Ruby master - Feature #14739

Improve fiber yield/resume performance

05/05/2018 12:59 AM - ioquatix (Samuel Williams)

Status: Closed
Priority: Normal
Assignee: ioquatix (Samuel Williams)
Target version: 2.6

Description

I am interested to improve Fiber yield/resume performance.

I've used this library before: [http://software.schmorp.de/pkg/libcoro.html](http://software.schmorp.de/pkg/libcoro.html) and handled millions of HTTP requests using it.

I'd suggest to use that library.

As this is used in many places in Ruby (e.g. enumerable) it could be a big performance win across the board.

Here is a nice summary of what was done for RethinkDB: [https://rethinkdb.com/blog/making-coroutines-fast/](https://rethinkdb.com/blog/making-coroutines-fast/)

Does Ruby currently reuse stacks? This is also a big performance win if it's not being done already.

History

#1 - 05/05/2018 01:25 AM - shyouhei (Shyouhei Urabe)
ioquatix (Samuel Williams) wrote:

Does Ruby currently reuse stacks?

Yes.

Not sure how fast libcoro is, though.

#2 - 05/05/2018 02:48 AM - ioquatix (Samuel Williams)

Here is the code [https://github.com/ioquatix/ruby/tree/fiber-libcoro](https://github.com/ioquatix/ruby/tree/fiber-libcoro)

UPDATE: I provided some benchmark details, but it turns out they were wrong. I've retracted it until I can provide correct information to prevent any confusion.

#3 - 05/05/2018 02:50 AM - ioquatix (Samuel Williams)

Not sure how fast libcoro is, though.

In my experience, the libcoro ASM implementation is the fastest implementation I found.

It's not much slower than a (normal) C function call.

#4 - 05/05/2018 03:38 AM - ioquatix (Samuel Williams)

```ruby
# Without libcoro
koyoko% ./build/bin/ruby ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers: 0.099961
execution time for 1000 messages: 19.505909

# With libcoro
koyoko% ./build/bin/ruby ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers: 0.099268
execution time for 1000 messages: 8.491746
```

It's about 2.2x faster.

That's about what I was expecting.
Can someone else confirm? Thanks.

#5 - 05/05/2018 03:54 AM - ioquatix (Samuel Williams)
Without libcoro (macOS)

```
^_^ > ./build/bin/ruby ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers:  0.298039
execution time for 1000 messages: 35.248941
```

With libcoro (macOS)

```
^_^ > ./build/bin/ruby ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers:  0.167117
execution time for 1000 messages: 15.460046
```

On macOS, it's about the same, 2.2x faster.

#6 - 05/05/2018 03:56 AM - ioquatix (Samuel Williams)
I don't know how to run a full benchmark of Ruby. Can someone help me with that? It would be interesting to get a more general idea of the performance.

#7 - 05/05/2018 08:41 AM - vo.x (Vit Ondruch)
I wonder what architectures libcoro supports? It seems it supports x86 a probably some ARM, but what about s390x and ppc64?

#8 - 05/05/2018 10:13 AM - nobu (Nobuyoshi Nakada)
And seems it requires gcc (variants) and non-Windows. coro.c can't compile with Visual C nor mingw gcc. Also, asm needed to be replaced with __asm__ to compile with Apple clang, and it is 3% faster.

```
$ ruby fiber_benchmark.rb 10000 1000
setup time for 10000 fibers:  0.227721
execution time for 1000 messages: 74.540142
```

```
$ ./ruby fiber_benchmark.rb 10000 1000
setup time for 10000 fibers:  0.293740
execution time for 1000 messages: 72.180107
```

https://github.com/nobu/ruby/tree/feature/fiber-libcoro

#9 - 05/05/2018 10:14 AM - ioquatix (Samuel Williams)
You can see the supported methods here.

https://github.com/ioquatix/ruby/blob/4a9c12d94aae1cf3a52ca65f026432cd03e9817bc/libcoro/coro.h#L99-L164

For the proof of concept, I forced it to use the ASM method, which supports 32-bit and 64-bit x86 CPUs and ARM (I've never tested it).

It would make sense to set up some configure tests to detect which one is available.

I'd also suggest if we move forward with this, we should remove most of the native implementation of coroutines in Ruby because they are slower and clutter up the implementation.

#10 - 05/05/2018 10:18 AM - ioquatix (Samuel Williams)
I've compiled this on both LLVM and GCC just fine.

