Ruby trunk - Feature #14723

[WIP] sleepy GC

04/29/2018 03:57 AM - normalperson (Eric Wong)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

The idea is to use "idle time" when process is otherwise sleeping
and using no CPU time to perform GC. It makes sense because real
world traffic sees idle time due to network latency and waiting
for user input.

Right now, it's Linux-only. Future patches will affect other sleeping
functions:

IO.select, Kernel#sleep, Thread#join, Process.waitpid, etc...

I don't know if this patch can be implemented for win32, right
now it's just dummy functions and that will be somebody elses
job. But all pthreads platforms should eventually benefit.

Before this patch, the entropy-dependent script below takes 95MB
consistently on my system. Now, depending on the amount of
entropy on my system, it takes anywhere from 43MB to 75MB.

I'm using /dev/urandom to simulate real-world network latency
variations. There is no improvement when using /dev/zero
because the process is never idle.

```
require 'net/http'
require 'digest/md5'
Thread.abort_on_exception = true
s = TCPServer.new('127.0.0.1', 0)
len = 1024 * 1024 * 1024
th = Thread.new do
c = s.accept
c.readpartial(16384)
c.write("HTTP/1.0 200 OK\r\nContent-Length: #{len}\r\n\r\n")
IO.copy_stream('/dev/urandom', c, len)
c.close
end

addr = s.addr
Net::HTTP.start(addr[3], addr[1]) do |http|
http.request_get('/') do |res|
dig = Digest::MD5.new
res.read_body { |buf|
dig.update(buf)
}
puts dig.hexdigest
end
end
```

The above script is also dependent on net/protocol using
read_nonblock. Ordinary IO objects will need IO#nonblock=true
to see benefits (because they never hit rb_wait_for_single_fd)

- gc.c (rb_gc_inprogress): new function (rb_gc_step): ditto
- internal.h: declare prototypes for new gc.c functions
- thread_pthread.c (gvl_contended_p): new function
- thread_win32.c (gvl_contended_p): ditto (dummy)

- thread.c (rb_wait_for_single_fd w/ ppoll): use new functions to perform GC while GVL is uncontended and GC is lazy sweeping or incremental marking [ruby-core:86265] ```

2 part patch broken out
https://80x24.org/spew/20180429035007.6499-2-e@80x24.org/raw
https://80x24.org/spew/20180429035007.6499-3-e@80x24.org/raw

Also on my "sleepy-gc" git branch @ git://80x24.org/ruby.git

History

#1 - 04/29/2018 04:52 AM - ko1 (Koichi Sasada)

Could you give us more detail algorithm?

2018/04/29 12:57□normalperson@yhbt.net□□□□:

> Issue #14723 has been reported by normalperson (Eric Wong).
>
> ---
>
> Feature #14723: [WIP] sleepy GC
> https://bugs.ruby-lang.org/issues/14723
>
> - Author: normalperson (Eric Wong)
> - Status: Open
> - Priority: Normal
> - Assignee:
>
> ## * Target version:
>
> The idea is to use "idle time" when process is otherwise sleeping
> and using no CPU time to perform GC. It makes sense because real
> world traffic sees idle time due to network latency and waiting
> for user input.
>
> Right now, it's Linux-only. Future patches will affect other sleeping
> functions:
>
> IO.select, Kernel#sleep, Thread#join, Process.waitpid, etc...
>
> I don't know if this patch can be implemented for win32, right
> now it's just dummy functions and that will be somebody elses
> job. But all pthreads platforms should eventually benefit.
>
> Before this patch, the entropy-dependent script below takes 95MB
> consistently on my system. Now, depending on the amount of
> entropy on my system, it takes anywhere from 43MB to 75MB.
>
> I'm using /dev/urandom to simulate real-world network latency
> variations. There is no improvement when using /dev/zero
> because the process is never idle.
>
> ```
> require 'net/http'
> require 'digest/md5'
> Thread.abort_on_exception = true
> s = TCPServer.new('127.0.0.1', 0)
> len = 1024 * 1024 * 1024
> th = Thread.new do
> c = s.accept
> c.readpartial(16384)
> c.write("HTTP/1.0 200 OK\r\nContent-Length: #{len}\r\n\r\n")
> IO.copy_stream('/dev/urandom', c, len)
> c.close
> end
>
> addr = s.addr
> Net::HTTP.start(addr[3], addr[1]) do |http|
> http.request_get('/') do |res|
> dig = Digest::MD5.new
> res.read_body { |buf|
> dig.update(buf)
> }
> puts dig.hexdigest
> end
> ```

