, and never mix core Mutex with that library's Mutex, if you occasionally need to use native threads.

Seriously: Ruby will never be that low level language that will gain serious performance through careful
separation of "green" vs "native" thread concepts.

Look at Go (yeah, i've said that, sorry): it were built to be fast practical language.
It has "green threads". But it has no separation "green vs native".
Single option, that digs into that separation, is "runtime.LockThread()" to give the goroutine separate scheduler on separate native thread.
And Go have no non-blocking io visible to user :-O . It is pretty annoying.

Sure, if one want to gain 99.99% of hardware performance, one will not use Go.
She will use C/C++/Rust, will build their own scheduler and event loop.

But if 95% is just ok, than Go is right tool.

Doubtfully there will be so huge performance difference between
"explicit Thread.scheduler= + that's scheduler synchronization primitives"
vs "standard hybrid Thread with standard hybrid Mutex/Queue".

Sure, hybrid Threads will be much harder to accomplish.
Sure, I could be mistaken entirely.

#120 - 07/05/2018 09:54 PM - ioquatix (Samuel Williams)

> They have their own IO scheduler because ruby had just native threads, which are bad as IO scheduler.

Thanks so much for your answer, it's very detailed and gives me a clear picture about what you are thinking.

In my experience, threads simply don't scale as as well as fibers, there is too much overhead. That being said, you are right they are sort of normative, the defacto, mechanism by which all things can become asynchronous. The problem with Ruby threads is that the are mutually exclusive when running Ruby code so they are pretty tricky to use in practice.

> With Thread.scheduler= it becomes: ....

There is no need to ever use Mutex, Queue within Thread.scheduler= threads. That's the whole point. Because it's cooperative concurrency. Just spawn your fibers, and the scheduler will swap them out when the block. If you want a "queue", use an array. Or, whatever data structure suits your requirements.

> Doubtfully there will be so huge performance difference between
> "explicit Thread.scheduler= + that's scheduler synchronization primitives"
> vs "standard hybrid Thread with standard hybrid Mutex/Queue".

In my experience, there is a large difference. As you saw in my article comparing Puma with Falcon.

> Ruby will never be that low level language that will gain serious performance through careful separation of "green" vs "native" thread concepts.

Falcon shows that it is possible, but replace in your statement native threads with processes due to limitations in Ruby.

#121 - 07/06/2018 07:48 AM - funny_falcon (Yura Sokolov)

> In my experience, threads simply don't scale as as well as fibers, there is too much overhead.

Native threads doesn't scale.
But we have example of Go: goroutine is a really green thread, and they do really scale.
And Ruby 1.8 had green threads. With patches from RubyEE, they were really fast.

Word "Thread" should not be red flag. It is just a word.
Bunch of hybrid threads scheduled on one native thread will be as fast as Fiber's, they will scale.

> The problem with Ruby threads is that the are mutually exclusive when running Ruby code

Fibers are also exclusive relative to each other.

> There is no need to ever use Mutex, Queue within Thread.scheduler= threads.

Excuse me for rude word, but it is bullshit. It is so often repeated, that every one starts to believe.
But reality is if there is concurrency (either due to parallelism, or due to asynchronous execution),
there is need for synchronization.
Especially it is strange to here from you, because socketry/async has Queue, Condition and Semaphore.

> There is no need to ever use Mutex, Queue within Thread.scheduler= threads.

Again: then why did you then did them for async?
If your Queue dosn't use native Mutex, that doesn't make your Queue different beast.
It is still synchronization mechanism, that have to be present in standard library.
And it should be fully concerned about all possible schedulers.

Of course, in the world of proposed hybrid Threads, there always will be a overhead
for Mutex, Queue etc for being concerned about hybrid threads. But overhead will be
really small if hybrid Threads are scheduled on a same native thread.

But what I'm talking about? CRuby already has really fast Mutex and Queue
implementation that doesn't use native mutex (beside of GVL).
Then which way they differs from Async::Semaphore (with limit=1) and Async::Queue ?
Single difference is they signals OS scheduler instead of reactor.
But doubtfully it will be too hard to change them to signal hybrid scheduler.

#122 - 07/06/2018 09:16 AM - ioquatix (Samuel Williams)

> But we have example of Go: goroutine is a really green thread, and they do really scale.

Yes, they are great, but it's probably impossible to implement in Ruby, and it still requires a lot of non-trivial synchronisation.

> Bunch of hybrid threads scheduled on one native thread will be as fast as Fiber's, they will scale.

Yes they will, but they will not be as pleasant to program. The memory and execution model is very complex for normal people to grok.

The memory model and execution model of fibers is very simple. I've had feedback from people who have used Async, and it's all been really great.

> Fibers are also exclusive relative to each other.

Yes, but by design, not by limitation of the interpreter (ala Threads/GVL).

> > There is no need to ever use Mutex, Queue within Thread.scheduler= threads.
>
> Excuse me for rude word, but it is bullshit. It is so often repeated, that every one starts to believe.

You cannot use primitives designed for thread synchronisation because it will block the entire thread, and it won't allow other fibers to execute. I didn't say that Async doesn't have synchronisation primitives.

> Then which way they differs from Async::Semaphore (with limit=1) and Async::Queue ?

Async doesn't have Mutex, since all fibers in a thread/reactor is naturally mutually exclusive. The implementation of the Async primitives leverages the concurrency model of fibers to make them simple, deterministic and robust.

In my mind Thread.scheduler doesn't require built in primitives, the underlying primitive is the mutual exclusion imposed by fibers. Anyone can build "primitives" like semaphore, queue, condition, etc. The same can not be said for Threads.

#123 - 07/06/2018 06:10 PM - funny_falcon (Yura Sokolov)

> Yes, they are great, but it's probably impossible to implement in Ruby.

It is impossible to implement Thread migration between native threads. All other is possible.

> Bunch of hybrid threads scheduled on one native thread will be as fast as Fiber's, they will scale.

> Yes they will, but they will not be as pleasant to program. The memory and execution model is very complex for normal people to grok. The memory model and execution model of fibers is very simple. I've had feedback from people who have used Async, and it's all been really great.

Don't get me wrong: most of code is linear and thread-safe. Synchronization is needed only for framework authors.
You (as Async author) see the need for synchronyzation. Most of ordinal programmers will never touch it.
More over: if language gives suitable synchronization primitives, it is not difficult to explain to average programmer how to use it.

Go (excuse me again for mention it) has a rich story on that:
99.9% of code written daily doesn't bother with concurrency at all.
I see every day hundreds lines are committed without any call to "Mutex" or even channel.
That is usual code, and it will work despite working on Thread or Fiber.

But, when programmer want to dig into concurrency, it is better to have tool that will work always,
ie it is better to have builtin scheduler and single set of synchronization primitives.

> Fibers are also exclusive relative to each other.

> Yes, but by design, not by limitation of the interpreter (ala Threads/GVL).

Again, I really think, there is no need to rely on "exclusive by design" in 99% cases.
And Ruby is not that language, where that 1% will gain much difference.

More over, my experience taught me, that every time I rely on "exclusive by design",
it bites me. Because today it has no yield point, tomorrow some one added "log.write()",
which wrote to network log collector, and BAHM, it yields and breaks everything.
If something should be protected, it have to be protected with primitive language gives me,
otherwise it is sleeping bug that will fire in a future.

> You cannot use primitives designed for thread synchronization because it will block the entire thread, and it won't allow other fibers to execute.

Unless you have green Thread, that is scheduled on same native thread, and synchronization primitive is concerned about.
Go's sync.Mutex works very well, regardless of number of native threads it works over.
And I don't see any reason, why Ruby's Mutex will be worse.

> Async doesn't have Mutex, since all fibers in a thread/reactor is naturally mutually exclusive.

What is Async::Semaphore.new(1) ? It is Async::Mutex, just without separate name.

> The implementation of the Async primitives leverages the concurrency model of fibers to make them simple, deterministic and robust.

An implementation of standard library's primitives will leverage the concurrency model of hybrid Threads to make them simple, deterministic and robust.

> In my mind Thread.scheduler doesn't require built in primitives

It will require primitives, provided by same library, that provides scheduler.

> Anyone can build "primitives" like semaphore, queue, condition, etc.

Anyone will have to use primitives shipped with the library that provides scheduler.
You told about "simplicity for user", but building this primitives is not easy.
Worse thing: any generic library, that wants to use primitives, but don't want to
rely on single "scheduler library" will have to have a way to find correct primitive
for each possible scheduler library.