I've never tried compiling it on Windows but it should work. It might require some work.

Also, asm needed to be replaced with __asm__ to compile with Apple clang

I didn't have this problem. What version of the developer tools are you using?

and it is 3% faster.

If you get that, something is wrong, it's definitely a much bigger improvement than that. Did you try it on Linux?

#11 - 05/05/2018 10:48 AM - ioquatix (Samuel Williams)
I am trying out your branch, and will report back. 3% is within the margin for error so it sounds like nothing changed for some reason. There will be
some explanation.

#12 - 05/05/2018 12:00 PM - nobu (Nobuyoshi Nakada)
ioutilix (Samuel Williams) wrote:

> Also, asm needed to be replaced with __asm__ to compile with Apple clang

> I didn't have this problem. What version of the developer tools are you using?

$ clang --version
Apple LLVM version 8.0.0 (clang-800.0.42.1)
Target: x86_64-apple-darwin15.6.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin

If you get that, something is wrong, it's definitely a much bigger improvement than that. Did you try it on Linux?

On Ubuntu 18.04, it has the effect with gcc version 7.3.0 (Ubuntu 7.3.0-16ubuntu3).

trunk

$ .x86_64-linux/exe/ruby src/fiber_benchmark.rb 10000 1000
setup time for 10000 fibers: 0.153903
execution time for 1000 messages: 25.395488

fiber-libcoro

$ make -C x86_64-linux prog > /dev/null && ./x86_64-linux/exe/ruby src/fiber_benchmark.rb 10000 1000
In file included from ../src/libcoro/coro.c:41:0,
    from ../src/cont.c:51:
../src/cont.c: In function ‘cont_free’:
../src/libcoro/coro.h:401:28: warning: statement with no effect [-Wunused-value]
    #define coro_destroy(ctx) (void *)(ctx)
          ^~~~~~~~~~~~~
../src/cont.c:37012: note: in expansion of macro ‘coro_destroy’
coro_destroy((coro_context *)&fib->context);
~~~~~~~~~~~
../src/cont.c: In function ‘fiberInitializeMachineStackContext’:
../src/libcoro/coro.c:331:1: warning: expected ‘coro_func (aka void (*)(void *))’ but argument is of type ‘__attribute__((noreturn)) void (*)(void)’
coro_create((coro_context *ctx, coro_func coro, void *arg, void *sptr, size_t ssize)
~~~~~~~~~~~
In file included from ../src/libcoro/coro.c:51:0:
../src/libcoro/libcoro.c:331:1: note: expected ‘coro_func (aka void (*)(void *))’ but argument is of type ‘__attribute__((noreturn)) void (*)(void)’
coro_create (coro_context *ctx, coro_func coro, void *arg, void *sptr, size_t ssize)
~~~~~~~~~~~
In file included from ../src/libcoro/coro.c:41:0,
    from ../src/cont.c:51:
../src/cont.c: In function ‘rb_fiber_terminate’:
../src/libcoro/coro.h:401:28: warning: statement with no effect [-Wunused-value]
    #define coro_destroy(ctx) (void *)(ctx)
          ^~~~~~~~~~~~~
../src/cont.c:1799:5: note: in expansion of macro ‘coro_destroy’
coro_destroy(&fib->context);
~~~~~~~~~~~
../src/cont.c: At top level:
ccl: warning: unrecognized command line option ‘-Wno-self-assign’
ccl: warning: unrecognized command line option ‘-Wno-constant-logical-and-operator’
ccl: warning: unrecognized command line option ‘-Wno-constant-logical-or-operator’
setup time for 10000 fibers: 0.146823
execution time for 1000 messages: 7.855211

#13 - 05/05/2018 12:03 PM - ioquatix (Samuel Williams)

Yes, that supports my own test as well.

koyoko% ruby --version
ruby 2.5.0p0 (2017-12-25 revision 61468) [x86_64-linux]
koyoko% ruby ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers: 0.094309

08/24/2022
execution time for 1000 messages: 22.248827

koyoko% ./build/bin/ruby --version
ruby 2.6.0dev (2018-05-03 fiber-libcoro 63333) [x86_64-linux]
last_commit=Use libcoro for Fiber implementation to improve performance.
koyoko% ./build/bin/ruby ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers: 0.104364
execution time for 1000 messages: 19.717851

koyoko% ./build/bin/ruby --version
ruby 2.6.0dev (2018-05-03 fiber-libcoro 63333) [x86_64-linux]
last_commit=Use libcoro for Fiber implementation to improve performance.
koyoko% ./build/bin/ruby ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers: 0.104798
execution time for 1000 messages: 8.988672

However, on macOS, I can't reproduce my original results. I apologise. I was playing around with stack allocation. I tried to revert back to that state, but couldn't reproduce the results I gave earlier.