> end
>
> The above script is also dependent on net/protocol using
> read_nonblock. Ordinary IO objects will need IO#nonblock=true
> to see benefits (because they never hit rb_wait_for_single_fd)
>
> - gc.c (rb_gc_inprogress): new function (rb_gc_step): ditto
> - internal.h: declare prototypes for new gc.c functions
> - thread_pthread.c (gvl_contended_p): new function
> - thread_win32.c (gvl_contended_p): ditto (dummy)
> - thread.c (rb_wait_for_single_fd w/ ppoll): use new functions to perform GC while GVL is uncontended and GC is lazy sweeping or incremental marking [ruby-core:86265] ```
>
> 2 part patch broken out
> https://80x24.org/spew/20180429035007.6499-2-e@80x24.org/raw
> https://80x24.org/spew/20180429035007.6499-3-e@80x24.org/raw
>
> Also on my "sleepy-gc" git branch @ git://80x24.org/ruby.git
>
> ---Files--------------------------------
> sleepy-gc-wip-v1.diff (5.37 KB)
>
> --
> https://bugs.ruby-lang.org/

#2 - 04/29/2018 06:04 AM - normalperson (Eric Wong)

"Atdot.net" ko1@atdot.net wrote:

> Could you give us more detail algorithm?

Pretty simple and I thought the patch was easy-to-read.

Background is we can use ppoll with zero-timeout
({.tv_sec = 0, .tv_nsec = 0 }) to check and return immediately
w/o releasing GVL. This means we can quickly check FD for
readiness.

This is a quick check and even optimized inside the Linux kernel[1].

thread_pthread.c also tracks GVL contention using .waiting field.

I define GC-in-progress as (is_lazy_sweeping || is_incremental_packing)
(same condition for gc.c:gc_rest() function)

Therefore, if GVL is uncontended and GC has work to-do, we use
zero-timeout ppoll and do incremental GC work (incremental mark

- lazy sweep) as long as we need to wait on FD.

If GC is done or if there is GVL contention, we fall back to use
old code path and release GVL.

For do_select case, it might be more expensive because select()
is innefficient for high FDs; but if a process is otherwise not
doing anything, I think it's OK to burn extra cycles to perform
GC sooner.

#3 - 04/29/2018 07:42 AM - normalperson (Eric Wong)

Eric Wong normalperson@yhbt.net wrote:

> This is a quick check and even optimized inside the Linux kernel[1].

Sorry, forgot link:
[1] https://bogomips.org/mirrors/linux.git/tree/fs/select.c?h=v4.16#n851
/* Optimise the no-wait case */

Also, epoll also optimizes for timeout == 0:
https://bogomips.org/mirrors/linux.git/tree/fs/eventpoll.c?h=v4.16#n1754

#4 - 04/29/2018 09:22 AM - normalperson (Eric Wong)

Also, added "thread.c (do_select): perform GC if idle"

https://80x24.org/spew/20180429090250.GA15634@dcvr/raw

And updated "sleepy-gc" git branch @ git://80x24.org/ruby.git
to 10bcc1908601e6f35ebef5ff66476b5cea6da96c.

I'm not sure if native_sleep() is worth doing GC on in most
cases (Mutex#lock, Queue#pop, ...) because that's waiting
on local resources from other threads within our process.

Typical callers of rb_wait_for_single_fd and do_select wait
on external events, so that means our own (Ruby) process is
idle.

I guess Kernel#sleep can do GC work, too (not sure how common it
is to use)

Process.waitpid, File#flock, IO#fcntl(F_SETLKW) are some next
targets where we can try non-blocking operations and GC before
trying blocking equivalents.