> The same can not be said for Threads.

With Threads, there will be Mutex, Queue, ConditionVariable in standard library.
Programmer doesn't have to reimplement the wheel. Especially if it is in standard library.

That is my point:

- you says "it will be easy to reimplement wheels",
- I say "but I don't want to reimplement wheels".

I really did implement wheels.
On top of EventMachine and EM::Synchrony I've made a lot of things many years ago.
I really do not wish for average programmer to step through that.
I want average programmer to take base Ruby installation, make a program with standard library,
and that program should run smoothly and fast. I want they could combine any gems, and that
gems doesn't fight against each other because they wants different schedulers.

Do you know, why Go is great (excuse me again)?
Because you have no much choice, but default choice is already great.
People get standard library, they get standard (and single possible) runtime, and they already can
do great things.

People don't want to make a choice. People wants to make a product.

That is what people got from Ruby too in a past.
It was before "asynchronous" programming became to be main stream.
But now there too many choices (for asynchronous programming),
and they are dying with the speed of birth.

I wish it will be:
"Use standard Thread.create.
It is fast, and scheduled asynchronously using hidden eventloop by default.
But if you really need to deal heavy with disk, or to do CPU calculation, implemented in C,
then spawn @pool=Thread::NativePool.new(10), and pass jobs to that pool with
result = @pool.do{ mytask } or future = @pool.push{ mytask }; future.get".

Looks like I'm too wordy and too emotional. Excuse me :-(

#124 - 07/06/2018 09:12 PM - normalperson (Eric Wong)

funny.falcon@gmail.com wrote:

> It is impossible to implement Thread migration between native
> threads. All other is possible.

It may be possible but we'd either lose performance and/or break
C extensions. So it won't happen soon :)

General update on this topic: I've been piggy-backing
groundwork for this features into trunk as separate issues:

- SIGCHLD-based rb_waitpid into trunk to fix a problem with
  MJIT and it finally appears stable across different platforms:
  https://bugs.ruby-lang.org/issues/14867
  It also helped me understand some portability quirks and
  differences of non-Free platforms I don't use, so I will
  be able to workaround them more easily in the future.

- Since many methods to be affected by green threads need
  timeout args, I'm working on implementing Timeout in the VM:
  https://bugs.ruby-lang.org/issues/14859
  Green-threads versions of rb_wait_for_single_fd, rb_thread_sleep*,
  rb_thread_select, etc... will need to manage their own timeouts,
  anyways.

- Queue+SizedQueue MUST work with this feature, as green thread
  need to communicate with each other. Same as "mailbox" or
  in actor model or pipes or whatever other languages call it.
  Groundwork for this was already laid in 2.5:
  https://bugs.ruby-lang.org/issues/13552

There will only be one API addition affecting green-thread
creation. Right now, most likely candidates are:

- Thread::Green.new {} # Just like Thread.new, but with "::Green"
- Thread.green {} # 5 characters shorter than above
- Thread.create(scheduler: ..., args:...) {} # maybe too verbose

#125 - 08/08/2018 01:21 AM - ioquatix (Samuel Williams)

Eric that is a great update.

I've been playing around with my gem async and I've come to the conclusion that it is a great way to do IO, but it does have some cases that need to be considered carefully.

In particular, when handling HTTP/2 with multiple streams, it's tricky to get good performance because utilising multiple threads is basically impossible (and this applies to Ruby in general). With HTTP/1, multiple "streams" could be easily multiplexed across multiple processes easily.

What this means is that a single HTTP/2 connection, even with multiple streams, is limited to a single thread with the fiver-based/green-thread design.

I actually see two sids to this: It limits bad connections to a single thread, which is actually a feature in some ways. On the other hand, you can't completely depend on multiplexing HTTP/2 streams to improve performance.

On the other hand, any green-thread based design is probably going to suffer from this problem, unless a work pool is used for actually generating responses. In the case of async-http, it exposes streaming requests and responses, so this isn't very easy to achieve.

I've also been thinking about timeouts.

I've been thinking about adding a general timeout to all socket operations. The user can set some global default, (or even set it to nil). When the user calls io.read or io.write there is an implicit timeout. I'm not sure if this is a good approach, but I don't think it's stupid, since io operations are naturally temporal so some kind of default temporal limit makes sense.

#126 - 08/08/2018 08:53 AM - normalperson (Eric Wong)

samuel@oriontransfer.net wrote:

> I've been playing around with my gem async and I've come to
> the conclusion that it is a great way to do IO, but it does
> have some cases that need to be considered carefully.

Right.

> In particular, when handling HTTP/2 with multiple streams,
> it's tricky to get good performance because utilising multiple
> threads is basically impossible (and this applies to Ruby in
> general). With HTTP/1, multiple "streams" could be easily
> multiplexed across multiple processes easily.

I'm no expert on HTTP/2, but I don't believe HTTP/2 was built
for high-throughput in mind. By "high-throughput", I mean
capable of maxing out the physical network or storage.

At least, multiplexing multiple streams over a single TCP
connection doesn't make any sense as a way to improve
throughput. Rather, HTTP/2 was meant to reduce latency by
avoiding TCP connection setup overhead, and maybe avoiding
slow-start-after-idle (by having less idle time). In other
words, HTTP/2 aims to make better use of a
heavy-in-memory-but-often-idle resource.

> What this means is that a single HTTP/2 connection, even with
> multiple streams, is limited to a single thread with the
> fiver-based/green-thread design.

I don't see that is a big deal because of what I wrote above.

> I actually see two sids to this: It limits bad connections to
> a single thread, which is actually a feature in some ways. On
> the other hand, you can't completely depend on multiplexing
> HTTP/2 streams to improve performance.

Right.

> On the other hand, any green-thread based design is probably going to suffer from this problem, unless a work pool is used for actually generating responses. In the case of async-http, it exposes streaming requests and responses, so this isn't very easy to achieve.

Exactly. As I've been say all aalong: use different concurrency primitives for different things. fork (or Guilds) for CPU/memory-bound processing; green threads and/or nonblocking I/O for low-throughput transfers (virtually all public Internet stuff), native Threads for high-throughput transfers (local/LAN/LFN).

So you could use a green thread to coordinate work to the work pool (forked processes), and still use a green thread to serialize the low-throughput response back to the client.

This is also why it's desirable (but not a priority) to be able to migrate green-threads to different Threads/Guilds for load balancing. Different stages of an application response will shift from being CPU/memory-bound to low-throughput trickles.

> I've also been thinking about timeouts.
>
> I've been thinking about adding a general timeout to all socket operations. The user can set some global default, (or even set it to nil). When the user calls io.read or io.write there is an implicit timeout. I'm not sure if this is a good approach, but I don't think it's stupid, since io operations are naturally temporal so some kind of default temporal limit makes sense.

Timeout-in-VM [Feature #14859] will be most optimized for apps using the same timeout all around. I'm not sure it's necessary to add a new API for this if we already have what's in timeout.rb

Also, adding a timeout arg to every single io.read/io.write call is going to be worse for performance, because every timeout use requires arming/disarming a timer. Whereas a single "Timeout.timeout" call only arms/disarms the timer once.

#127 - 08/08/2018 11:22 AM - phluid61 (Matthew Kerwin)

[snipping some bits, because I can only speak to what I know]

On Wed, 8 Aug 2018 at 18:50, Eric Wong normalperson@yhbt.net wrote:

> samuel@oriontransfer.net wrote:
>
> > In particular, when handling HTTP/2 with multiple streams, it's tricky to get good performance because utilising multiple threads is basically impossible (and this applies to Ruby in general). With HTTP/1, multiple "streams" could be easily multiplexed across multiple processes easily.
>
> I'm no expert on HTTP/2, but I don't believe HTTP/2 was built for high-throughput in mind. By "high-throughput", I mean capable of maxing out the physical network or storage.

It was originally invented to reduce perceived latency, both in terms of time-to-first-paint and time-to-last-byte, in solitary servers as well as data centres and CDNs. As such throughput was definitely a goal, but not the only one.

There is some synchronisation: the server has to read a few bytes of each frame it receives before it can demux them to independent handlers; and when transmitting you have to block for CONTINUATION frames if any are in progress, and for flow control if you're sending DATA. But aside from those bottlenecks, each request/response can be

handled completely in parallel. Does that really have that big of an impact on throughput?

