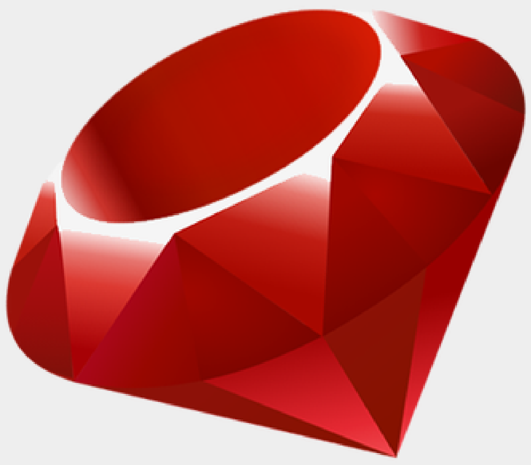
Pre-emptive fiber-based concurrency in MRI Ruby

Posted on February 7, 2021 by wjwh



Ruby has recently released version 3.0.0, which includes the `FiberScheduler` interface. I have written about this before and I think it has a lot of potential because asynchronous I/O with event loops is pretty great. If you send off a query to an upstream API and it will take a few dozen milliseconds to respond, you can do a lot of work in the intervening time. Especially for web app backends, which usually spend a lot of time waiting for the results of database queries and/or various API calls, asynchronous I/O can provide huge throughput improvements. One downside of the current implementation is that it exclusively uses cooperative multitasking; every `Fiber` needs to explicitly call `Fiber.yield` at some appropriate moment to allow other `Fiber`s to run. However, if a CPU-intensive tasks has not been written with `Fiber`s in mind, they can "hog the event loop" and prevent other tasks from running in time. This can cause excessive latency for the other tasks. In this article I'll go over a few ways to have a `Fiber` automatically call `yield` if it takes "too much" time.

Nonblocking (?) `Fiber`s

To demonstrate the problem with CPU intensive tasks in a `Fiber`, consider the following short program:

```
require 'fiber'
require 'async'

# a deliberately inefficient function to use some CPU time [1]
# on my machine, fib(40) takes about 7.5 seconds
def fib(n)
  if n <= 1
    1
  else
    (fib(n-1) + fib(n-2))
  end
end

def sleeper
  Async {
    start = Time.now
    sleep 1
    puts "Tried sleeping for one second, actual time: #{Time.now - start}"
  }
end

def big_calculation
  Async {
    x = fib 40
    puts "fib(40) = #{x}"
  }
end

Async { |task|
  sleeper
  sleeper
  big_calculation
}
```

This will output the following:

```
fib(40) = 165580141
Tried sleeping for one second, actual time: 7.59390655
Tried sleeping for one second, actual time: 7.593904645
```

As you can see, after starting two `Fiber`s that sleep for a second, the Fibonacci calculation kicks off and takes almost three seconds. Because the `fib` function never calls `Fiber.yield`, the `Async` scheduler does not get a chance to resume the sleeping `Fiber`s until the `fib` `Fiber` is done. The sleeping `Fiber`s will effectively sleep WAY longer than intended. In a real application, those sleeps may have been part of a timeout for an API call and overshooting could cause some seriously unwanted behavior.

BTW: This program uses the `async` framework because I happen to know it well, but anything built upon MRI `Fiber`s will have the same type of behavior.

Various ways to automatically yield

The problem of tasks not yielding in time (or at all) in cooperative multitasking systems is well known and has been studied extensively. The usual solution is to augment the runtime with some way of forcing tasks to yield. This solution is known as pre-emptive multitasking. For example, Erlang uses “reductions” (roughly equivalent to function calls) to determine when a thread has used enough CPU time and needs to be switched out; Haskell code uses a recurring timer that provides a trigger to switch and there are many more methods.

