

Ruby - Feature #18339

GVL instrumentation API

11/15/2021 09:29 AM - byroot (Jean Boussier)

Status: Closed

Priority: Normal

Assignee:

Target version:

Description

GVL instrumentation API

Context

One of the most common if not the most common way to deploy Ruby application these days is through threaded runners (typically puma, sidekiq, etc).

While threaded runners can offer more throughput per RAM than forking runners, they do require to carefully set the concurrency level (number of threads).

If you increase concurrency too much, you'll experience GVL contention and the latency will suffer.

The common way today is to start with a relatively low number of threads, and then increase it until CPU usage reach an acceptably high level (generally ~75%).

But this method really isn't precise, require to saturate one process with fake workload, and doesn't tell how much threads are waiting on the GVLs, just how much the CPU is used.

Because of this, lots of threaded applications are not that well tuned, even more so because the ideal configuration is very dependant on the workload and can vary over time. So a decent setting might not be so good six months later.

Ideally, application owners should be able to continuously see the impact of the GVL contention on their latency metric, so they can more accurately decide what throughput vs latency tradeoff is best for them and regularly adjust it.

Existing instrumentation methods

Currently, if you want to measure how much GVL contention is happening, you have to use some lower level tracing tools such as bpftrace, or dtrace. These are quite advanced tools and require either root access or to compile Ruby with different configuration flags etc.

They're also external, so common Application Performance Monitoring (APM) tools can't really report it.

Proposal

I'd like to have a C-level hook API around the GVL, with 3 events:

- RUBY_INTERNAL_EVENT_THREAD_READY
- RUBY_INTERNAL_EVENT_THREAD_RESUME
- RUBY_INTERNAL_EVENT_THREAD_PAUSE

Such API would allow to implement C extensions that collect various metrics about the GVL impact, such as median / p90 / p99 wait time, or even per thread total wait time.

Additionally it would be very useful if the hook would pass some metadata, most importantly, the number of threads currently waiting. People interested in a lower overhead monitoring method not calling clock_gettime could instrument that number of waiting thread instead. It would be less accurate, but enough to tell wether there might be problem.

With such metrics, application owners would be able to much more precisely tune their concurrency setting, and deliberately chose their own tradeoff between throughput and latency.

Implementation

I submitted a PR for it <https://github.com/ruby/ruby/pull/5500> (lacking windows support for now)

The API is as follow:

- `rb_thread_hook_t * rb_thread_event_new(rb_thread_callback callback, rb_event_flag_t event)`
- `bool rb_thread_event_delete(rb_thread_hook_t * hook)`

The overhead when no hook is registered is just a single unprotected boolean check, so close to zero.

Associated revisions

Revision 9125374726bf68c05ee7585d4a374ffc5efc5db - 06/03/2022 01:13 PM - byroot (Jean Boussier)

[Feature #18339] GVL Instrumentation API

Ref: <https://bugs.ruby-lang.org/issues/18339>

Design:

- This tries to minimize the overhead when no hook is registered. It should only incur an extra unsynchronized boolean check.
- The hook list is protected with a read-write lock as to cause contention when some hooks are registered.
- The hooks MUST be thread safe, and MUST NOT call into Ruby as they are executed outside the GVL.
- It's simply a noop on Windows.

API:

```
rb_internal_thread_event_hook_t * rb_internal_thread_add_event_hook(rb_internal_thread_event_callback callback, rb_event_flag_t internal_event, void *user_data);
bool rb_internal_thread_remove_event_hook(rb_internal_thread_event_hook_t * hook);
```

You can subscribe to 3 events:

- READY: called right before attempting to acquire the GVL
- RESUMED: called right after successfully acquiring the GVL
- SUSPENDED: called right after releasing the GVL.

The hooks MUST be threadsafe, as they are executed outside of the GVL, they also MUST NOT call any Ruby API.

Revision 9125374726bf68c05ee7585d4a374ffc5efc5db - 06/03/2022 01:13 PM - byroot (Jean Boussier)

[Feature #18339] GVL Instrumentation API

Ref: <https://bugs.ruby-lang.org/issues/18339>

Design:

- This tries to minimize the overhead when no hook is registered. It should only incur an extra unsynchronized boolean check.
- The hook list is protected with a read-write lock as to cause contention when some hooks are registered.
- The hooks MUST be thread safe, and MUST NOT call into Ruby as they are executed outside the GVL.
- It's simply a noop on Windows.

API:

```
rb_internal_thread_event_hook_t * rb_internal_thread_add_event_hook(rb_internal_thread_event_callback callback, rb_event_flag_t internal_event, void *user_data);
bool rb_internal_thread_remove_event_hook(rb_internal_thread_event_hook_t * hook);
```

You can subscribe to 3 events:

- READY: called right before attempting to acquire the GVL
- RESUMED: called right after successfully acquiring the GVL
- SUSPENDED: called right after releasing the GVL.