I will continue to investigate.

#14 - 05/05/2018 12:17 PM - ioquatix (Samuel Williams)
Okay, I found out what happened.

On macOS, you need to set

#include "libcoro/coro.c"
#define FIBER_USE_NATIVE 1

Otherwise it won't take the optimal code path. My apologies, I think as I was playing with the code I made that change but didn't commit it after I started patching it to work on Linux, since it seems on Linux that's the default.

Here is the performance improvement.

"_^_" > ./build/bin/ruby ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers: 0.165381
execution time for 1000 messages: 14.267517

"_^_" > ./build/bin/ruby ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers: 0.160629
execution time for 1000 messages: 6.307580

So, it's similar speed-up.

I tried to compile without libcoro, but with #define FIBER_USE_NATIVE 1, but it fails because swapcontext/makecontext is deprecated on macOS and compile fails.

#15 - 05/05/2018 12:30 PM - ioquatix (Samuel Williams)
I updated my branch with a few changes.

I'm sorry I didn't rebase on your branch.

I think once we decide if this is a good idea or not, we can decide how best to integrate it with Ruby. I just wanted to make a proof of concept to show it was a good improvement to performance.

My suggestion would be to remove the implementations from cont.c and update libcoro to support all required platforms. The API provided by libcoro is really great and a nice wrapper.

It should be possible to build libcoro on Windows. I do have Windows with Visual Studio set up but I really have no idea how to use it :) However, it wouldn't be silly to update libcoro to make it compile without problems on all supported platforms. It's quite an "old" implementation, but it does work really well. There are some other implementations available too, some are more modern, but I found this one was pretty good.

It might make sense to fork libcoro into a separate repo, I don't mind maintaining it, I already have a fork of it actually, and it's a bit different from the one here. But, it would make sense to update it a bit.

#16 - 05/05/2018 12:55 PM - ioquatix (Samuel Williams)
I was reading https://sourceware.org/ml/libc-help/2016-01/msg00008.html and noticed the following regarding *context functions:

these functions are deprecated/dead -- they no longer exist in the latest POSIX specification. the preference would be to stop using them. i think we might consider dropping them in a future glibc version.
Of course they still exist, but yes they are deprecated, and non-existent in the latest POSIX standard. I might even remove it from my fork of libcoro.

#17 - 05/05/2018 08:28 PM - shevegen (Robert A. Heiler)

However, it wouldn't be silly to update libcoro to make it compile without problems on all supported platforms.

I can't speak for matz and the ruby core team, but in the past there were (feature-)proposals that were rejected since they were only specific for e. g. Linux - that is, improvements pertaining to Linux, but not other OS. I think matz wants to have ruby be as OS-agnostic as possible; in other words to work on as many OS as possible, too. And there are quite some people who use ruby on windows as well, for one reason or another.

As for benchmarks, I think any noticable improvement is a win and may fit into the "ruby 3 is 3x as fast as ruby 2.0", but to get to that, it may be more important to verify that the improvements could also work on windows. Even 3% would be considerable. :)

By the way, I think there are some ruby-devs who use windows too ... greg I think. May take a little before the issue here is seen by them; they could probably help. (I use linux myself so I won't be of much help.)

#18 - 05/05/2018 10:21 PM - ioquatix (Samuel Williams)

The windows code path for fibers is relatively trivial both in libcoro and cont.c, so I wouldn't be too concerned about windows support. It shouldn't be much effort to make it work well in libcoro or keep existing windows code path.

Thanks for your concern and support, and I hope we can get some traction with this improvement.

I use fibers a lot (https://github.com/socketry/async; is a fiber [stackful coroutine] based concurrency library). My next step is to benchmark the improvement. It obviously won't be anywhere near 2.2x for real code, but I think it should at least be noticeable.