> At least, multiplexing multiple streams over a single TCP connection doesn't make any sense as a way to improve throughput. Rather, HTTP/2 was meant to reduce latency by avoiding TCP connection setup overhead, and maybe avoiding slow-start-after-idle (by having less idle time). In other words, HTTP/2 aims to make better use of a heavy-in-memory-but-often-idle resource.

It shouldn't be that hard to saturate your network card, if you've got enough data to write, and the other end can consume it fast enough. The single TCP connection and application-layer flow control is meant to avoid problems like congestion and bufferbloat, on top of reducing slow-start, TIME_WAIT, etc. so throughput should in theory be pretty high. I guess ramming it all into a single TLS stream doesn't help, as there is some fairly hefty overhead that necessarily runs in a single thread. I'd like to say that's why I argued so hard for https://tools.ietf.org/html/rfc7540#section-3.2 to be included in the spec, but it's actually just coincidental.

> > What this means is that a single HTTP/2 connection, even with multiple streams, is limited to a single thread with the fiver-based/green-thread design.
> >
> > I actually see two sids to this: It limits bad connections to a single thread, which is actually a feature in some ways. On the other hand, you can't completely depend on multiplexing HTTP/2 streams to improve performance.
>
> Right.
>
> > On the other hand, any green-thread based design is probably going to suffer from this problem, unless a work pool is used for actually generating responses. In the case of async-http, it exposes streaming requests and responses, so this isn't very easy to achieve.

Hmm, I think that's what I just said. But then, horses for courses -- if a protocol is designed one way, and an application is designed another, there won't be a great mesh.

> Exactly. As I've been say all aalong: use different concurrency primitives for different things. fork (or Guilds) for CPU/memory-bound processing; green threads and/or nonblocking I/O for low-throughput transfers (virtually all public Internet stuff), native Threads for high-throughput transfers (local/LAN/LFN).
>
> So you could use a green thread to coordinate work to the work pool (forked processes), and still use a green thread to serialize the low-throughput response back to the client.
>
> This is also why it's desirable (but not a priority) to be able to migrate green-threads to different Threads/Guilds for load balancing. Different stages of an application response will shift from being CPU/memory-bound to low-throughput trickles.

Yeah, all of this.

[snipped the rest]

Cheers
--
Matthew Kerwin
https://matthew.kerwin.net.au/

#128 - 08/09/2018 08:04 AM - ioquatix (Samuel Williams)

I agree with everything being said here.

However, just to point out, async doesn't support per-operation timeouts right now. It only supports timeout blocks:

```
timeout(5) do
   # ... some IO operation that blocks, if it takes longer than 5 seconds, would fail with Timeout error
end
```

It works pretty well. That being said, if user forgets to add timeout, then it can become a problem.

#129 - 08/09/2018 08:25 AM - ioquatix (Samuel Williams)

Also, do you think you can make Ruby's native timeout safe? My understanding is that it was a bit unpredictable. With async, the timeout will only affect IO operations, so it is predictable (since you already expect IO operations to fail sometimes).

#130 - 08/09/2018 08:42 AM - normalperson (Eric Wong)

samuel@oriontransfer.net wrote:

> Also, do you think you can make Ruby's native timeout safe? My understanding is that it was a bit unpredictable. With async, the timeout will only affect IO operations, so it is predictable (since you already expect IO operations to fail sometimes).

Right, interrupts make it unpredictable. We can control interrupt timing with Thread.handle_interrupts, but the API is verbose. Moving Timeout into the VM would allow an interrupt-free implementation; so any sleeping calls (Queue#pop, IO.select, waitpid,...) will all be aware of timeouts and not rely on interrupts.

Public API design would require a separate discussion; but first I want to be able to move Timeout into the VM.

#131 - 08/10/2018 09:33 AM - ko1 (Koichi Sasada)

We discussed about naming.

```
X IoThread.new{}
X GreenThread.new{}
X Thread::Green.new{}  #124 “most likely candidates”(vote: hsbt)
X Thread.green{}           #124 “most likely candidates”
X Thread.create(scheduler: …, args: ...) #124 “most likely candidates”
Thread::Coop.new{}
Thread::Cooperative.new{} # (usa)
Thread::Nonpreemptive.new{} # (usa; these 2 names are very long, then they are good ^^)
X NonpreemptiveThread.new {} # (mrkn)
X NPThread.new {} # (mrkn)
X Thread::Light (ko1)
```

X: rejected.

Discussion is follow:

- (1) At first, "green" is rejected because "green" is how to implement and there are several "green threads" can support preemption (such as Ruby 1.8).
- (2) New toplevel name is rejected because is should be under Thread naming.
- (3)Thread.create is rejected because it seems to make Thread class object and auto-fiber should be different class.
- (4) Cooperative is rejected because it is also ambiguous.
- (5) Nonpreemptive is considerable because it is different from what the application programmers (users) want.
- (6) Thread::Light is rejected because this name does not make sense what the class is.

So last devmeeting, we can't conclude the naming issue.

BTW, I think this class should be "Thread" (I'm against (3), which by Matz).

I think auto-fibers are Threads which have special attribute (scheduler). auto-fibers should have same methods in Thread.
I'm not sure why Matz does not like this idea, though.

There are several similar examples:

- Proc (proc and lambda)
- Fiber (semi-coroutine and coroutine (after Fiber#transfer)

Weak examples:

- Hash (compare_by_identity)
- Struct (keyword_init)
- Enumerator (support size or not)

#132 - 08/10/2018 11:44 AM - ioquatix (Samuel Williams)

> Right, interrupts make it unpredictable. We can control

I think as long as it's documented which APIs might cause a timeout to trigger, it's okay. I think when it happens as part of loop iteration, it can be unexpected. If you write code in an exception-safe way, for the most part it shouldn't be a problem. But my experience is that a lot of code is not capable of handling this unexpected flow control. Hence the design of Async and it's timeout mechanism, can only happen when Fiber is resumed.

#133 - 08/14/2018 12:52 AM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> Issue #13618 has been updated by ko1 (Koichi Sasada).
>
> We discussed about naming.

```
X IoThread.new{}
X GreenThread.new{}
X Thread::Green.new{}  #124 “most likely candidates”(vote: hsbt)
X Thread.green{}           #124 “most likely candidates”
X Thread.create(scheduler: …, args: ...) #124 “most likely candidates”
Thread::Coop.new{}
Thread::Cooperative.new{} # (usa)
Thread::Nonpreemptive.new{} # (usa; these 2 names are very long, then they are good ^^)
```

Huh? Why is a long name good? Long names waste screen space
and increase typo errors.

> X NonpreemptiveThread.new {} # (mrkn)
> X NPThread.new {} # (mrkn)
> X Thread::Light (ko1)

Another option: Thread::Coro.new {}

```
X: rejected.

Discussion is follow:

* (1) At first, "green" is rejected because "green" is how to
implement and there are several "green threads" can support
preemption (such as Ruby 1.8).
```

I am thinking of adding preemption support to this feature for
compatibility with 1.8

- (2) New toplevel name is rejected because is should be under Thread naming.

OK. Otherwise, I would push for "Threadlet" name.

- (3)Thread.create is rejected because it seems to make Thread class object and auto-fiber should be different class.

OK

- (4) Cooperative is rejected because it is also ambiguous.

I don't think it's ambiguous, but too long.

- (5) Nonpreemptive is considerable because it is different from what the application programmers (users) want.

Right; and again, I think preemptible can be an option.

- (6) Thread::Light is rejected because this name does not make sense what the class is.

Probably, yes.

> I think auto-fibers are Threads which have special attribute (scheduler).
> auto-fibers should have same methods in Thread.
> I'm not sure why Matz does not like this idea, though.

Agreed, I want to maximize compatibility with code written for
Ruby 1.8, but cost too much memory to run with Ruby 1.9/2.x

> There are several similar examples:
>
> - Proc (proc and lambda)
> - Fiber (semi-coroutine and coroutine (after Fiber#transfer)

So this made me think of "Thread::Coro"

Other ideas: Thread::CSP or Thread::Sequential (probably too long)

https://en.wikipedia.org/wiki/Coroutine
https://en.wikipedia.org/wiki/Communicating_sequential_processes

> Weak examples:
>
> - Hash (compare_by_identity)
> - Struct (keyword_init)
> - Enumerator (support size or not)

Maybe "Thread.sequential {}" is OK.

