End of boot advisory API for RubyVM

06/28/2022 01:21 PM - byroot (Jean Boussier)

Description

Context

Many optimizations in the Ruby VM rely on lazily computed caches: Strings coderange, constant caches, method caches, etc etc. As such even without JIT, some operations need a bit of a warm up, and might be flushed if new constants are defined, new code is loaded, or some objects are mutated.

Additionally these lazily computed caches can cause increased memory usage for applications relying on Copy-on-Write memory. Whenever one of these caches is updated post fork, the entire memory page is invalidated. Precomputing these caches at the end of boot, even if based on heuristic, could improve Copy-on-Write performance.

The classic example is the objects generation, young objects must be promoted to the old generation before forking, otherwise they'll get invalidated on the next GC run. That's what [https://github.com/ko1/nakayoshi_fork](https://github.com/ko1/nakayoshi_fork) addresses.

But there are other sources of CoW invalidation that could be addressed by MRI if it had a clear notification when it needs to be done.

Proposal

If applications had an API to notify the virtual machine that they're done loading code and are about to start processing user input, it would give the VM a good point in time to perform optimizations on the existing code and objects.

e.g. could be something like RubyVM.prepare, or RubyVM.ready.

It's somewhat similar to Matz's static barrier idea from RubyConf 2020, except that it wouldn't disable any feature.

Potential optimizations

nakayoshi_fork already does the following:

- Do a major GC run to get rid of as many dangling objects as possible.
- Promote all surviving objects to the highest generation
- Compact the heap.

But it would be much simpler to do this from inside the VM rather than do cryptic things such as 4.times { GC.start } from the Ruby side.

It's also not good to do this on every fork, once you fork the first long lived child, you shouldn't run it again. So decorating fork is not a good hook point.

Also after discussing with @hawthorn (John Hawthorn), @tenderlovemaking (Aaron Patterson) and @alanwu (Alan Wu), we believe this would open the door to several other CoW optimizations:

Precompute inline caches

Even though we don't have hard data to prove it, we are convinced that a big source of CoW invalidation are inline caches. Most ISeq are never invoked during initialization, so child processed are forked with mostly cold caches. As a result the first time a method is executed in the child, many memory pages holding ISeq are invalidated as caches get updated.

We think MRI could try to precompute these caches before forking children. Constant cache particularly should be resolvable statically see [https://github.com/ruby/ruby/pull/6187](https://github.com/ruby/ruby/pull/6187).

Method caches are harder to resolve statically, but we can probably apply some heuristics to at least reduce the cache misses.
Copy on Write aware GC

We could also keep some metadata about which memory pages are shared, or even introduce a "permanent" generation. The Instagram engineering team introduced something like that in Python (ticket, PR).

That makes the GC aware of which objects live on a shared page. With this information the GC can decide to no free dangling objects leaving on these pages, not to compact these pages, etc.

Scan the coderange of all strings

Strings have a lazily computed coderange attribute in their flags. So if a string is allocated at boot, but only used after fork, on first use its coderange may need to be computed and the string mutated.

Using https://github.com/ruby/ruby/pull/6076, I noticed that 58% of the strings retained at the end of the boot sequence had an UNKNOWN coderange.

So eagerly scanning the coderange of all strings could also improve Copy on Write performance.

malloc_trim

This hook will also be a good point to release unused pages to the system with malloc_trim.

Related issues:

Related to Ruby master - Feature #11164: Garbage collector in Ruby 2.2 provokes unexpected CoW added

Rejected

History

#1 - 06/28/2022 01:22 PM - byroot (Jean Boussier)
- Related to Feature #11164: Garbage collector in Ruby 2.2 provokes unexpected CoW added

#2 - 06/30/2022 09:27 AM - byroot (Jean Boussier)
- Description updated

Another possible optimization I just found:

Strings have a lazily computed coderange attribute in their flags. So if a string is allocated at boot, but only used after fork, its coderange may be computed and the string mutated.

Using https://github.com/ruby/ruby/pull/6076, I noticed that 58% of the strings retained at the end of the boot sequence had an UNKNOWN coderange.