Then, the next question is: do we start making all connected
SOCK_STREAM sockets non-blocking by default again? (as in Ruby
1.8)

I'm not sure about nonblock-by-default for pipes,
SOCK_SEQPACKET, and listen sockets; because they have
round-robin behavior which allows fair load distribution across
forked processes.

#5 - 04/29/2018 11:03 AM - normalperson (Eric Wong)

Eric Wong normalperson@yhbt.net wrote:

> I'm not sure if native_sleep() is worth doing GC on in most
> cases (Mutex#lock, Queue#pop, ...) because that's waiting
> on local resources from other threads within our process.

Nevermind, native_sleep benefits from this because local threads
may release GVL in ways which cannot trigger GC from select/ppoll.

Thus we need to rely on their dependent threads (using
Queue#pop, ConditionVariable#wait or similar) to sleep and
trigger GC:

https://80x24.org/spew/20180429105029.GA23412@dcvr/raw

#6 - 04/29/2018 11:50 PM - sam.saffron (Sam Saffron)

I really really like this, its a free performance boost with almost no downsides.

I guess the simplest way of measuring it would be to run something like Discourse bench with and without the patch. In theory we should get better timings after the patch cause it decreases odds that the various GC processes will run when the interpreter wants to run Ruby.

Implementation wise it seems like you only have it on rb_wait_for_single_fd, is there any way you can make this work with the pg gem? It just builds on libpq per: https://www.postgresql.org/docs/8.3/static/libpq-async.html so maybe you would need to expose an end point for libpq to "trigger" partial gc processes just when you send a query?

#7 - 04/30/2018 04:42 AM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> I really really like this, its a free performance boost with
> almost no downsides.

Almost... I need to revisit PATCH 4/2 due to
th->status changes and finalizers running causing compatibility
problems.

> I guess the simplest way of measuring it would be to run
> something like Discourse bench with and without the patch. In

> theory we should get better timings after the patch cause it decreases odds that the various GC processes will run when the interpreter wants to run Ruby.

Depends on benchmark, if a benchmark is pinning things to 100% CPU usage then I expect no improvements. But I don't think real-world network servers are often at 100% CPU use.

> Implementation wise it seems like you only have it on rb_wait_for_single_fd

PATCH 3/2 added select() support, too

> , is there any way you can make this work with the pg gem? It just builds on libpq per: https://www.postgresql.org/docs/8.3/static/libpq-async.html so maybe you would need to expose an end point for libpq to "trigger" partial gc processes just when you send a query?

I'd need to look more deeply, but I recall 'pg' being one of the few gems which worked well with 1.8 threads because FDs were exposed for Ruby to select() on.

So I'm not sure what they're doing these days where they give the Ruby VM no way to distinguish between waiting on external resource (FD) or doing something CPU-intensive locally.

I guess you can cheat for now and do:

```
Thread.new do
  r, w = IO.pipe
  loop { r.wait_readable(0.01) }
end
```

Which will constantly do incremental mark + lazy sweep. But cross-thread free() is probably still bad on most malloc implementations...

If 4/2 worked reliably (tests pass, though...)):

Thread.new { loop { sleep(0.01) } }

(gotta run, back later-ish)

#8 - 05/01/2018 01:22 AM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> Implementation wise it seems like you only have it on rb_wait_for_single_fd, is there any way you can make this work with the pg gem? It just builds on libpq per: https://www.postgresql.org/docs/8.3/static/libpq-async.html so maybe you would need to expose an end point for libpq to "trigger" partial gc processes just when you send a query?

Actually, it seems it seems pg is using rb_thread_fd_select in some places which will benefit from sleep detection, here.

pgconn_block -> wait_socket_readable -> rb_thread_fd_select

So it looks like the PG::Connection#async_exec/async_query/block methods will all hit that. So it looks like PG users can automatically benefit from this work (as well as some of the auto-fiber stuff).