#134 - 08/14/2018 08:32 AM - ko1 (Koichi Sasada)

On 2018/08/14 9:42, Eric Wong wrote:

> I am thinking of adding preemption support to this feature for
> compatibility with 1.8

So that "auto-fiber" proposal is to provide green threads like Ruby 1.8?

Like:

```
model 1: Userlevel Thread
Same as traditional ruby thread.
```

in thread.c comment (I wrote 13 years ago!).

I don't against this idea, but I think it is hard to select these
options by Ruby programmers. I think changing Thread implementation
model from native thread (1:1 model) to green thread mode (1:N model) is
better for Ruby programmers.

To change them, we need to discuss pros. and cons. of them carefully.
There are several good points (the biggest advantage of 1:1 model is
friendly for outer libraries) but are bad points (1:1 model has
performance penalties, and recent glibc malloc arena issues and so on).

I don't think it is a good idea to choose such internal implementation
by Ruby programmers. ... easy?

--
// SASADA Koichi at atdot dot net

#135 - 08/14/2018 09:00 AM - dsferreira (Daniel Ferreira)

normalperson@yhbt.net wrote:

> I am thinking of adding preemption support to this feature for

> compatibility with 1.8

non-preemptive vs preemptive.
coroutines are non-preemptive.
threads are preemptive.
Are we talking about having the two behaviours in this new feature?

> So this made me think of "Thread::Coro"

What is the logic behind "Coro"?

> Other ideas: Thread::CSP or Thread::Sequential (probably too long)

Does it mean we will have the CSP algebraic operators available?

Reading through this conversation it feels we are dealing with a feature with a lot of concepts incorporated into it.
Can we get a resume of all the functionality we expect to have?
Some code examples would be great.
Or even feature comparison with other languages.
I believe we must do that kind of documentation to show to the community in a clear way the new ruby async possibilities.
I'm willing to help in planning and developing it.

Many thanks.

#136 - 08/14/2018 05:52 PM - normalperson (Eric Wong)

Koichi Sasada ko1@atdot.net wrote:

> On 2018/08/14 9:42, Eric Wong wrote:
>
> > I am thinking of adding preemption support to this feature for
> > compatibility with 1.8
>
> So that "auto-fiber" proposal is to provide green threads like Ruby 1.8?

Yes; this was always the idea.

> Like:
>
> ```
> model 1: Userlevel Thread
>   Same as traditional ruby thread.
> ```
>
> in thread.c comment (I wrote 13 years ago!).
>
> I don't against this idea, but I think it is hard to select these options by
> Ruby programmers. I think changing Thread implementation model from native
> thread (1:1 model) to green thread mode (1:N model) is better for Ruby
> programmers.

No, I want to keep current Thread 1:1 model because it is useful
for filesystem operations and some CPU/memory intensive tasks (zlib).

Changing "Thread" now to 1:N would also break code written for 1.9..2.5

1:N model is good for most network operations (high latency, low throughput).

> To change them, we need to discuss pros. and cons. of them carefully. There
> are several good points (the biggest advantage of 1:1 model is friendly for
> outer libraries) but are bad points (1:1 model has performance penalties,
> and recent glibc malloc arena issues and so on).

Agreed.

> I don't think it is a good idea to choose such internal implementation by
> Ruby programmers. ... easy?

I think it is necessary to give that control to programmers.

We can educate them on pros and cons of each and to use them
in combination.

```
1:1   + C-ext parallelism for SMP systems (example: zlib)
+ filesystem parallelism (no non-blocking I/O on FS ops)
+ external library compatibility
+ compatibility with 1.9..2.5 code

- high memory use (malloc arenas, kernel structs)
- kernel resource limitations (Process::RLIMIT_NPROC)


1:N   + C10K problem (or C100K :P)
+ compatibility with old 1.8 designs
+ low memory and resource use
(malloc and kernel never sees extra data structures)

- cannot take advantage of SMP or multiple filesystems alone
```

The key is a programmer may combine 1:1 and 1:N for different
parts of the program flow. So where the program is bottlenecked
on filesystem, it can rely on 1:1 behavior, but when the program
is waiting on slow network traffic, it can rely on 1:N behavior

For example: I have 4 filesystems, I might have 32 native threads
(8 threads/FS to keep kernel IO scheduler busy).
But I will still serve 100K network clients with 100K 1:N threads
and that can even use 1 native thread.

#137 - 08/14/2018 06:32 PM - normalperson (Eric Wong)

danieldasilvaferreira@gmail.com wrote:

> normalperson@yhbt.net wrote:
>
> > I am thinking of adding preemption support to this feature for
> > compatibility with 1.8
>
> non-preemptive vs preemptive.
> coroutines are non-preemptive.
> threads are preemptive.
> Are we talking about having the two behaviours in this new feature?

"Preemptive" is a minor detail, here. I don't care that much about
it; it is a single bit flag we can implement at a later time.

> > So this made me think of "Thread::Coro"
>
> What is the logic behind "Coro"?

Short for "Coroutine".

> > Other ideas: Thread::CSP or Thread::Sequential (probably too long)
>
> Does it mean we will have the CSP algebraic operators available?

No, so probably "CSP" is not a good name for this. I am not a
formal nomenclature person; I make engineering decisions which
are ultimately sympathetic to:

a) compatibility with existing codebases
b) hardware limitations

> Reading through this conversation it feels we are dealing with a feature with a lot of concepts incorporated into it.
> Can we get a resume of all the functionality we expect to have?

It shouldn't be any different than how Ruby threads are currently used.
Only creation is different:

"Thread.new {}" vs "Thread::Coro.new {}"

> Or even feature comparison with other languages.

I don't know feature details of other languages well enough to comment. Basically, this is re-introduction of green threads from Ruby 1.8; but I still want to keep benefits of 1.9-2.5 native threads.

See my other reply to ko1 in this thread [ruby-core:88484] for pros/cons of both:
http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/88484
(or https://public-inbox.org/ruby-core/20180814174702.GA32360@dcvr/)

However, I don't know other languages (Haskell/GHC, Go, Erlang) well enough to describe APIs; but I know they have lightweight threads (M:N) which use less memory than native 1:1 threads.

What I don't like about transparent M:N threading is (AFAIK for those languages) they don't give programmers a choice about when to use native vs green. M:N threading is fine when you want parallelism in SMP because timeslices are predictable when your bottleneck is CPU/memory on SMP systems.

Implementations of M:N falls down when you want parallelism across multiple filesystems because timeslices become unpredictable. This is a problem for low-end SSDs and HDDs especially, but also network filesystems and USB sticks.

Making the Ruby VM transparently aware of multiple filesystems and bottlenecks/characteristics of each mountpoint may be out-of-scope. I'm not aware of any language runtime takes that into account; so we can leave that to the Ruby user.

Also, C Ruby generally sucks at taking advantage of SMP because of GVL. However, we are currently great at dealing with parallelism across multiple filesystems because of 1:1 threads.

> I believe we must do that kind of documentation to show to the community in a clear way the new ruby async possibilities.
> I'm willing to help in planning and developing it.

The goal is to make migration/testing easy and minimize rewrite cost. So programmers can gsub(/\bThread.new\b/, "Thread::Coro.new") in places where only 1:N threads make sense (see [ruby-core:88484])

#138 - 08/27/2018 08:32 PM - normalperson (Eric Wong)

> https://bugs.ruby-lang.org/issues/13618

Btw, many of my recent changes (Misc #15014), and process.c/thread*.c cleanups in r64542, r64575, r64576, r64577 are preparatory work for this feature that were beneficial regardless.

I have at least one more preparatory change which should be fairly straightforward; but unfortunately makes some of the current code more verbose (because of `container_of'):

https://80x24.org/spew/20180827201123.4364-1-e@80x24.org/raw
"unify sync_waiter, waitpid_state, waiting_fd w/ rb_sched_waiter"

I don't know when I will commit it, yet...

But even if it's more verbose, I think it helps illustrate the concept of using ccan/list for scheduling our native threads.

Autoload has similar thread waiting, but I'm not sure if should be affected by auto-fiber switching.

https://bugs.ruby-lang.org/issues/13618

Greg, do you think you can try this change for portability?
No rush, though, I will be mostly AFK and Queue/SizedQueue
changes will be trivial with the prep work so far...