So eagerly scanning the coderange of all strings could also improve Copy on Write performance.

This also makes me think that this API isn't only useful for forking setup. Even if you use only threads or fibers, you may want to tell the VM that you are done loading and that it's now time to perform optimizations. So the API may use a more generic name.

#3 - 07/16/2022 03:19 AM - ioquatix (Samuel Williams)
This is a really nice idea. My current implementation uses GC.compact during prefork stage, and it shows a big advantage. I'm happy to test any proposals with real world workloads.

#4 - 07/27/2022 04:55 PM - byroot (Jean Boussier)
Another optimization that could be invoked from this method is malloc_trim.

#5 - 07/30/2022 02:33 AM - Dan0042 (Daniel DeLorme)
I think the state of Copy-on-Write is already pretty decent, but any improvement is of course very welcome. As to naming, since this is mainly for preforking servers, what about Process.prefork?

#6 - 07/30/2022 06:19 AM - byroot (Jean Boussier)

the state of Copy-on-Write is already pretty decent,

It depends how you look at it. In the few apps on which I optimized CoW as much as I could, only between 50% and 60% of the parent process memory is shared. That really isn't that good.

08/23/2022
We discussed this issue at the dev meeting. We did not reach any conclusion, but I'd like to share some comments.

**What and how efficient is this proposal?**

Some attendees wanted to confirm quantitative evaluation of the benefits this proposal would bring.

@ko1 (Koichi Sasada) said that he created nakayoshi_fork as a joke gem. He didn't expect people to use it seriously, and he didn't have serious quantitative measurements.

(I've heard people say that memory usage has been reduced by nakayoshi_fork, but it would be nice to be properly confirm this advantage before introduction.)

**How is it integrated with Process._fork?**

Process._fork has been introduced as an zero-argument API. This API is supposed to be overridden, so we cannot add an argument easily. If we keep Process._fork as is, we need to do some GC processes like nakayoshi_fork before the hook of Process._fork. Is it OK?

**Are "short-lived" forks needed?**

How much are "short-lived" forks used nowadays? The major use case where Process.exec is called shortly after Process.fork, is covered by Process.spawn. If there is few use cases for "short-lived" forks, we may change the default behavior to "long-lived". However, we sometimes use fork in tests, to invoke a temporal web server, for example. Calling GC whenever calling fork might be too heavy.

**Is GC called whenever fork(long_lived: true) is called?**

Here is a typical server code that uses fork:

```ruby
loop do
  sock = servsock.accept
  if fork(long_lived: true)
    ...
  end
end
```

The parent process creates only a socket object for each iteration. It looks somewhat useless to call full GC in the parent process every time fork(long_lived: true) is called. A more intelligent strategy may be preferable here.

**#8 - 08/02/2022 09:03 AM - byroot (Jean Boussier)**

@mame (Yusuke Endoh)

it would be nice to be properly confirm this advantage before introduction.

https://bugs.ruby-lang.org/issues/11164 is an example of how bad things can go without nakayoshi_fork (or similar). I can get production data from some of our apps if you wish, but the effect is going to be very app dependent, so I'm not sure if it's very relevant. You can craft demo-apps for which memory usage totally blow up if you don't promote objects to the old generation before forking.

**How is it integrated with Process._fork?**

Since I wrote this, I'm now convinced that it shouldn't be a fork argument, but a distinct API on RubyVM, since if you fork multiple workers you don't want to run them things again as it could invalidate CoW in previous workers.

So IMO RubyVM.prepare_for_long_lived_fork is the proper API (aside from the name which I doubt is desirable).

Also please note that several of the proposed optimization can only be done from inside Ruby, so decorating like nakayoshi_fork does is not an option. Hence the main reason of this proposal.

**#9 - 08/03/2022 01:32 AM - mame (Yusuke Endoh)**

So IMO RubyVM.prepare_for_long_lived_fork is the proper API (aside from the name which I doubt is desirable).

After calling this, all fork calls are treated as "long-lived". Is my understanding right?

