

Ruby - Bug #18482

Fiber can not disable scheduler

01/12/2022 05:48 PM - jakit (Jakit Liang)

Status:	Rejected	Backport: 2.6: UNKNOWN, 2.7: UNKNOWN, 3.0: UNKNOWN, 3.1: UNKNOWN
Priority:	Normal	
Assignee:	ioquatix (Samuel Williams)	
Target version:		
ruby -v:	ruby 3.1.0p0 (2021-12-25 revision fb4df44d16) [arm64-darwin20]	
Description		
class Fiber can not disable scheduler with it's parameter.		
When parameter is false:		
<pre>require 'fiber' require 'io/nonblock' class SimpleScheduler def initialize @readable = {} @writable = {} @waiting = {} @ready = [] @blocking = 0 @urgent = IO.pipe end def run while @readable.any? or @writable.any? or @waiting.any? or @blocking.positive? or @ready.any? readable, writable = IO.select(@readable.keys + [@urgent.first], @writable.keys, [], 0) readable&.each do io if fiber = @readable.delete(io) fiber.resume end end writable&.each do io if fiber = @writable.delete(io) fiber.resume end end @waiting.keys.each do fiber if current_time > @waiting[fiber] @waiting.delete(fiber) fiber.resume end end ready, @ready = @ready, [] ready.each do fiber fiber.resume end end end def io_wait(io, events, timeout) unless (events & IO::READABLE).zero? @readable[io] = Fiber.current end end end</pre>		

```

    unless (events & IO::WRITABLE).zero?
      @writable[io] = Fiber.current
    end

    Fiber.yield
    return events
  end

  def kernel_sleep(duration = nil)
    block(:sleep, duration)
    return true
  end

  def block(blocker, timeout = nil)
    if timeout
      @waiting[Fiber.current] = current_time + timeout
      begin
        Fiber.yield
      ensure
        @waiting.delete(Fiber.current)
      end
    else
      @blocking += 1
      begin
        Fiber.yield
      ensure
        @blocking -= 1
      end
    end
  end

  def unblock(blocker, fiber)
    @ready << fiber
    io = @urgent.last
    io.write_nonblock('.')
  end

  def close
    run
    @urgent.each(&:close)
    @urgent = nil
  end

  private
  def current_time
    Process.clock_gettime(Process::CLOCK_MONOTONIC)
  end

  scheduler = SimpleScheduler.new
  Fiber.set_scheduler(scheduler)

  puts "Go to sleep!"

  f = Fiber.new(false) do
    puts "Going to sleep"
    sleep(1)
    puts "I slept well"
  end

  f.resume

  puts "Wakey-wakey, sleepyhead"

```

Result:

Go to sleep!

```
Going to sleep
Wakey-wakey, sleepyhead
I slept well
```

And when parameter is true:

```
require 'fiber'
require 'io/nonblock'

class SimpleScheduler
  def initialize
    @readable = {}
    @writable = {}
    @waiting = {}
    @ready = []
    @blocking = 0
    @urgent = IO.pipe
  end

  def run
    while @readable.any? or @writable.any? or @waiting.any? or @blocking.positive? or @ready.any?
      readable, writable = IO.select(@readable.keys + [@urgent.first], @writable.keys, [], 0)

      readable&.each do |io|
        if fiber = @readable.delete(io)
          fiber.resume
        end
      end

      writable&.each do |io|
        if fiber = @writable.delete(io)
          fiber.resume
        end
      end

      @waiting.keys.each do |fiber|
        if current_time > @waiting[fiber]
          @waiting.delete(fiber)
          fiber.resume
        end
      end

      ready, @ready = @ready, []
      ready.each do |fiber|
        fiber.resume
      end
    end

    def io_wait(io, events, timeout)
      unless (events & IO::READABLE).zero?
        @readable[io] = Fiber.current
      end
      unless (events & IO::WRITABLE).zero?
        @writable[io] = Fiber.current
      end

      Fiber.yield
      return events
    end

    def kernel_sleep(duration = nil)
      block(:sleep, duration)
      return true
    end

    def block(blocker, timeout = nil)
```

```

    if timeout
      @waiting[Fiber.current] = current_time + timeout
    begin
      Fiber.yield
    ensure
      @waiting.delete(Fiber.current)
    end
  else
    @blocking += 1
  begin
    Fiber.yield
  ensure
    @blocking -= 1
  end
end
end

def unblock(blocker, fiber)
  @ready << fiber
  io = @urgent.last
  io.write_nonblock('.')
end

def close
  run
  @urgent.each(&:close)
  @urgent = nil
end

private
def current_time
  Process.clock_gettime(Process::CLOCK_MONOTONIC)
end
end

scheduler = SimpleScheduler.new
Fiber.set_scheduler(scheduler)

puts "Go to sleep!"

f = Fiber.new(true) do
  puts "Going to sleep"
  sleep(1)
  puts "I slept well"
end

f.resume

puts "Wakey-wakey, sleepyhead"