Ruby code runs in a Virtual Machine (VM) known as YARV, which is both an advantage and a disadvantage: on the downside we need to take extra care not to break internal VM data structures, but on the plus side most VMs have many more inbuilt possibilities for altering running code on the fly than compiled binaries have. MRI Ruby has several ways to alter existing code on the fly. One such way is using the `Tracepoint` API, which allows you (amongst other things) to run arbitrary code before every method call. My first idea was to implement an Erlang-type scheme where before every method call we decrement a global variable and call `Fiber.yield` if it becomes zero. Coding this up only took a few minutes and it did work, but sadly it had a *huge* overhead. The `fib` call above took over five times as long compared to the function without the `Tracepoint` hook. I didn’t profile too in-depth but I suspect that this was because a block passed as a `Tracepoint` hook needs to be Ruby code. Even if it calls into C immediately, it incurs substantial overhead. Luckily, some digging through the MRI source code revealed that it was also possible to pass in a C function pointer directly with `rb_add_event_hook`. You can call it like `rb_add_event_hook(reduce_reduction_counter, RUBY_EVENT_CALL | RUBY_EVENT_C_CALL, 0);`, where `reduce_reduction_counter` is a function taking no arguments and returning `void`. This function will then be called on every method call, whether the function is implemented in C or in Ruby. As I understand it, `reduce_reduction_counter` would NOT be called for internal C functions, only those called directly from Ruby.

Staying in C land the whole time did provide a substantial speedup, but it was still significantly slower than without the trace function. In some extremely unscientific benchmarking, code with the C event hook still took almost twice as long to complete as code without the event hook. This makes sense, since it would need to call at least double the amount of functions (the original method call and also the event hook). It was possible to only incur this cost for certain `Fiber`s (i.e., if you knew beforehand when it would be needed) but I was still not very happy with it.

Timer-based yielding

Another way of pre-empting arbitrary code is by requesting the operating system to do the interrupting for us. Ruby contains a `Signal` module that lets us trap Posix signals and run arbitrary code in response. With the `timer_create` and `timer_settime` syscalls it is pretty straightforward to request for an arbitrary signal to be delivered to our process after some (short) time has elapsed. This method is used in (amongst others) the runtime of the Haskell language [2] to manage thread switching. It is easy to set this up in MRI with the following C code:

```
#include "ruby.h"
#include "extconf.h"
#include <signal.h>
#include <time.h>

timer_t fiber_timer;
struct itimerspec its_50ms;

VALUE rb_arm_fiber_timer() {
  timer_settime(fiber_timer, 0, &its_50ms, NULL);
  return Qnil;
}

void Init_yieldingfiber()
{
  VALUE fiberClass = rb_const_get(rb_cObject, rb_intern("Fiber"));
  VALUE yieldingFiberClass = rb_define_class("YieldingFiber", fiberClass);
  rb_define_singleton_method(yieldingFiberClass,
    "arm_fiber_timer!",
    rb_arm_fiber_timer,
    0);

  // When the timer expires, send us SIGUSR2
  struct sigevent sev;
  sev.sigev_notify = SIGEV_SIGNAL;
  sev.sigev_signo = SIGUSR2;
  sev.sigev_value.sival_ptr = &fiber_timer;

  timer_create(CLOCK_MONOTONIC, &sev, &fiber_timer);
  // oneshot timer set to expire in 50 ms
  its_50ms.it_value.tv_sec = 0;
  its_50ms.it_value.tv_nsec = 50000000;
  // no interval plz
```

```
    its_50ms.it_interval.tv_sec = 0;
    its_50ms.it_interval.tv_nsec = 0;
  }
```

We define a subclass of `Fiber` (creatively named `YieldingFiber`) and define a singleton method `arm_fiber_timer!` on it which calls `timer_settime()`. Now whenever we call `YieldingFiber.arm_fiber_timer!` in Ruby code, approximately 50 milliseconds later the process will receive a `SIGUSR2` signal. The default handler for this signal will just end the program, so we need to install our own handler first. Additionally, we want to make sure that, if the signal arrives *after* the `Fiber` has yielded by itself (for example, because the computation took less than 50 ms), the operation of the `Fiber` scheduler is not adversely affected by the signal. We can do this by checking if the current `Fiber` is actually such a `YieldingFiber` and only `yield`ing if it is. (This assumes that the scheduler does not run in a `YieldingFiber`.)

```
class YieldingFiber < Fiber
  def resume(**args)
    YieldingFiber.arm_fiber_timer!
    super(**args)
  end
end

Signal.trap("USR2") {
  if Fiber.current.is_a? YieldingFiber
    Fiber.yield
  end
}
```