Also please note that several of the proposed optimization can only be done from inside Ruby, so decorating like nakayoshi_fork does is not an option.

I know that, but it seemed hard to me to convince the committers to change the API first for optimizations that have not been implemented yet and we
don’t know how effective they will be. IMO, it is good to focus on the use case of nakayoshi_fork since it is already implemented and used by not a few people. If there is a proper evaluation of the effect of nakayoshi_fork, it would be easier to persuade @matz (Yukihiro Matsumoto).

#10 - 08/03/2022 06:42 AM - byroot (Jean Boussier)

After calling this, all fork calls are treated as "long-lived". Is my understanding right?

Well, this wouldn’t change anything to the Process.fork implementation. I think I need to rewrite the ticket description because it is now confusing, I’ll do it in a minute.

Also as said before I don’t even think this is specific to forking servers anymore, I think RubyVM.make_ready or something like that would be just fine. Even if you don’t fork, optimizations such as precomputing inline caching could improve performance of the first request.

it is good to focus on the use case of nakayoshi_fork

Ok, so here’s a thread from when Puma added it as an option two years ago, https://github.com/puma/puma/issues/2258#issuecomment-630510423

After fixing the config bug in nakayoshi_fork, Codetriage is now showing about a 10% reduction in memory usage

Some other people report good numbers too, but generally they enabled other changes at the same time.

#11 - 08/03/2022 07:10 AM - byroot (Jean Boussier)
- Subject changed from Long lived fork advisory API (potential Copy on Write optimizations) to End of boot advisory API for RubyVM
- Description updated

Ok, Ip updated the description, it's still very much focused on CoW, but hopefully it should now be more clear that’s it’s not the only benefit.

Also it now only ask a method on RubyVM, which could perfectly be marked as experimental, so API change concerns should be minimal.

#12 - 08/10/2022 06:21 PM - Dan0042 (Daniel DeLorme)

I think the terminology used here might cause some confusion in the discussion.

"End of boot" makes it sound like this API would be useful for non-forking servers once they have finished their "boot" sequence. But from what I understand this is still very much a fork-specific API. Is there any point to precompute inline caches if there is no fork?

"Long lived" children processes are not really the point I think? Imagine a (ridiculous) architecture where the parent keeps spawning children and each child serves a single request before dying. Despite being short-lived, these processes would benefit from this API. So it's not about preparing children for being long-lived, it's about preparing the parent for having any children.

#13 - 08/10/2022 06:24 PM - byroot (Jean Boussier)

Is there any point to precompute inline caches if there is no fork?

Yes, the first "request" (or whatever your unit of work is) won't have to do it. So you are moving some work to boot time, instead of user input processing time.

these processes would benefit from this API.

For the CoW parts no, not much. If the child isn't going to live for long, it's unlikely to invalidate that many pages.

#14 - 08/18/2022 06:51 AM - matz (Yukihiro Matsumoto)

I am OK with adding this feature, but I have some concerns with the place and the name. RubyVM is not globally available (e.g., not for JRuby or TruffleRuby). And I don't think prepare or ready describes the whole functionality.

Matz.

#15 - 08/18/2022 06:55 AM - byroot (Jean Boussier)

Thank you Matz.

RubyVM is not globally available (e.g., not for JRuby or TruffleRuby).
Yes, what was on purpose because the behavior would be very VM specific, some VM might not even to have it. It's not meant to be a cross implementation feature.

And I don't think prepare or ready describes the whole functionality.

I'll try to come up with other names.

**#16 - 08/18/2022 07:14 AM - Eregon (Benoit Daloze)**

An API to notify "end of boot" seems useful beyond just fork COW optimizations, as you say. For instance a JIT might use that as a hint for what to compile/stop compiling/purge the queue during boot/reset compilation counters/etc. So it shouldn't be under RubyVM which means only available on CRuby (forever).

Maybe a Kernel class method?

Kernel.booted/Kernel.application_booted/Kernel.code_loaded/Kernel.startup_done maybe?

**#17 - 08/18/2022 07:16 AM - byroot (Jean Boussier)**

What about ObjectSpace?