```

Result (was still the same):

```

Go to sleep!
Going to sleep
Wakey-wakey, sleepyhead
I slept well

```

While make the set_scheduler line commented:

```

scheduler = SimpleScheduler.new
# Fiber.set_scheduler(scheduler) // Here is commented

puts "Go to sleep!"

f = Fiber.new(false) do
  puts "Going to sleep"
  sleep(1)

```

```
puts "I slept well"
end
```

Result is right:

```
Go to sleep!
Going to sleep
I slept well
Wakey-wakey, sleepyhead
```

Maybe in some situation.

I wrote my gem without Scheduler. But user defined its Scheduler for his or her logic code.

It will break the sequence of Fiber which was needed for my gem.

Also, using Fiber in the Enumerator situation will be broke down too:

```
db.with_each_row_of_result(sql_stmt) do |row|
  yield row
end
```

[[<https://blog.appsignal.com/2018/11/27/ruby-magic-fibers-and-enumerators-in-ruby.html>]]

It will break the sequence of db rows when doing enum such like python's generator.

=====

Also, another question is that I saw something was talk in:

<https://bugs.ruby-lang.org/issues/16786>

I think there would be a better way to improve this.

You can see, in c++, std::thread is easy to create and join a new thread.

If someone make a std::thread::scheduler into STL of C++.

And let user to implement its std::thread::handler to implement the virtual methods (interface or callback) to use it.

And std::thread::scheduler holds an independent thread pool which is not separated.

What do you think about this std::thread::scheduler?

What about make a golang's GMP into std::thread or std::coroutine.

Why not STL do that?

Why not STL let std::thread become a self-scheduled module?

Otherwise, the sense of implement Scheduler as async await may be a good idea, but there has module named Ractor can solve it.

Maybe:

```
IO.async do |readable, writeble|
  if readable
    # code
  end
end
```

=====

In other programming language, like Python.

Python never let it's Generator mixed with async IO but add async syntax:

```
async def coro():      # a coroutine function
```

```
await smth()
```

```
async def asyncgen(): # an asynchronous generator function
    await smth()
    yield 42
```

Python goes in a right way. Methods can run async and something like `IO.write()` can put in it.

```
async def async_write(data): # a coroutine function
    IO.write(data)
```

```
async def send(message): # an asynchronous generator function
    await async_write("hello " + message)
    yield 1
```

<https://www.python.org/dev/peps/pep-0525/#id10>

Also, JavaScript use async syntax to identify the async procedure.

Above that, I think the Fiber.scheduler may not be a good idea. Because Ractor is here.

Ractor can run methods async and we can put `IO.write` in it and make it "async".

History

#1 - 01/12/2022 05:48 PM - jakit (Jakit Liang)

- *ruby -v set to 3.1.0*

#2 - 01/12/2022 05:48 PM - jakit (Jakit Liang)

- *ruby -v changed from 3.1.0 to ruby 3.1.0p0 (2021-12-25 revision fb4df44d16) [arm64-darwin20]*

#3 - 01/12/2022 05:52 PM - jakit (Jakit Liang)

- *Description updated*

#4 - 01/12/2022 05:53 PM - mame (Yusuke Endoh)

- *Assignee set to ioquatix (Samuel Williams)*

#5 - 01/12/2022 06:01 PM - jakit (Jakit Liang)

- *Description updated*

#6 - 01/12/2022 06:06 PM - jakit (Jakit Liang)

- *Description updated*

#7 - 01/12/2022 07:32 PM - jakit (Jakit Liang)

- *Description updated*

#8 - 01/12/2022 07:37 PM - jakit (Jakit Liang)

- *Description updated*

#9 - 01/12/2022 08:56 PM - ioquatix (Samuel Williams)

- *Status changed from Open to Rejected*

Did you read the documentation?

<https://rubyapi.org/3.1/o/fiber#method-c-new>