That said, it looks like they're using rb_thread_fd_select on a single FD, and Linux users would be better off if they used rb_wait_for_single_fd instead. The latter has been optimized for Linux since 1.9.3 to avoid malloc and O(n) behavior based on FD number.

#9 - 05/01/2018 02:31 AM - ko1 (Koichi Sasada)

My concerns are:

(1) Full GC (like GC.start) or step incremental marking/sweeping (to guarantee (or to reduce) the worst stopping time because of GC for every IO operation).
(2) How to know GC is required (if we invoke GC.start on any I/O (blocking) operations, it should be harmful)

Seeing your comment #2,

> Therefore, if GVL is uncontended and GC has work to-do, we use
> zero-timeout ppoll and do incremental GC work (incremental mark
>
> - lazy sweep) as long as we need to wait on FD.

they should be:

(1) do a step for incremental marking/sweeping
(2) only when incremental marking or sweeping

They are very reasonable for me.

My understanding, your proposal in pseudo code is (pls correct me if it is wrong):

```
def io_operation
  while true
    if !GVL.contended? && GC.has_incremental_task?
      if result = io_operation(timeout: 0) > 0 # timeout = 0 means return immediately
        # There are result
        return result
       else
        GC.do_step
       end
    else
      GVL.release
      io_operation(timeout = long time)
      GVL.acquire
    end
  end
end
```

No problem for me.

However, your code https://80x24.org/spew/20180429035007.6499-3-e@80x24.org/raw

```
+int
+rb_gc_step(const rb_execution_context_t *ec)
+{
+    rb_objspace_t *objspace = rb_ec_vm_ptr(ec)->objspace;
+
+    gc_rest(objspace);
+
+    return rb_gc_inprogress(ec);
+}
```

gc_rest() do all of rest steps. Is it intentional?

Another tiny comments:

- static const struct timespec zero;

zero doesn't seem to be initialized. intentional?

Note:

After introducing Guild, getting contended status should be high-cost (we need to use lock to see this info).
However, we can eliminate this check if we shrink the target: only have one Guild (== current MRI).

#10 - 05/01/2018 03:22 AM - normalperson (Eric Wong)

ko1@atdot.net wrote:

> My understanding, your proposal in pseudo code is (pls correct me if it is wrong):

Correct.

> gc_rest() do all of rest steps. Is it intentional?

I thought about that myself. I haven't measured impact much and
decided to have less code.

We can also try the following to favor sweep before mark:

```
--- a/gc.c
+++ b/gc.c
@@ -6534,7 +6534,14 @@ rb_gc_step(const rb_execution_context_t *ec)
{
rb_objspace_t *objspace = rb_ec_vm_ptr(ec)->objspace;

-    gc_rest(objspace);
+    if (is_lazy_sweeping(heap_eden)) {
+        gc_sweep_rest(objspace);
+    }
+    else if (is_incremental_marking(objspace)) {
+        PUSH_MARK_FUNC_DATA(NULL);
+        gc_marks_rest(objspace);
+        POP_MARK_FUNC_DATA();
+    }

return rb_gc_inprogress(ec);
}
```

> Another tiny comments:
>
> - static const struct timespec zero;
>
> zero doesn't seem to be initialized. intentional?

Yes, static and global are variables are auto-initialized to zero.
AFAIK this is true of all C compilers.

> Note:
>
> After introducing Guild, getting contended status should be high-cost (we need to use lock to see this info).
> However, we can eliminate this check if we shrink the target: only have one Guild (== current MRI).

So one objspace will be shared by different guilds?

We may use atomics to check. I think sweep phase can be made
lock-free in the future.

Originally I wanted to make sweep lock-free before making this
patch, but it seems unnecessary at the moment.

#11 - 05/01/2018 03:33 AM - ko1 (Koichi Sasada)

On 2018/05/01 12:18, Eric Wong wrote:

> > gc_rest() do all of rest steps. Is it intentional?
>
> I thought about that myself. I haven't measured impact much and
> decided to have less code.

On worst case, it takes few seconds. We have "incremental" mechanism so
we should use same incremental technique here too.

> > Another tiny comments:
> >
> > - static const struct timespec zero;
>
> zero doesn't seem to be initialized. intentional?