#19 - 05/06/2018 09:48 AM - shyouhei (Shyouhei Urabe)

I'm neutral. This is a feature request but the "feature" being discussed is the speed of execution. It is by nature different from each other. If this improvement could be truly transparent (and seems currently it is), I think there are chances for acceptance. Wider support for different OSes is definitely nice-to-have of course.

#20 - 05/06/2018 10:09 AM - ioquatix (Samuel Williams)

Thanks for your feedback. When I made this issue, I could only select "Bug", "Feature" or "Misc". Should I have selected "Misc" instead?

#21 - 05/06/2018 11:06 AM - ioquatix (Samuel Williams)

I test in some real world applications today. The first is async, which has a performance test for read context switch overhead: https://github.com/socketry/async/blob/master/spec/async/performance_spec.rb

This isn't direct comparison since I'm using rvm with ruby head and my branch, but it's pretty close.

Without libcoro fibers

Async::Wrapper
Warming up --------------------------------------
Wrapper#wait_readable
  1.801k i/100ms
Reactor#register
  2.087k i/100ms
Calculating -------------------------------------
Wrapper#wait_readable
  176.789k (± 5.7%) i/s - 880.689k in 5.004582s
Reactor#register
  227.882k (± 2.9%) i/s - 1.140M in 5.004740s

Comparison:
  Reactor#register: 227882.2 i/s
  Wrapper#wait_readable: 176789.3 i/s - 1.29x slower

With libcoro fibers (12% more context switch for read operations)
Async::Wrapper
Warming up --------------------------------------
Wrapper#wait_readable  2.217k i/100ms
Reactor#register        2.380k i/100ms
Calculating ---------------
Wrapper#wait_readable  197.116k (± 2.7%) i/s - 986.565k in 5.008582s
Reactor#register        256.078k (± 4.4%) i/s - 1.278M in 5.003710s

Comparison:
    Reactor#register:  256077.8 i/s
    Wrapper#wait_readable: 197115.9 i/s - 1.30x slower
#22 - 05/06/2018 11:17 AM - ioquatix (Samuel Williams)

Compare async-dns with bind9 for the same workload:

# Without libcoro-fiber
Async::DNS::Server         user  system  total     real
simple response
Running 2m test @ http://127.0.0.1:9292/
8 threads and 8 connections
Thread Stats                Avg     Stdev    Max     +/- Stdev
Latency         110.06us 647.25us 67.72ms  99.33%
Req/Sec          12.58k  3.07k  26.94k    70.77%
12021990 requests in 2.00m, 401.28MB read
Requests/sec: 100100.72
Transfer/sec: 3.34MB

# With libcoro-fiber (no significant difference)
Async::DNS::Server         user  system  total     real
simple response
Running 2m test @ http://127.0.0.1:9292/
8 threads and 8 connections
Thread Stats                Avg     Stdev    Max     +/- Stdev
Latency         106.47us 834.32us 99.45ms  99.46%
Req/Sec          12.66k  2.95k  17.61k    71.12%
12093398 requests in 2.00m, 403.66MB read
Requests/sec: 100694.76
Transfer/sec: 3.36MB

This one was a toss-up, I'd say there was no significant difference.

#23 - 05/06/2018 12:17 PM - ioquatix (Samuel Williams)

I tested async-http, a web server, it has a basic performance spec using wrk as the client.

I ran it several times and report the best result of each below. It's difficult to make a judgement. I'd like to say performance was improved but if so, < 5%. However, this benchmark is testing an entire web server stack. Context switching only happens a few times per request.. If I had to take a guess, maybe not more than 4 times (accept, read request, write response). In many cases, we only context switch if the operation would block which is unlikely for small request/response on loopback interface.

# Without libcoro-fiber
Async::HTTP::Server
simple response
Running 2m test @ http://127.0.0.1:9292/
8 threads and 8 connections
Thread Stats                Avg     Stdev    Max     +/- Stdev
Latency         110.06us 647.25us 67.72ms  99.33%
Req/Sec          12.58k  3.07k  26.94k    70.77%
12021990 requests in 2.00m, 401.28MB read
Requests/sec: 100100.72
Transfer/sec: 3.34MB

# With libcoro-fiber
Async::HTTP::Server
simple response
Running 2m test @ http://127.0.0.1:9292/
8 threads and 8 connections
Thread Stats                Avg     Stdev    Max     +/- Stdev
Latency         106.47us 834.32us 99.45ms  99.46%
Req/Sec          12.66k  2.95k  17.61k    71.12%
12093398 requests in 2.00m, 403.66MB read
Requests/sec: 100694.76
Transfer/sec: 3.36MB

This result surprised me a little bit, but now that I think about it, it could make sense (there is also the possibility I made a mistake or the benchmark is bad). Because the cost of network (read/write) and processing (parsing, generating response, buffers, GC) far outweigh the fiber yield/resume, which is already minimised. In real world situations, the results should lean more in favour of libcoro.