Whenever a `YieldingFiber` is `resume`d, it will arm the signal timer first. If the `YieldingFiber` takes longer than 50 ms until it either finishes or `yield`s, the signal will interrupt it and the signal handler will call `Fiber.yield`. Ruby signal handlers are safe for the VM, the block will only ever be called when the VM reaches a safe point for it to run. If the `YieldingFiber` takes less than 50 ms, there are two options: by the time the signal arrives, either the program has progressed to another `YieldingFiber` or it has not. If it hasn't, `Fiber.current.is_a? YieldingFiber` will return `false` and the signal handler will be a no-op. If it has, the `resume` operation will have called `YieldingFiber.arm_fiber_timer!` and the signal will have been reset. The semantics of `Fiber` are slightly peculiar in that `Fiber#resume` will actually change the result of `Fiber.current`, but the `resume` methods of subclasses of `Fiber` do not change that result until they call `super`. This means that if the signal is triggered before or during the call to `YieldingFiber.arm_fiber_timer!`, the current `Fiber` will not yet be a `YieldingFiber` and nothing will happen. If the timer would have been triggered after `YieldingFiber.arm_fiber_timer!`, it will have been reset and now the timer won't expire for another 50 ms. Depending on operating system implementation, there *might* be a race condition where the timer is triggered but the signal does not get handled by Ruby until after the timer is reset *and* the `super` call to `Fiber#resume` is completed. In that case, the `YieldingFiber` will yield much faster than after 50 ms. I believe this to be a very unlikely scenario and in any case the `YieldingFiber` will be scheduled in again as usual next time the scheduler loops around. [3]

There is one thing left to do to make our original program work as desired: the `async` framework does not work with a Ruby `Fiber` but with its own `Task` abstraction. Simply calling `Fiber.yield` from a `YieldingFiber` will return to the scheduler. This means that we need to improve our `YieldingFiber` class so that it can play nice with the `Reactor` class from `async`:

```
class YieldingFiber < Fiber
  def initialize(reactor, &block)
    @reactor = reactor
    super(&block)
  end

  def resume(**args)
    YieldingFiber.arm_fiber_timer!
    super(**args)
  end

  def yield_to_scheduler
    @reactor << self
    Fiber.yield
  end
end

Signal.trap("USR2") {
  if Fiber.current.is_a? YieldingFiber
    Fiber.current.yield_to_scheduler
  end
}
```

The changes are very superficial: when constructing a `YieldingFiber` we also pass in a reference to the reactor and when yielding back to the scheduler we first add our `YieldingFiber` back to the run queue of the reactor so that it will be scheduled again “soon”. This works because in the async framework, reactors allow anything that “looks like” a `Fiber` in a duck typing sense. Now we can run our original program with the `fib` calculation running in a `YieldingFiber` and it will work much better:

```
require 'fiber'
require 'async'
require 'yieldingfiber'

# a deliberately inefficient function to use some CPU time
# on my machine, fib(40) takes about 7.5 seconds
def fib(n)
  if n <= 1
    1
  else
    (fib(n-1) + fib(n-2))
  end
end

def sleeper
  Async {
    start = Time.now
    sleep 1
    puts "Tried sleeping for one second, actual time: #{Time.now - start}"
  }
end
```

```
def big_calculation(reactor)
  yf = YieldingFiber.new(reactor) {
    x = fib 38
    puts "fib(38) = #{x}"
  }
  reactor << yf
end

Async { |task|
  sleeper
  sleeper
  big_calculation(task.reactor)
}
```

This outputs the following:

```
Tried sleeping for one second, actual time: 1.000511959
Tried sleeping for one second, actual time: 1.000553547
fib(38) = 63245986
```

As you can see, the sleeping `Fiber`s now no longer have to wait for the big `fib` calculation and report sleep times very close to the intended one second. There is still some overhead because the scheduler gets invoked much more often, but based on my measurements the total runtime of the program increased only about 4% compared to the original version. Depending on your workload, this might be an acceptable tradeoff for gaining much better latency in timeout- and I/O-bound `Fiber`s.