The hooks MUST be threadsafe, as they are executed outside of the GVL, they also MUST NOT call any Ruby API.

Revision 91253747 - 06/03/2022 01:13 PM - byroot (Jean Boussier)

[Feature #18339] GVL Instrumentation API

Ref: <https://bugs.ruby-lang.org/issues/18339>

Design:

- This tries to minimize the overhead when no hook is registered. It should only incur an extra unsynchronized boolean check.
- The hook list is protected with a read-write lock as to cause contention when some hooks are registered.
- The hooks MUST be thread safe, and MUST NOT call into Ruby as they are executed outside the GVL.
- It's simply a noop on Windows.

API:

```
rb_internal_thread_event_hook_t * rb_internal_thread_add_event_hook (rb_internal_thread_event_callback callback, rb_event_flag_t internal_event, void *user_data);  
bool rb_internal_thread_remove_event_hook (rb_internal_thread_event_hook_t * hook);
```

You can subscribe to 3 events:

- READY: called right before attempting to acquire the GVL
- RESUMED: called right after successfully acquiring the GVL
- SUSPENDED: called right after releasing the GVL.

The hooks MUST be threadsafe, as they are executed outside of the GVL, they also MUST NOT call any Ruby API.

Revision b6c1e1158d71b533b255ae7a2731598455918071 - 06/17/2022 07:08 AM - byroot (Jean Boussier)

GVL Instrumentation API: add STARTED and EXITED events

[Feature #18339]

After experimenting with the initial version of the API I figured there is a need for an exit event to cleanup instrumentation data. e.g. if you record data in a {thread_id -> data} table, you need to free associated data when a thread goes away.

Revision b6c1e1158d71b533b255ae7a2731598455918071 - 06/17/2022 07:08 AM - byroot (Jean Boussier)

GVL Instrumentation API: add STARTED and EXITED events

[Feature #18339]

After experimenting with the initial version of the API I figured there is a need for an exit event to cleanup instrumentation data. e.g. if you record data in a {thread_id -> data} table, you need to free associated data when a thread goes away.

Revision b6c1e115 - 06/17/2022 07:08 AM - byroot (Jean Boussier)

GVL Instrumentation API: add STARTED and EXITED events

[Feature #18339]

After experimenting with the initial version of the API I figured there is a need for an exit event to cleanup instrumentation data. e.g. if you record data in a {thread_id -> data} table, you need to free associated data when a thread goes away.

History

#1 - 11/16/2021 01:38 PM - mame (Yusuke Endoh)

Looks interesting.

I understood that the basic usage is to measure elapsed time between RUBY_INTERNAL_EVENT_GVL_ACQUIRE_ENTER and RUBY_INTERNAL_EVENT_GVL_ACQUIRE_EXIT, to increase the number of threads when the elapsed time is short, and to decrease the number when the time is long. Is my understanding right? I have no idea how to use RUBY_INTERNAL_EVENT_GVL_RELEASE.

I have one concern. I think the hooks are called without GVL, but I wonder if the current hook implementation is robust enough without GVL.

Another idea is to provide a method Thread.gvl_waiting_thread_count. Puma or an application monitor can call it periodically (every one second, every request, or something), reduce thread pool size when the count is high, and add a new thread when the count is low. This is less flexible, but maybe more robust.

require to saturate one process with fake workload

I couldn't understand this. Could you explain this?

#2 - 11/16/2021 01:50 PM - byroot (Jean Boussier)

to increase the number of threads when the elapsed time is short, and to decrease the number when the time is long. Is my understanding right?

That is correct.

Another potential use case is to simply instrument the number of waiting thread, less precise, but faster.

I have no idea how to use `RUBY_INTERNAL_EVENT_GVL_RELEASE`.

I mostly suggest it for completeness. I don't have an use case for it, but maybe you might want to instrument how long threads hold the GVL etc.

I think the hooks are called without GVL

Yes, that is indeed a concern. I looked at the GC hooks and I was under the impression that it was already the case there. I might be wrong.

Another idea is to provide a method `Thread.gvl_waiting_thread_count`.

Yes, it would be less flexible and less precise but would help. I'd still prefer the C hooks though.

How I see it `gvl_waiting_thread_count` is useful for low overhead monitoring, it will tell you you might have a problem. Once you see there's one, you might want to temporarily enable actual time measurement to finely tune your service.

require to saturate one process with fake workload / I couldn't understand this. Could you explain this?