Just for interest, I also collect system call stats.

# Without libcoro

<table>
<thead>
<tr>
<th>% time</th>
<th>seconds</th>
<th>usecs/call</th>
<th>calls</th>
<th>errors</th>
<th>syscall</th>
</tr>
</thead>
<tbody>
<tr>
<td>45.76</td>
<td>4.635066</td>
<td>2</td>
<td>2095278</td>
<td></td>
<td>sendto</td>
</tr>
<tr>
<td>32.47</td>
<td>3.288891</td>
<td>1</td>
<td>4191323</td>
<td></td>
<td>rt_sigprocmask</td>
</tr>
<tr>
<td>20.90</td>
<td>2.117062</td>
<td>1</td>
<td>2095611</td>
<td>324</td>
<td>recvfrom</td>
</tr>
<tr>
<td>0.67</td>
<td>0.068189</td>
<td>9741</td>
<td>7</td>
<td></td>
<td>poll</td>
</tr>
</tbody>
</table>

# With libcoro

<table>
<thead>
<tr>
<th>% time</th>
<th>seconds</th>
<th>usecs/call</th>
<th>calls</th>
<th>errors</th>
<th>syscall</th>
</tr>
</thead>
<tbody>
<tr>
<td>45.76</td>
<td>4.635066</td>
<td>2</td>
<td>2095278</td>
<td></td>
<td>sendto</td>
</tr>
<tr>
<td>32.47</td>
<td>3.288891</td>
<td>1</td>
<td>4191323</td>
<td></td>
<td>rt_sigprocmask</td>
</tr>
<tr>
<td>20.90</td>
<td>2.117062</td>
<td>1</td>
<td>2095611</td>
<td>324</td>
<td>recvfrom</td>
</tr>
<tr>
<td>0.67</td>
<td>0.068189</td>
<td>9741</td>
<td>7</td>
<td></td>
<td>poll</td>
</tr>
</tbody>
</table>

08/24/2022
I was evaluating its performance in a C++ program.

I found that it’s not thread safe due to global variables. I change them to thread local to fix the issue, it works well.

I just want to reinforce that this was a proof of concept, if we decide to roll with such an implementation, it requires more work. I am happy to help with that but it would be good to get some feedback regarding whether such a contribution would be acceptable before investing so much time.

Sorry I can't read all of your comments because it too long :p

As you quoted first,

Here is a nice summary of what was done for RethinkDB: https://rethinkdb.com/blog/making-coroutines-fast/

In this article:

A lightweight swapcontext implementation
It shows that swapcontext has extra overhead because of sigprocmask system call.

```
rt_sigprocmask was gone because it's not invoked by libcoro unless using swapcontext.
```

Yes.

Last year, I tried modified swapcontext that article introduced, and I got good performance.
(I found Fiber resume/yield ping ping and I found sigprocmask is one overhead, and google about it, and I also found same page :) )

However, introduced swapcontext is based on glibc, so there is a license problem that we can't merge it into Ruby source code.

Using libcoro (I don't see the library, but as you say) seems to use same tech, so it is one idea to employ.
However, I'm not sure it is the best way.

No conclusion, but it is my current comment.

Thanks,
Koichi

---

#26 - 05/08/2018 10:41 AM - duerst (Martin Dürst)
ioquatix (Samuel Williams) wrote:

Thanks for your feedback. When I made this issue, I could only select "Bug", "Feature" or "Misc". Should I have selected "Misc" instead?

"Feature" should be okay.

#27 - 05/09/2018 02:23 PM - ioquatix (Samuel Williams)

Thanks Koichi, for your valuable response and I appreciate your past work in this area.

I started hacking on my own implementation for x64. It is slightly simpler than libcoro.

I have been reviewing x64 ABI, and it should be pretty trivial to support both 64-bit Windows ABI and 64-bit System V ABI (Linux, Mac, Solaris, BSD). The amount of code is < 200 lines for both ABIs.