Future changes to this feature shouldn't impact, portability.
The main one is [PATCH 2/2]

- [PATCH 1/2] unify sync_waiter, waitpid_state, waiting_fd w/ rb_sched_waiter
  https://80x24.org/spew/20180901131012.22138-1-e@80x24.org/raw
- [PATCH 2/2] Thread::Coro: green threads implemented using fibers
  https://80x24.org/spew/20180901131012.22138-2-e@80x24.org/raw

The following changes since commit 929e9713bbfd76140bced29c6f398904ae9d4a85:

complex.c: simplify division result (2018-09-01 07:34:31 +0000)

are available in the Git repository at:

https://80x24.org/ruby.git coro-r64610

for you to fetch changes up to 13c51c9c9ae39dae08497cfe0eb119244c4d2224:

Thread::Coro: green threads implemented using fibers (2018-09-01 12:56:08 +0000)

Newest updates in this version:

- new name: Thread::Coro (reject Thriber/Threadlet)
- Fiber#transfer used for switching
  Thanks to funny.falcon for suggestion [ruby-core:87776],
  it made the code better, even :)
- improve IO.select mapping in kqueue/epoll (st_table => ccan/list),
  since there is no need for lookup, only scan
- sync to use MJIT-friendly rb_waitpid (added th->nogvl_runq
  since waitpid(2) can run without GVL)
- "ensure" support. Only for Thread::Coro, Fiber has no ensure
  (see [Bug #595]). "ensure" is REQUIRED for auto-scheduling
  safety. I am not sure if regular Fiber can support "ensure"
  in a way which is both performant and backwards-compatible.
  With Thread::Coro being a new feature, there is no backwards
  compatibility to worry about, so the "ensure" support adds
  practically no overhead
- more code sharing between iom_{select,kqueue,epoll.h}
- switch to rb_hrtime_t for timeouts
- extract timeout API, so non-timeout-arg users can benefit from
  reduced. This will make Timeout-in-VM support easier and more
  orthogonal to this one.

Changes to existing data structures:

- rb_thread_t .runq - list of Thread::Coros to auto-resume .nogvl_runq - same as above for out-of-GVL use

.waiting_coro - list of blocked Coros, for "ensure"
support. For auto-scheduling, we
must have ensure support because
Fiber does not yet support ensure
<https://bugs.ruby-lang.org/issues/595

- rb_execution_context_t .enode - link to rb_thread_t waiting_coro

rb_vm_t.iom - Ruby I/O Manager (rb_iom_t) :)

As usual, understanding the data structures first should help
you understand the code.

Eric Wong (2):
unify sync_waiter, waitpid_state, waiting_fd w/ rb_sched_waiter
Thread::Coro: green threads implemented using fibers

> Autoload has similar thread waiting, but I'm not sure if should
> be affected by auto-fiber switching.

Along those lines, I'm not sure what the semantic should be with
regards to Mutex. Mutex is to protect data from parallel
modifications, but Thread::Coro do not run in parallel.
if a native Thread and Thread::Coro contend on the same Mutex;
what should happen?

#140 - 09/02/2018 09:32 AM - normalperson (Eric Wong)

> https://bugs.ruby-lang.org/issues/13618
>
> I have at least one more preparatory change which should be fairly
> straightforward; but unfortunately makes some of the current code
> more verbose (because of `container_of'):
>
> https://80x24.org/spew/20180827201123.4364-1-e@80x24.org/raw
> "unify sync_waiter, waitpid_state, waiting_fd w/ rb_sched_waiter"

Here's a work-in-progress patch showing how Queue+SizedQueue would
work with "struct rb_sched_waiter":
https://80x24.org/spew/20180902091557.24130-1-e@80x24.org/raw

It goes on top of my coro-r64610 branch posted at [ruby-core:88800]:
https://public-inbox.org/ruby-core/20180901131301.5peghdyrtmks5mka@dcvr

Several tests are skipped because Thread::Coro#stop?/#status/#[]=
aren't implemented, yet. Will fix later when not AFK.

Another weird case to think about (not supported, yet):

How should Coro switching work inside Signal.trap?

```
q = Queue.new
trap(:INT) { q.push(:INT) }
q.pop
```

I guess we'll need to figure out a way to support it...

#141 - 09/04/2018 09:36 PM - MSP-Greg (Greg L)

normalperson (Eric Wong) Eric,

I applied the two patches in 'note 139', and three test suites stopped.. With test-all running parallel, it's difficult to tell what caused the issue.

btest - stopped on #354 test_insns.rb
spec - the log looked garbled, so I'm not sure.

I've downloaded the build, I'll see if I can run some of the tests locally...

If I misunderstood about what to apply or when, sorry. Thanks, Greg

#142 - 09/05/2018 09:52 PM - normalperson (Eric Wong)

Greg.mpls@gmail.com wrote:

> Issue #13618 has been updated by MSP-Greg (Greg L).
>
> normalperson (Eric Wong) Eric,

> I applied the two patches in 'note 139', and three test suites stopped.. With test-all running parallel, it's difficult to tell what caused the issue.
>
> btest - stopped on #354 test_insns.rb

OK, extremely unexpected because no I/O happens, there.

> spec - the log looked garbled, so I'm not sure.
>
> I've downloaded the build, I'll see if I can run some of the tests locally...
>
> If I misunderstood about what to apply or when, sorry. Thanks, Greg

Thanks Greg,

You can also checkout my git repo @ commit 13c51c9c9ae39dae08497cfe0eb119244c4d2224

https://80x24.org/ruby.git (branch: coro-r64610)

Can you try breaking out the first patch? It's only a
restructuring so there should be no OS-dependent changes:

https://80x24.org/spew/20180827201123.4364-1-e@80x24.org/raw
"unify sync_waiter, waitpid_state, waiting_fd w/ rb_sched_waiter"

Thanks.

I still need to fix some problems with my work-in-progress
changes to support SizedQueue/Queue.

#143 - 09/12/2018 08:32 PM - normalperson (Eric Wong)

> It goes on top of my coro-r64610 branch posted at [ruby-core:88800]:
> https://public-inbox.org/ruby-core/20180901131301.5peghdyrtmks5mka@dcvr
>
> Several tests are skipped because Thread::Coro#stop?/#status/#[]=
> aren't implemented, yet. Will fix later when not AFK.

Implemented; but there's some unrelated bug for kqueue/FreeBSD
I'm still working on...

> Another weird case to think about (not supported, yet):
>
> How should Coro switching work inside Signal.trap?
>
> q = Queue.new
> trap(:INT) { q.push(:INT) }
> q.pop

Still not sure what to do, there.

#144 - 09/13/2018 08:17 AM - matz (Yukihiro Matsumoto)

The latest proposal includes time slice scheduling, so it is not a cooperative thread. I object to the name Thread::Coro. If being lightweight is the biggest characteristic, it should be Thread::Lite or Thread::Light.

Matz.

#145 - 09/13/2018 09:22 AM - normalperson (Eric Wong)

matz@ruby-lang.org wrote:

> The latest proposal includes time slice scheduling, so it is
> not a cooperative thread.

Thanks for the comment.
I haven't implemented timeslice, yet, so I won't :)

> I object to the name Thread::Coro. If being lightweight is

> the biggest characteristic, it should be Thread::Lite or Thread::Light.

Thread::Light is probably acceptable, I still prefer "Coro" in name.

Is "Thread.light {}" shortcut for "Thread::Light.new {}" acceptable?

I am strongly against "Lite", it sounds like an advertisement for foods which may cause cancer.

#146 - 09/13/2018 04:23 PM - shevegen (Robert A. Heiler)

Thread::Light sounds ok to me personally. I think the more informal name "Lite" may have some people wonder about that particular informal name (well aside from the association normalperson had), so I think Light is a better name here.

#147 - 09/21/2018 05:58 PM - shan (Shannon Skipper)

Thread::Light seems nice. Or how about Thread::Feather? A thread as light as a feather.

#148 - 09/28/2018 02:42 AM - normalperson (Eric Wong)

shannonskipper@gmail.com wrote:

> Thread::Light seems nice. Or how about Thread::Feather? A thread as lights as a feather.

Sorry, but no to "Feather". Again, I don't want to introduce new usage of words.

"Light" is common usage as far as "LWP" ("Light-weight process") from some OSes and many usage of "Light-weight" to describe various computing tools and primitives, so more acceptable...

I am only afraid "light" will be confused as colour/brightness by people from UI/UX background.