> Yes, static and global are variables are auto-initialized to zero.

> AFAIK this is true of all C compilers.

Sorry, I missed static const. Thank you.

> > Note:
> >
> > After introducing Guild, getting contended status should be high-cost (we need to use lock to see this info).
> > However, we can eliminate this check if we shrink the target: only have one Guild (== current MRI).
>
> So one objspace will be shared by different guilds?

Yes.

> I think sweep phase can be made
> lock-free in the future.

Agreed.

--
// SASADA Koichi at atdot dot net

#12 - 05/01/2018 03:52 AM - normalperson (Eric Wong)

Koichi Sasada ko1@atdot.net wrote:

> On 2018/05/01 12:18, Eric Wong wrote:
>
> > > gc_rest() do all of rest steps. Is it intentional?
> >
> > I thought about that myself. I haven't measured impact much and
> > decided to have less code.
>
> On worst case, it takes few seconds. We have "incremental" mechanism so we
> should use same incremental technique here too.

Oh sorry, I realize I was using the wrong gc.c functions :x
Something like:

```
--- a/gc.c
+++ b/gc.c
@@ -6533,8 +6533,12 @@ int
rb_gc_step(const rb_execution_context_t *ec)
{
rb_objspace_t *objspace = rb_ec_vm_ptr(ec)->objspace;
-
-    gc_rest(objspace);
+    if (is_lazy_sweeping(&objspace->eden_heap)) {
+  gc_sweep_step(objspace, &objspace->eden_heap);
+    }
+    else if (is_incremental_marking(objspace)) {
+        /* FIXME TODO */
+    }

return rb_gc_inprogress(ec);
}
```

I haven't looked at incremental mark, yet :x

#13 - 05/01/2018 03:52 AM - ko1 (Koichi Sasada)

On 2018/05/01 12:47, Eric Wong wrote:

> Oh sorry, I realize I was using the wrong gc.c functions :x
> Something like:

Thank you. No problem.

More performance check will be great (to write a NEWS entry :))

--
// SASADA Koichi at atdot dot net

#14 - 05/01/2018 08:52 AM - normalperson (Eric Wong)

Koichi Sasada ko1@atdot.net wrote:

> On 2018/05/01 12:47, Eric Wong wrote:
>
> > Oh sorry, I realize I was using the wrong gc.c functions :x
> > Something like:
>
> Thank you. No problem.
>
> More performance check will be great (to write a NEWS entry :))

I have some folks interested in backport for 2.4 and 2.5.
Much of the code I write uses String#clear and other techniques
to reduce memory too aggressively to benefit.
I can make some patches to benchmark/ from existing examples
in commit messages.

Anyways v2 of the series is available:

The following changes since commit 41f4ac6aa21588722a6323dbbc34274b7e9aec49:

ast.c: use enum in switch for warnings (2018-05-01 06:55:43 +0000)

are available in the Git repository at:

git://80x24.org/ruby.git sleepy-gc-v2

for you to fetch changes up to 9d1609d318821b11614da6f952acadf7d3a3e083:

thread.c: native_sleep callers may perform GC (2018-05-01 07:57:21 +0000)

v2 updates:

- [PATCH 2/4] uses correct functions for incremental work
- [PATCH 3/4] accounts for select(2) clobbering its timeval arg
- [PATCH 4/4] totally redone; native_sleep callers are all rather complex and it can be improved in future patches

---

Eric Wong (4):
thread.c (timeout_prepare): common function
gc: rb_wait_for_single_fd performs GC if idle (Linux)
thread.c (do_select): perform GC if idle
thread.c: native_sleep callers may perform GC

Individual patches available at:
https://80x24.org/spew/20180501080844.22751-2-e@80x24.org/raw
https://80x24.org/spew/20180501080844.22751-3-e@80x24.org/raw
https://80x24.org/spew/20180501080844.22751-4-e@80x24.org/raw
https://80x24.org/spew/20180501080844.22751-5-e@80x24.org/raw