For all other ABIs, I suggest using existing code path. I am happy to release this code to Ruby/MRI under whatever license is suitable.

Please be patient while I finish off the patch, when it is done I will update here.

#28 - 05/09/2018 11:34 PM - ioquatix (Samuel Williams)

What compiler is used to compile 64-bit Ruby on Windows?

#29 - 05/10/2018 04:28 AM - ioquatix (Samuel Williams)

Here is the initial code.

https://github.com/kurocha/coroutine

It implements a semantically similar interface to libcoro, but it supports native coroutines on win32, win64 and amd64. I should add a ucontext wrapper (makecontext/swapcontext) for other platforms, then I think all platforms are supported. libcoro didn't have good windows support.

I've put this code under the MIT license.

#30 - 05/14/2018 07:40 AM - sam.saffron (Sam Saffron)

Does this change move us any closer to being able to ship fibers between threads?

#31 - 05/14/2018 07:49 AM - ko1 (Koichi Sasada)

Sorry I missed comments.

How to ship with this library? Bundle it or download by others?
(this is similar discussion with jemalloc :))

#32 - 05/14/2018 08:05 AM - ioquatix (Samuel Williams)

@ko1 (Koichi Sasada) I would suggest we make a Ruby specific version, but we can also try to make generic static library so that it can be maintained separately. I already have some other projects using coroutines so it's useful to me to have a C library implementation which is maintained well.
This is an interesting question which I did specifically try to address in this implementation. I will give you the details.

Typical implementation of Fiber uses thread local variables for main fiber and currently executing fiber Fiber.current. Because of this, it's annoying to ship fiber between threads. Additionally, I'd argue that moving fibers between threads is inherently not safe. I'd kindly suggest that a coroutine which can be resumed on different threads is not a "Fiber" but a "Green Thread". The fundamental difference is how Fiber is implemented, and it depends on thread local storage. For example, how would Fibers#resume work on a different thread if it's executing already? Right now, yield and resume are VERY efficient because they don't have to check anything like this.

However, coroutines are the underlying abstraction for implementing Fiber and they CAN be moved across threads.

This particular implementation was designed very carefully to allow for this. In particular, coroutine_transfer function takes two arguments, a coroutine to store the current stack, and a coroutine to restore it's stack. In particular, coroutine_transfer passes both these arguments to the start function, and additionally, coroutine_transfer returns the coroutine that invoked it, so returning back doesn't require any shared state. Because of this, the implementation avoids any kind of "global" state, it's all on the coroutine stack.

Therefore, with this coroutine library, we can nicely implement green threads too, but you'd need to provide additional guarantees/locking around coroutine_transfer. If you want to transfer a coroutine to another thread, you need to move the coroutine_context data structure (contains stack) to the new thread, and the new thread needs to call coroutine_transfer. The coroutine can simply call coroutine_transfer to return back, using either the argument from or the result of a previous coroutine_transfer.

So, the short answer is yes.

I also finished implementing for arm64, and hopefully can implement for arm32 soon. I test on raspberry pi :) I don't know about PowerPC, I don't have any hardware to test this. Can we test in a VM?

Here is the test which shows coroutine arguments and coroutine_transfer result.

https://github.com/kurocha/coroutine/blob/9bbd5e514c2e0f8f3c7c1c277aa6deb5e337a9c7/test/Coroutine/transfer.cpp#L17-L21

The reason for COROUTINE macro is that on win32, in order to avoid lots of stack manipulation, we need to use __fastcall.

I've made a new branch with the new implementation above.

It shows a slightly improved performance improvement over libcoro.

Here is without the PR:

```
^_^ > ./build/bin/ruby ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers: 0.161763
execution time for 1000 messages: 14.018874
setup time for 10000 fibers: 1.572869
execution time for 1000 messages: 13.778874
setup time for 10000 fibers: 0.917040
execution time for 1000 messages: 13.942525
setup time for 10000 fibers: 1.616929
execution time for 1000 messages: 14.281334
```

And here it is with the PR, on macOS (the same system used in previous benchmarks):

```
^_^ > ./build/bin/ruby ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers: 0.160637
execution time for 1000 messages: 6.009332
setup time for 10000 fibers: 6.009332
execution time for 1000 messages: 6.246711
setup time for 10000 fibers: 0.242718
execution time for 1000 messages: 6.142166
setup time for 10000 fibers: 0.233410
execution time for 1000 messages: 5.994752
setup time for 10000 fibers: 0.288830
execution time for 1000 messages: 6.216617
```

Performance is about 2~2.5x faster depending on your analysis. Both creation and execution time is improved. But remember this is micro-benchmark.