Yes, to measure the effects of your number of threads, you must saturate your server. Meaning having all threads always working, otherwise your setting might look fine one day, but once you have a spike of traffic the performance tanks.

So usually you do this with some synthetic (fake) traffic.

The problem is that not all work loads are equal. Some endpoints will be CPU intensive, some other IO intensive, it's very hard to create synthetic traffic that perfectly reflect production. And even if you can, in a few weeks or months the pattern may have changed.

That is why I think constant monitoring of the situation in production is preferable to a "test bench" method using fake load. Does it make more sense?

#3 - 11/16/2021 02:20 PM - mame (Yusuke Endoh)

I see! You are creating an artificially high load condition to adjust the best settings for high loads, aren't you?

And you think this feature will allow to automatically adjust settings according to the current actual load.

I don't know how difficult it is to implement such an automatic adjustment, but it is super cool if possible.

Thank you for the clarification!

#4 - 11/16/2021 03:02 PM - byroot (Jean Boussier)

You are creating an artificially high load condition to adjust the best settings for high loads, aren't you?

Yes.

And you think this feature will allow to automatically adjust settings according to the current actual load.

That's a possibility, I'm not certain of that. The main objective short term is monitoring for humans, but yes a nice feature would be puma/sidekiq/etc being able to apply backpressure by reducing their concurrency dynamically.

#5 - 11/18/2021 02:11 PM - mame (Yusuke Endoh)

[@matz \(Yukihiro Matsumoto\)](#) left this to [@ko1 \(Koichi Sasada\)](#), and ko1 was basically positive for this proposal.

byroot (Jean Boussier) wrote in [#note-2](#):

I think the hooks are called without GVL

Yes, that is indeed a concern. I looked at the GC hooks and I was under the impression that it was already the case there. I might be wrong.

According to ko1, GC hooks are called with GVL, so this is the first time to call hooks in parallel. The current hook implementation is not robust at all

for such a usage. It will break if one thread adds/removes a new hook and another thread iterates the hook list simultaneously. You need to lock the data structure by using a mutex, or copy GVL-related hook set to thread-local storage before releasing the GVL.

#6 - 11/18/2021 02:15 PM - byroot (Jean Boussier)

ko1 was basically positive for this proposal.

That's great to hear!

The current hook implementation is not robust at all for such a usage.

Understood. So since [@ko1 \(Koichi Sasada\)](#) is mostly positive on the idea, how should we proceed? Is it realistic to try to harden the existing `rb_tracepoint_new` API, or should this GVL instrumentation API just not rely on it?

#7 - 12/03/2021 04:09 AM - ko1 (Koichi Sasada)

ah, it not easy to implement...
could you try?

#8 - 12/03/2021 09:04 AM - byroot (Jean Boussier)

could you try?

Sure. Would likely be a next year thing anyway.

However Is a new dedicated hook API ok? Since apparently making the tracepoint API usable outside the GVL would be very hard.

#9 - 01/27/2022 04:22 PM - byroot (Jean Boussier)

I opened a draft for it: <https://github.com/ruby/ruby/pull/5500>

[@ko1 \(Koichi Sasada\)](#) if you'd like to have a look.

#10 - 02/08/2022 06:52 PM - byroot (Jean Boussier)

- *Description updated*

#11 - 02/15/2022 05:52 AM - ko1 (Koichi Sasada)

- Could you write a picture (or flow) what happens on GVL hooks with GVL acquire/release?
- It seems it doesn't use TracePoint data structures, so I think we don't need to use `RUBY_INTERNAL_EVENT_*` naming.
- I'm planning to introduce M:N threads so I'm not sure what you want to monitor on it.

#12 - 02/15/2022 08:11 AM - byroot (Jean Boussier)

Could you write a picture (or flow) what happens on GVL hooks with GVL acquire/release?

I'm sorry, I don't understand the question.

so I think we don't need to use `RUBY_INTERNAL_EVENT_*` naming.

Sure. It was just for consistency, but I can rename it to anything.

I'm planning to introduce M:N threads so I'm not sure what you want to monitor on it.

I'm not very familiar with M:N threads, but from my limited understanding yes it might make what I'm trying to do difficult.

What I want to do with these hooks is to be able to tell users whether they spawned too many threads. So measure how much time threads spent waiting to be scheduled when they could have been executing.

Maybe there would be a better solution for this?

#13 - 04/14/2022 07:28 AM - byroot (Jean Boussier)

- *Description updated*

#14 - 06/03/2022 01:14 PM - byroot (Jean Boussier)

- *Status changed from Open to Closed*

Merged as <https://github.com/ruby/ruby/commit/9125374726fbf68c05ee7585d4a374ffc5efc5db> after several rounds of review with [@ko1 \(Koichi Sasada\)](#)