Of course, Threads and Strings can also be the same thing :>

#149 - 11/14/2018 10:04 PM - normalperson (Eric Wong)

> https://bugs.ruby-lang.org/issues/13618

Should we bother supporting them? AFAIK they're tricky/surprising for regular Thread and maybe we should stick to message passing (e.g. SizedQueue)

It would involve moving th->pending_interrupt_* stuff over to `ec'.

#150 - 11/20/2018 08:52 AM - normalperson (Eric Wong)

> https://bugs.ruby-lang.org/issues/13618

Updated pull request against r65832 with new name: "Thread::Light"

The following changes since commit 8d9a9aab67d6d517995532737a37379c20dc7f76:

thread_pthread.c (rb_reserved_fd_p): false-positive on negative FD (2018-11-20 07:27:28 +0000)

are available in the Git repository at:

https://80x24.org/ruby.git thread-light-r65832

for you to fetch changes up to d03da60f2ef9cf176744168bf2c12b7ab948879c:

Eric Wong (2):
unify sync_waiter, waitpid_state, waiting_fd w/ rb_sched_waiter
Thread::Light: green threads implemented using fibers

2-patch series (squashed history because there were many names for
this feature (auto-fiber/Thriber/Threadlet/Thread::Coro) and now finally
Thread::Light:

https://80x24.org/spew/20181120083413.43523-2-e@80x24.org/raw
https://80x24.org/spew/20181120083413.43523-3-e@80x24.org/raw

This adds following scheduling points:

C-API scheduling points:

- rb_wait_for_single_fd
- rb_thread_fd_select
- rb_waitpid (SIGCHLD platforms only)

These three functions are used by many Ruby methods, which
all now become scheduling points.

additional Ruby API scheduling points:

- Kernel#sleep
- Thread.pass
- IO.copy_stream
- {Queue,SizedQueue}#{push,pop}

FIXME: the "select"-based implementation still has some missed
events problems. Fortunately, relevant production systems
use kqueue or epoll which have no known problems at the moment
(tested FreeBSD 11.2 and Linux 4.19.2)

Thread::Light local storage is implemented (#[], #[]=, #fetch, #key?,
and #keys) just like normal Fibers.

Thread::Light#stop? and Thread::Light#status are analogous to
their regular Thread methods.

Thread::Light#run and Thread::Light#wakeup are supported for
waking up from Kernel#sleep.

Mutex and ConditionVariable are NOT scheduling points for
Thread::Light switching; however they may process signal
handling and handle I/O dispatch for other native threads.

Thread::Light.list does not exist, yet (needed?).

I don't know what to do about Thread#raise/Thread#kill...

#151 - 11/20/2018 10:22 AM - ko1 (Koichi Sasada)

On 2018/11/20 17:44, Eric Wong wrote:

> Mutex and ConditionVariable are NOT scheduling points for
> Thread::Light switching; however they may process signal
> handling and handle I/O dispatch for other native threads.

Why not?
How to synchronize multiple Thread::Light instances (lthreads here)?

--
// SASADA Koichi at atdot dot net

#152 - 11/20/2018 03:22 PM - normalperson (Eric Wong)

Koichi Sasada ko1@atdot.net wrote:

> On 2018/11/20 17:44, Eric Wong wrote:

> Mutex and ConditionVariable are NOT scheduling points for
> Thread::Light switching; however they may process signal
> handling and handle I/O dispatch for other native threads.

Why not?
How to synchronize multiple Thread::Light instances (lthreads here)?

I'm not sure how deadlock detection would work, and I don't
think there is data race there.

Main synchronization should be Queue/SizedQueue (like "mailbox"
in actor model).

But I think this is a good change to maintain compatibility
and avoid inadvertant switching:

```
diff --git a/vm_core.h b/vm_core.h
index 9e10b321da..2244afb524 100644
--- a/vm_core.h
+++ b/vm_core.h
@@ -1842,24 +1842,27 @@ static inline int
 rb_tl_switchable(const rb_execution_context_t *ec)
 {
     const rb_thread_t *th = rb_ec_thread_ptr(ec);

     /* dangerous, don't allow switching inside trap handler: */
     if (ec->interrupt_mask & TRAP_INTERRUPT_MASK) return 0;

+    /* don't switch if a Mutex is held */
+    if (th->keeping_mutexes) return 0;
+
     /* auto-fibers can switch away to root fiber */
     if (rb_tl_sched_p(ec)) return 1;

     /* no auto-fibers, yet, but we can create and switch to them */
     if (!th->root_fiber) return 1;

     /* root fiber can switch to auto-fibers, because ensure works */
     if (th->root_fiber == ec->fiber_ptr) return 1;

     /*
      * no auto-switching away from regular Fibers because they lack
      * ensure support: https://bugs.ruby-lang.org/issues/595
      */
     return 0;
 }

 /* tracer */
```

(Gotta run, back in 16 hours maybe?)

#153 - 11/20/2018 08:58 PM - shevegen (Robert A. Heiler)

I see that this will probably be discussed in the upcoming developer meeting.

I have no particular pro or con to add to any of the functionality, yet alone
internal code-side of it, but I would like to point out that any API decision
exposed, e. g. Thread::Light or Thread::Feather (are you thinking about feathers
of a duck), will also have to be what ruby users may (have to) use. And it
is not that simple for every ruby user to keep track of what to use when,
where and how when it comes to threads in general (there is also mutex and
fibers and who knows what else).

I understand that this issue is not primarily about API design as such and
more about internal handling of the code, but what I am trying to say is to
not forget the average ruby user.

Even "limited" threads have been useful in ruby (remember the old
pickaxe example of downloading many files at the same time via
Thread.new, rather than processing one-file at a time).

If possible I would like to see all the "parallelized" ruby code
to be simple, if possible. Simple like ruby users using String,

Hash, Array, Enumerable ... which I think is a lot simpler than
the thread-related parts of ruby, but this may just be my own
opinion since admittedly I have not written that much ruby code
concerned with threads/mutexes/fibers.

#154 - 11/21/2018 11:03 AM - normalperson (Eric Wong)

> But I think this is a good change to maintain compatibility
> and avoid inadvertant switching:

Updated patch on top of 2/2 to disable switching properly
with Mutex locked:
https://80x24.org/spew/20181121095520.v4ddgpn6lufbvuvt@whir/raw

Updated 2/2 with above patch squashed in (no change to 1/2):
https://80x24.org/spew/20181121105744.14737-1-e@80x24.org/raw

Updated pull request on top of r65903:
The following changes since commit 2f023c5dbaadede9ceac3eb9ac0e73f3050e5ada:

Get rid of variable modifiers of BSD make (2018-11-21 10:09:21 +0000)

are available in the Git repository at:

https://80x24.org/ruby.git thread-light-r65903

for you to fetch changes up to 61f89082a728fa8a37e014378becdcf62bf971f0:

Thread::Light: green threads implemented using fibers (2018-11-21 10:14:06 +0000)

Eric Wong (2):
unify sync_waiter, waitpid_state, waiting_fd w/ rb_sched_waiter
Thread::Light: green threads implemented using fibers

```
common.mk                                          |  15 +
configure.ac                                       |  32 +
cont.c                                             | 437 +++++++++--
eval.c                                             |   1 +
fiber.h                                            |  71 ++
hrtime.h                                           |   8 +
include/ruby/io.h                                  |   2 +
io.c                                               |  36 +-
iom.h                                              | 109 +++
iom_common.h                                       | 253 +++++++
iom_epoll.h                                        | 652 ++++++++++++++++
iom_internal.h                                     | 596 +++++++++++++++
iom_kqueue.h                                       | 804 ++++++++++++++++++++
iom_pingable_common.h                              | 224 ++++++
iom_select.h                                       | 577 +++++++++++++++
process.c                                          | 212 +++---
.../wait_for_single_fd/test_wait_for_single_fd.rb  |  69 ++
test/lib/leakchecker.rb                            |   9 +
test/net/http/test_http.rb                         |   2 +-
test/ruby/test_thread_light.rb                     | 828 +++++++++++++++++++++
test/ruby/test_thread_queue.rb                     |  85 ++-
thread.c                                           | 501 ++++++++++-----
thread_pthread.c                                   |  80 +-
thread_sync.c                                      | 127 ++--
thread_win32.c                                     |  16 +-
vm.c                                               |  10 +
vm_core.h                                          |  63 +-
27 files changed, 5372 insertions(+), 447 deletions(-)
create mode 100644 fiber.h
create mode 100644 iom.h
create mode 100644 iom_common.h
create mode 100644 iom_epoll.h
create mode 100644 iom_internal.h
create mode 100644 iom_kqueue.h
create mode 100644 iom_pingable_common.h
create mode 100644 iom_select.h
create mode 100644 test/ruby/test_thread_light.rb
```