Also have a Tor .onion mirror if https://80x24.org/ breaks again:
http://hjrcffqmbrq6wope.onion/spew/20180501080844.22751-2-e@80x24.org/raw
http://hjrcffqmbrq6wope.onion/spew/20180501080844.22751-3-e@80x24.org/raw
http://hjrcffqmbrq6wope.onion/spew/20180501080844.22751-4-e@80x24.org/raw
http://hjrcffqmbrq6wope.onion/spew/20180501080844.22751-5-e@80x24.org/raw

#15 - 05/02/2018 02:22 AM - ko1 (Koichi Sasada)

On 2018/05/01 17:46, Eric Wong wrote:

> Individual patches available at:
> https://80x24.org/spew/20180501080844.22751-2-e@80x24.org/raw
> https://80x24.org/spew/20180501080844.22751-3-e@80x24.org/raw
> https://80x24.org/spew/20180501080844.22751-4-e@80x24.org/raw
> https://80x24.org/spew/20180501080844.22751-5-e@80x24.org/raw

I'm not sure how to see all of diffs in one patch. Do you have?

Anyway, small comments:

> https://80x24.org/spew/20180501080844.22751-3-e@80x24.org/raw
>
> - /* TODO: should this check is_incremental_marking() ? */

Any problem to check it?

> +rb_gc_step(const rb_execution_context_t *ec)

How about to add assertion that rb_gc_inprogress() returns true?

```
--- a/internal.h
+++ b/internal.h
@@ -1290,6 +1290,10 @@ void rb_gc_writebarrier_remember(VALUE obj);
void ruby_gc_set_params(int safe_level);
void rb_copy_wb_protected_attribute(VALUE dest, VALUE obj);

+struct rb_execution_context_struct;
+int rb_gc_inprogress(const struct rb_execution_context_struct *);
+int rb_gc_step(const struct rb_execution_context_struct *);
+
```

How about to add them into gc.h?

https://80x24.org/spew/20180501080844.22751-4-e@80x24.org/raw

I have no enough knowledge to review it.
Nobu?

https://80x24.org/spew/20180501080844.22751-5-e@80x24.org/raw

> @@ -288,8 +294,17 @@ rb_mutex_lock(VALUE self)

I can't understand why GC at acquiring (and restarting) timing is
needed. Why?

For other functions, I have a same question.happen.

--
// SASADA Koichi at atdot dot net

#16 - 05/02/2018 02:52 AM - normalperson (Eric Wong)

Koichi Sasada ko1@atdot.net wrote:

> On 2018/05/01 17:46, Eric Wong wrote:
>
> > Individual patches available at:
> > https://80x24.org/spew/20180501080844.22751-2-e@80x24.org/raw
> > https://80x24.org/spew/20180501080844.22751-3-e@80x24.org/raw
> > https://80x24.org/spew/20180501080844.22751-4-e@80x24.org/raw
> > https://80x24.org/spew/20180501080844.22751-5-e@80x24.org/raw
>
> I'm not sure how to see all of diffs in one patch. Do you have?

I fetch and run "git diff" locally which gives me many options

```
REMOTE=80x24
git remote add $REMOTE git://80x24.org/ruby.git
git fetch $REMOTE
git diff $OLD $NEW
```

$OLD and $NEW are commits which "git request-pull" outputs in my previous
emails:

```
> The following changes since commit $OLD
```

```
>
>   $OLD_SUBJECT
>
> are available in the Git repository at:
>
>   git://80x24.org/ruby.git BRANCH
>
> for you to fetch changes up to $NEW
```

You can also:

```
curl https://80x24.org/spew/20180501080844.22751-2-e@80x24.org/raw \
```

https://80x24.org/spew/20180501080844.22751-3-e@80x24.org/raw \
https://80x24.org/spew/20180501080844.22751-4-e@80x24.org/raw \
https://80x24.org/spew/20180501080844.22751-5-e@80x24.org/raw \
| git am

(I run scripts from my $EDITOR and mail client, of course :)

> Anyway, small comments:
>
> > https://80x24.org/spew/20180501080844.22751-3-e@80x24.org/raw
> >
> > - /* TODO: should this check is_incremental_marking() ? */
>
> Any problem to check it?