I was also interested in mjit performance:

Without PR, enabled mjit:

```
^_^ > ./build/bin/ruby --jit ./fiber_benchmark.rb 10000 1000
```
setup time for 10000 fibers: 0.172145
execution time for 1000 messages: 25.176702
setup time for 10000 fibers: 1.654751
execution time for 1000 messages: 14.729177
setup time for 10000 fibers: 1.016810
execution time for 1000 messages: 15.154141
setup time for 10000 fibers: 1.726305
execution time for 1000 messages: 14.797269
setup time for 10000 fibers: 2.025997
execution time for 1000 messages: 15.124753

With PR, enabled mjit:

```
x_x > ./build/bin/ruby --jit ./fiber_benchmark.rb 10000 1000
setup time for 10000 fibers: 0.179744
execution time for 1000 messages: 13.793318
setup time for 10000 fibers: 0.354717
execution time for 1000 messages: 10.664870
setup time for 10000 fibers: 0.308818
execution time for 1000 messages: 6.956352
setup time for 10000 fibers: 0.378568
execution time for 1000 messages: 6.553922
setup time for 1000 fibers: 0.295583
execution time for 1000 messages: 7.274086
```

We can see it still needs a bit of work.

I will try to isolate some interesting results from higher level frameworks.

The updated branch is here: https://github.com/ioquatix/ruby/tree/native-fiber

It only work on Darwin x64 at the moment, because changes to autoconf do not cover all platforms yet. I'll fix this soon.

#35 - 06/02/2018 01:07 AM - ioquatix (Samuel Williams)

I fixed autoconf issues and built on Linux. The performance improvement was even more impressive.

```
koyoko% ruby --version
ruby 2.6.0dev (2018-06-01 native-fiber 63544) [x86_64-linux]
last_commit=Better support for amd64 platforms
koyoko% ruby ./fiber_benchmark.rb
setup time for 10000 messages: 3.433891
setup time for 1000 fibers: 0.015365
execution time for 10000 messages: 3.177730
execution time for 1000 fibers: 0.010035
```

@nobu (Nobuyoshi Nakada) do you mind confirming?

#36 - 06/02/2018 01:34 AM - ioquatix (Samuel Williams)

Here is a more realistic benchmark which fiber context switch is only a tiny percentage of the actual run-time.

A brief summary of the benchmark: async-http uses an event-driven stackful coroutine (fiber) based design. Each request allocates a fiber, and each blocking operation (i.e. read) results in Fiber.yield. Once the IO is ready, Fiber#resume is called. So, for each request being processed, we expect...
several calls to Fiber.yield. async is optimistic so it tries to perform the operation e.g. read and only yields if it results in EWOULDBLOCK so in some cases (especially in synthetic benchmarks) some scheduling may be elided.

koyoko% rvm use 2.6
Using /home/samuel/.rvm/gems/ruby-2.6.0-preview2
koyoko% ruby --version
ruby 2.6.0-preview2 (2018-05-31 trunk 63539) [x86_64-linux]
koyoko% bundle exec rake wrk
Running 10s test @ http://127.0.0.1:9294/
  1 threads and 1 connections
    Thread Stats Avg Stdev  Max +/- Stdev
    Latency 63.59us 77.52us 4.53ms  98.32%
    Req/Sec  16.68k  1.07k 18.32k   74.26%
167544 requests in 10.10s, 14.54MB read
Requests/sec: 16589.33
Transfer/sec:  1.44MB

Running 10s test @ http://127.0.0.1:9294/
  1 threads and 1 connections
    Thread Stats Avg Stdev  Max +/- Stdev
    Latency 60.85us 34.26us 1.39ms  95.82%
    Req/Sec  16.82k  0.87k 18.49k   70.00%
167424 requests in 10.00s, 14.53MB read
Requests/sec: 16742.19
Transfer/sec:  1.45MB