#155 - 11/22/2018 01:28 AM - dm1try (Dmitry Dedov)

normalperson (Eric Wong) thank you for your work!

normalperson (Eric Wong) wrote:

> https://80x24.org/ruby.git thread-light-r65903
>
> for you to fetch changes up to 61f89082a728fa8a37e014378becdcf62bf971f0:
>
> Thread::Light: green threads implemented using fibers (2018-11-21 10:14:06 +0000)

I've just tested this branch(osx/kqueue). It looks like sleep does not work as expected(it blocks forever) inside light thread if some blocking queue is used, see ex:

```
work = Queue.new

Thread.start  do
  5.times do |i|
    work.push(i)
    warn "work##{i} added"
  end

  sleep
end

while i = work.pop
  Thread::Light.start(i) do |i|
    sleep 0.1
    warn "work##{i} done"
  end
end
```

#156 - 11/22/2018 02:22 AM - normalperson (Eric Wong)

me@dmitry.it wrote:

> normalperson (Eric Wong) thank you for your work!

You're welcome.

> I've just tested this branch(osx/kqueue). It looks like sleep does not work as expected(it blocks forever) inside light thread if some blocking queue is used, see ex:

Sorry, sleep is a little wonky and I didn't put much effort
into it since it's rarely used :x. I can reproduce it on
FreeBSD kqueue and Linux, too, so it'll be fixed soon (sorry
busy with personal stuff but should have more time next week)

Thanks for testing. Everything else should work...

I think you also need to join thread::Light to ensure they
finish, otherwise there's nothing to "drive" them.

#157 - 11/22/2018 10:42 AM - normalperson (Eric Wong)

> > I've just tested this branch(osx/kqueue). It looks like
> > sleep does not work as expected(it blocks forever) inside
> > light thread if some blocking queue is used, see ex:
>
> I think you also need to join thread::Light to ensure they
> finish, otherwise there's nothing to "drive" them.

The following improves scheduling for Kernel#sleep and
Queue#pop/#push by allowing them to "drive" Thread::Light
(and also adds Thread#join as a Thread::Light scheduling point)

https://80x24.org/spew/20181122100334.evcxdcorsjrhqfj2@dcvr/raw

Thanks again Dmitry.

Updated pull request (above change squashed):

The following changes since commit 0bd8193eba5139812c18f779ba5831b3c7df01d7:

ext/socket/init.c (rsock_socket0): non-blocking for non-SOCK_NONBLOCK (2018-11-22 10:13:21 +0000)

are available in the git repository at:

https://80x24.org/ruby.git thread-light-r65925

for you to fetch changes up to 2801f3e4daeec1b21bc138b468a2617c10a3ea6a:

I need sleep myself :<

#158 - 11/22/2018 03:40 PM - dm1try (Dmitry Dedov)

normalperson (Eric Wong) wrote:

> are available in the git repository at:
>
> https://80x24.org/ruby.git thread-light-r65925
>
> for you to fetch changes up to 2801f3e4daeec1b21bc138b468a2617c10a3ea6a:

Nice! Now it works. Moreover, the example above was extracted from a bigger one example for a common web scenario: "dumb" http server which accepts clients requests to the queue in a native thread and then handles each request in a light thread by doing some IO(single query to postgres) in the handler. Before the change async_exec blocked the execution in a light thread btw, it works now. I was not sure if this was the same problem or not so I decided to postpone it)

I like the results but I'm going to do more testing on this weekend, thank you again!

> I need sleep myself :<

wish you your "sleep" will work as expected :)

#159 - 11/28/2018 10:22 AM - matz (Yukihiro Matsumoto)

The first proposal of auto fiber was fibers with implicit context switch on I/O operations. They are fundamentally not fibers, but threads (without time slice context switching). Since they switch context less often, they are less dangerous than normal threads. But now it seems to be changed to user level preemptive threads with time slice switching.

Regarding the fact that I regret adding threads to the language because they are too difficult to use correctly, I don't want to add yet another variation of threads. So the current light-weight thread proposal is not acceptable to me.

It seems that there are two types of concurrency demand. The one for CPU intensive tasks and the other for I/O intensive tasks. To address the former, ko1 (Koichi Sasada) is working on Guilds. And he said the concept of Guild can be applicable to I/O intensive tasks as well. I am not confident about his statement yet. But I'd like to see if Guild works (or not).

If Guild turns out to be unsuitable for I/O intensive tasks, there's a chance for the original auto fiber for I/O intensive tasks.

Recently I talked with some companies using Ruby (on Rails) and found out that the expectation for I/O multiplexing is greater than I thought. If the demand is big enough, it may be better to add auto fiber to the language without waiting for Guilds, because it's simpler and easier to use. We need more investigation.

Matz.

#160 - 11/28/2018 10:52 AM - normalperson (Eric Wong)

matz@ruby-lang.org wrote:

> The first proposal of auto fiber was fibers with implicit context switch on I/O operations. They are fundamentally not fibers, but threads (without time slice context switching). Since they switch context less often, they are less dangerous than normal threads. But now it seems to be changed to user level preemptive threads with time slice switching.

Hi Matz, thanks for your response.

There is no time-slice for switching current Thread::Light.

It only switch when execution context cannot proceed:
(select/wait_for_single_fd, waitpid, sleep, Queue/SizedQueue blocking,

Thread*#join/#value) along with Thread.pass

> Regarding the fact that I regret adding threads to the language because they are too difficult to use correctly, I don't want to add yet another variation of threads. So the current light-weight thread proposal is not acceptable to me.

OK. I'm trying to avoid the same mistakes, so I opted to avoid timeslice switching.

The current design discourages Mutex/ConditionVariable (holding ANY Mutex acts like 1.8 Thread.exclusive) and only switch at well-defined points.

> It seems that there are two types of concurrency demand. The one for CPU intensive tasks and the other for I/O intensive tasks. To address the former, ko1 (Koichi Sasada) is working on Guilds. And he said the concept of Guild can be applicable to I/O intensive tasks as well. I am not confident about his statement yet. But I'd like to see if Guild works (or not).

Right, I am also skeptical as Guild is implemented using native thread.

"I/O intensive" mean different things. Examples:

1. serving large files to clients in fast LAN (10G)
2. serving tens/hundreds of thousands of clients from far off corners of the world with slow, crappy connections

They require different approaches; current native Thread is great for 1. (especially if the local HDD/SSD is a bottleneck).

Thread::Light should be ideal for 2; but can also handle 1 if combined with native Thread / fork.

My goal is to be able to allow the programmer to support both

1. and 2. in the same process.

> If Guild turns out to be unsuitable for I/O intensive tasks, there's a chance for the original auto fiber for I/O intensive tasks.
>
> Recently I talked with some companies using Ruby (on Rails) and found out that the expectation for I/O multiplexing is greater than I thought. If the demand is big enough, it may be better to add auto fiber to the language without waiting for Guilds, because it's simpler and easier to use. We need more investigation.

Really? It seems to me nobody is interested in this feature, so I'll probably abandon it as I can't afford to hack on Ruby much longer.

Note: as you know, I won't make public appearance due to privacy and personal safety concerns, I also won't use proprietary messaging systems; so all I know is what I read in email.

#161 - 11/28/2018 11:12 AM - normalperson (Eric Wong)

Changes in this iteration:

- reduced stack usage in the select()-based implementation (also made some io.c changes today in trunk to cut stack)
- refined auto-switching disable for critical sections

I plan on switching https://public-inbox.org/git/ (my busiest public website) to use a demo server with Thread::Light.