Drawbacks of auto-yielding

The code I presented is still very brittle and has a number of drawbacks:

- Yielding every 50 ms introduces about 4% runtime overhead (in this almost worst-case test) compared to the fully cooperative multitasking version. This is mostly the extra time spent in the scheduler, the timer-setting system calls are only about 250 ns each and so almost imperceptable compared to the 50 ms a `YieldingFiber` could take.
- It only works on Linux and under MRI due to the timer system calls used. Other operating systems and Ruby implementations provide similar abstractions but it would require some work to truly make it portable.
- This solution uses a lot of Ruby-level calls like `Fiber.current.is_a? YieldingFiber`. This helps with readability but increases overhead. These calls could be made into C calls for additional speed.
- Using `SIGUSR2` may conflict with other gems. Haskell uses `SIGALRM`, which is less likely to be already taken. It is possible to dynamically alter the signal number used, so this should not be a big problem but it does need work.
- The Ruby VM will only handle signals at certain points in its operation. Code in C extensions is almost certainly not hitting those points, so if you have gems that do long calculations in pure C the deadline for a `YieldingFiber` may be violated quite a bit. This can also be a problem for FFI calls in Haskell and Erlang, so it is unclear how much of an issue this really is.
- While the solution can be extended to multiple threads (and one scheduler per thread), a Ruby `Fiber` is bound to its original thread. If you have a program that does more work than a single core can handle you might get unlucky and get all your `YieldingFibers` on the same thread. That thread will

be very busy while other threads might be idle. “Work stealing” as in Haskell or Go is the usual solution, but transferring `Fiber`s between threads is not possible in Ruby (yet). [4]

All this can be fixed with enough work, I just wanted to make clear that this is not anywhere near production ready.

Conclusion

In this blog post I went over a few methods to allow for pre-emptive scheduling of `Fiber`s in MRI Ruby. It turns out that with OS-supplied interrupts, it is fairly straightforward to accomplish, and the overhead is pretty minimal.

I’m a big fan of lightweight thread systems and think that (in 2021) for almost all modern server programs, an event loop per thread with pre-emption and inter-scheduler work stealing is close to the optimal architecture [5] for the language runtime. It provides high concurrency with much less overhead than OS level threads, while still allowing the programmer to code in an easy “straight-line” fashion without callback hell and the like. The `FiberScheduler` interface provides a decent step towards this architecture for Ruby programs, but the cooperative multithreading model of `Fiber`s has severe downsides. If they could (optionally) become pre-emptible that would make it a lot more feasible to use `Fiber`s everywhere.

[1] This Fibonacci implementation was used because it does an O(2^N) amount of function calls, which is basically the worst case for the tracing implementation. It is also easy to tweak its required running time by changing the argument.

[2] Technically, Haskell only uses interval timers for its single-threaded runtime which multiplexes multiple Haskell threads onto a single operating system thread. There is also a multithreaded runtime for M:N threading but that one uses a timerfd.

[3] If you have such strict timing requirements around your code that you can’t handle a few dozen ms delay, Ruby with an async scheduling framework is a pretty poor fit for your usecase anyway, so this scenario shouldn’t really matter in practice.

[4] In this reddit post the designer of the `async` framework /u/ioquatix makes the point that this might not actually be a problem in practice.

[5] I also expect “real” asynchronous I/O to gain a lot of popularity in the coming few years through `io_uring` on Linux and `IOCP` on Windows.