Running 10s test @ http://127.0.0.1:9294/
  1 threads and 1 connections
    Thread Stats Avg Stdev  Max +/- Stdev
    Latency 62.44us 54.34us 3.81ms  97.62%
    Req/Sec  16.62k  1.00k 18.09k   67.33%
166959 requests in 10.10s, 14.49MB read
Requests/sec: 16530.76
Transfer/sec:  1.43MB

Running 10s test @ http://127.0.0.1:9294/
  1 threads and 1 connections
    Thread Stats Avg Stdev  Max +/- Stdev
    Latency 61.89us 32.53us 687.00us 94.29%
    Req/Sec  16.54k  1.20k 18.37k   67.33%
166105 requests in 10.10s, 14.42MB read
Requests/sec: 16445.91
Transfer/sec:  1.43MB

Here is with the PR:

koyoko% rvm use ruby-head-fiber
Using /home/samuel/.rvm/gems/ruby-head-fiber
koyoko% ruby --version
ruby 2.6.0dev (2018-06-01 native-fiber 63544) [x86_64-linux]
lst_commit=Better support for amd64 platforms
koyoko% bundle exec rake wrk
Running 10s test @ http://127.0.0.1:9294/
  1 threads and 1 connections
    Thread Stats Avg Stdev  Max +/- Stdev
    Latency 62.53us 73.11us 5.02ms  97.96%
    Req/Sec  16.89k  1.35k 19.46k   63.37%
168863 requests in 10.10s, 14.65MB read
Requests/sec: 16719.77
Transfer/sec:  1.45MB

Running 10s test @ http://127.0.0.1:9294/
  1 threads and 1 connections
    Thread Stats Avg Stdev  Max +/- Stdev
    Latency 61.90us 37.64us 1.70ms  95.89%
    Req/Sec  16.89k  1.22k 18.57k   72.28%
169694 requests in 10.10s, 14.73MB read
Requests/sec: 16802.33
Transfer/sec:  1.45MB

Here is with the PR:
This is actually better than I expected. I would say there is a practical improvement of about ~5%. In this situation it’s very workload dependent, but I'm glad that I saw something.

#37 - 06/03/2018 02:47 AM - ioquatix (Samuel Williams)
I've made a short blog post about this PR: https://www.codeotaku.com/journal/2018-06/improving-ruby-fibers/index

#38 - 06/06/2018 09:32 PM - cremes (Chuck Remes)
I'd like to link this to another open issue regarding Fiber migration between threads. https://bugs.ruby-lang.org/issues/13821

@ioquatix (Samuel Williams), please note in the above-referenced bug that I put in a link to the "boost" documentation regarding coroutine movement between threads. An explicit API to lock/unlock ownership of the fiber to a thread would probably resolve some of the complaints people raise about fiber migration. If it's explicit, more guarantees can be made. Default behavior should be the current behavior where Fibers cannot migrate.

Thanks for your work on this.

#39 - 06/06/2018 11:59 PM - ioquatix (Samuel Williams)
@cremes (Chuck Remes) Thanks for your positive feedback and linking me to related issues.

The coroutine implementation was specifically designed to handle cross-thread migrations, in the sense that all the required state to yield/resume is passed as arguments/returns to/from the coroutine.

What this means is that no global/thread-local state is required and thus when moving a coroutine to another thread, there is almost no additional data to sync which is nice from an API point of view.

The bigger challenge is how Ruby Fiber is implemented. It does make it tricky. I would be happy to work towards this. I see the following path being viable:

- Merge these changes.
- Simplify the Fiber implementation by removing all the other implementations from cont.c and if necessary move these to the coroutine code (but ideally remove them).
- With the simplified Fiber code base, explore the overheads of Fiber creation/context switching and figure out the right places to put locking/checks (e.g. for locks being held, etc).

#40 - 09/13/2018 07:25 AM - matz (Yukihiro Matsumoto)
OK, it sounds reasonable. We will give you commit privilege.

Matz.

#41 - 09/13/2018 07:33 AM - hsbt (Hiroshi SHIBATA)
Hi, ioquatix.

I send an invitation of the ruby core team. Please check it.

#42 - 12/11/2018 11:55 PM - ioquatix (Samuel Williams)
- Status changed from Open to Closed
- Assignee set to ioquatix (Samuel Williams)
This is now implemented across: arm32, arm64, ppc64le, win32, win64, x86, amd64. Thanks to everyone who helped with this. This is a really awesome first step to improving Ruby Fiber performance.