The following changes since commit c80aeb527e855950823f252ff382ea24a03a0c2d:

remove two unnecessary variables (np2 and np3) (2018-11-28 07:07:59 +0000)

are available in the git repository at:

https://80x24.org/ruby.git thread-light-r66072

for you to fetch changes up to 4de667326f5238bcae7dc40dd11d879e6704f1af:

Thread::Light: green threads implemented using fibers (2018-11-28 10:47:19 +0000)

Broken out patches:

https://80x24.org/spew/20181128105225.1211-1-e@80x24.org/raw
https://80x24.org/spew/20181128105225.1211-2-e@80x24.org/raw

#162 - 11/28/2018 12:19 PM - k0kubun (Takashi Kokubun)

> I plan on switching https://public-inbox.org/git/ (my busiest public website) to use a demo server with Thread::Light.

I'm interested in the "demo server" as a real-world use case of Thread::Light. Is the code available somewhere? Also, is there any plan to use Thread::Light on each Unicorn process (hybrid of multi-process and multi-Thread::Light, similar to puma.gem)?

> It seems to me nobody is interested in this feature

I personally expect ioquatix (Samuel Williams) will use (or support as one of concurrency driver) Thread::Light for his falcon.gem, in the context of https://www.codeotaku.com/journal/2018-11/fibers-are-the-right-solution/index.

#163 - 11/28/2018 07:52 PM - normalperson (Eric Wong)

> https://bugs.ruby-lang.org/issues/13618#change-75238

takashikkbn@gmail.com wrote:

> I'm interested in the "demo server" as a real-world use case
> of Thread::Light. Is the code available somewhere?

I haven't written it, yet. I'll update the thread as soon
as I have something. I guess I'll need Timeout support
for Thread::Light, too.

Hopefully this week...

> Also, is there any plan to use Thread::Light on each Unicorn
> process (hybrid of multi-process and multi-Thread::Light,
> similar to puma.gem)?

It would be an insult to existing unicorn users to integrate
features which are completely opposite to the original design.

That said, it could replace nginx for unicorn users by being a
fullly-buffering reverse proxy; but it'll be orthogonal and
interchangeable with nginx.

> I personally expect ioquatix (Samuel Williams) will use (or support as one of concurrency driver) Thread::Light for his falcon.gem, in the context of https://www.codeotaku.com/journal/2018-11/fibers-are-the-right-solution/index.

ioquatix (Samuel Williams) seems happy with libev and "async". I don't believe
libev is the right fit for Ruby or existing Ruby codebases.
I also want to support Queue and SizedQueue.

#164 - 11/28/2018 08:07 PM - ioquatix (Samuel Williams)

k0kubun (Takashi Kokubun) I like the general ideas presented here but the implementation is too heavy/specific for my use case. For example, it won't work on JRuby, TruffleRuby, or other implementations. That's the major benefit of using a gem for the reactor implementation.

Additionally, I need to use Fiber for task switching. Does Thread::Light allow yield/resume?

The tricky part about supporting a different backend is whether it changes semantics of operations. If it does, it's a big problem. If you try to use two different libraries of code that expect different backend semantics, it can cause many headache. That's why I think as an IO reactor, this is really great, but as a high level interface, I'm not convinced I can use it.

I think we need to separate the model for concurrency from the backend selector/reactor implementation, and the front end should provide the appropriate hooks (i.e. wait_readable/wait_writable). Rust take this approach I believe.

normalperson (Eric Wong) libev is okay, it's abstracted by nio4r which is battle tested. It's already proven that it can improve I/O concurrency of existing Ruby code base with no change to code provided you use an async aware application server e.g. falcon.

#165 - 11/29/2018 12:22 AM - normalperson (Eric Wong)

samuel@oriontransfer.net wrote:

> k0kubun (Takashi Kokubun) I like the general ideas presented here but the implementation is too heavy/specific for my use case. For example, it won't work on JRuby, TruffleRuby, or other implementations. That's the major benefit of using a gem for the reactor implementation.

Fiber doesn't really benefit (in terms of memory use) on JRuby, either, right? I don't know much about implementations outside of this one.

> Additionally, I need to use Fiber for task switching. Does Thread::Light allow yield/resume?

We can expose yield/resume easily; but I don't know if there is use case for it... There is already Kernel#sleep support and Thread::Light#run

Maybe a Thread::Light#raise implementation? I don't know if interrupts should be handled via rb_thread_t or rb_execution_context_t (right now, it's mixed and confusing)

> The tricky part about supporting a different backend is whether it changes semantics of operations. If it does, it's a big problem.

What semantic changes are you talking about? I'm doing my best to make things transparent to native Thread users.

The biggest difference is waitpid probably won't benefit on Windows because lack of SIGCHLD (maybe usa (Usaku NAKAMURA) can workaround it with polling or IOCP).

> If you try to use two different libraries of code that expect different backend semantics, it can cause many headache. That's why I think as an IO reactor, this is really great, but as a high level interface, I'm not convinced I can use it.

Really, stop thinking in terms of "reactor". It does not map well to applications requiring many native threads (as I've said this many times, already).

There is no reactor for this and never will be.
Instead, the scheduling mechanism wires into kernel-provided queues (e.g. "kqueue", I've said this before, too).

#166 - 11/29/2018 11:33 AM - normalperson (Eric Wong)

Eric Wong normalperson@yhbt.net wrote:

> samuel@oriontransfer.net wrote:
>
> > k0kubun (Takashi Kokubun) I like the general ideas presented here but the implementation is too heavy/specific for my use case. For example, it won't work on JRuby, TruffleRuby, or other

> > implementations. That's the major benefit of using a gem for
> > the reactor implementation.
>
> Fiber doesn't really benefit (in terms of memory use) on JRuby,
> either, right? I don't know much about implementations outside
> of this one.
>
> > Additionally, I need to use Fiber for task switching. Does
> > Thread::Light allow yield/resume?
>
> We can expose yield/resume easily; but I don't know if there
> is use case for it... There is already Kernel#sleep support
> and Thread::Light#run

So Thread::Light#run wakes up from IO#wait as well; but that's
fixable.

I also noticed I implemented Thread::Light#wakeup as an alias of
Thread::Light#run; because I wanted to avoid extra allocation.
Thread#wakeup doesn't invoke the scheduler, #run does; so I
want ::Light to match that semantic.

#167 - 12/15/2018 12:22 PM - normalperson (Eric Wong)

> https://bugs.ruby-lang.org/issues/13618

Rebased against r66407, fixed fork+GC bug on th->interrupt_lock
and added RDoc for Thread::Light class

https://80x24.org/ruby.git thread-light-r66407
(also, the "thread-light" branch is a moving branch which gets rebased)

for you to fetch changes up to 9e602794916178f43ef51c1a90b0636d42967804

Still wasting my time with memory leaks with fork leapfrogging
[ruby-core:90545] :<

#168 - 01/01/2019 10:58 PM - dm1try (Dmitry Dedov)

- File ruby_2019-01-02-014201_dmp.crash added

- File crush.log added

normalperson (Eric Wong) wrote:

> > https://bugs.ruby-lang.org/issues/13618
>
> Rebased against r66407, fixed fork+GC bug on th->interrupt_lock
> and added RDoc for Thread::Light class
>
> https://80x24.org/ruby.git thread-light-r66407
> (also, the "thread-light" branch is a moving branch which gets rebased)
>
> for you to fetch changes up to 9e602794916178f43ef51c1a90b0636d42967804
>
> Still wasting my time with memory leaks with fork leapfrogging
> [ruby-core:90545] :<

Eric, FYI, something is wrong with native threads on osx in this update(though I'm not sure if it worked before)
code example:

```
# test.rb
require 'socket'

server = TCPServer.new('localhost', 2345)
clients = Queue.new
Thread.start { clients.push(server.accept) }.join
```

see crash report logs for more information

#169 - 01/04/2019 02:52 PM - normalperson (Eric Wong)

me@dmitry.it wrote:

> Eric, FYI, something is wrong with native threads on osx in this update(though I'm not sure if it worked before)
> code example:

I just pushed out some minor fixes to "thread-light" branch
at https://80x24.org/ruby.git 32af044448f29f65cfe15f96333ace95c01d24be

I could not reproduce your issue on FreeBSD 11.2 (no OSX or
non-Free software for me)

Honestly, I'm likely to abandon this feature because I have more
important (non-Ruby) things to work on.

Files

0001-auto-fiber-schedule-for-rb_wait_for_single_fd-and-rb.patch	82.8 KB	06/01/2017	normalperson (Eric Wong)
crush.log	6.64 KB	01/01/2019	dm1try (Dmitry Dedov)
ruby_2019-01-02-014201_dmp.crash	39.8 KB	01/01/2019	dm1try (Dmitry Dedov)