Probably no problem, old comment. I originally only intended to
do lazy-sweep since I have not studied incremental marking,
much.

> > +rb_gc_step(const rb_execution_context_t *ec)
>
> How about to add assertion that rb_gc_inprogress() returns true?

I don't think that's safe. For native_sleep callers; we release
GVL after calling rb_gc_step; so sometimes rb_gc_step becomes
a no-op (because other thread took GVL and did GC).

> --- a/internal.h
> +++ b/internal.h
> @@ -1290,6 +1290,10 @@ void rb_gc_writebarrier_remember(VALUE obj);
> void ruby_gc_set_params(int safe_level);
> void rb_copy_wb_protected_attribute(VALUE dest, VALUE obj);
>
> +struct rb_execution_context_struct;
> +int rb_gc_inprogress(const struct rb_execution_context_struct *);
> +int rb_gc_step(const struct rb_execution_context_struct *);
> +
>
> How about to add them into gc.h?

Sure.

> https://80x24.org/spew/20180501080844.22751-4-e@80x24.org/raw
>
> I have no enough knowledge to review it.
> Nobu?
>
> https://80x24.org/spew/20180501080844.22751-5-e@80x24.org/raw
>
> > @@ -288,8 +294,17 @@ rb_mutex_lock(VALUE self)
>
> I can't understand why GC at acquiring (and restarting) timing is needed.
> Why?
>
> For other functions, I have a same question.happen.

For mutex_lock, it only does GC if it can't acquire immediately.
Since mutex_lock cannot proceed, it can probably do GC.

I release GVL at mutex_lock before GC since it needs to give
the other thread a chance to release the mutex.

One problem I have now is threads in THREAD_STOPPED_FOREVER
state cannot continuously perform GC if some other thread
is constantly making garbage and never sleeping.

```
nr = 100_000
th = Thread.new do
File.open('/dev/urandom') do |rd|
nr.times { rd.read(16384) }
end
end

# no improvement, since it enters sleep and stays there
th.join

# instead, this works (but wastes battery if there's no garbage)
true until th.join(0.01)
```

So maybe we add heuristics for entering sleep for methods in
thread.c and thread_sync.c and possibly continuing to schedule
threads in THREAD_STOPPED_FOREVER state to enable them to
perform cleanup. I don't think this is urgent, and we can
ignore this case for now.

#17 - 05/02/2018 03:52 AM - ko1 (Koichi Sasada)

On 2018/05/02 11:49, Eric Wong wrote:

> I fetch and run "git diff" locally which gives me many options
>
> ```
> REMOTE=80x24
> git remote add $REMOTE git://80x24.org/ruby.git
> git fetch $REMOTE
> git diff $OLD $NEW
> ```
>
> $OLD and $NEW are commits which "git request-pull" outputs in my previous
> emails:
>
> > The following changes since commit $OLD
> >
> > $OLD_SUBJECT
> >
> > are available in the Git repository at:
> >
> > git://80x24.org/ruby.git BRANCH
> >
> > for you to fetch changes up to $NEW
>
> You can also:
>
> ```
> curl https://80x24.org/spew/20180501080844.22751-2-e@80x24.org/raw \
> https://80x24.org/spew/20180501080844.22751-3-e@80x24.org/raw \
> https://80x24.org/spew/20180501080844.22751-4-e@80x24.org/raw \
> https://80x24.org/spew/20180501080844.22751-5-e@80x24.org/raw \
> | git am
> ```
>
> (I run scripts from my $EDITOR and mail client, of course :)

Great. Thank you!

> > +rb_gc_step(const rb_execution_context_t *ec)
> >
> > How about to add assertion that rb_gc_inprogress() returns true?
>
> I don't think that's safe. For native_sleep callers; we release

> GVL after calling rb_gc_step; so sometimes rb_gc_step becomes
> a no-op (because other thread took GVL and did GC).

OK. I assumed that this "step" API is used with "rb_gc_inprogress()".
But it is not correct.

> > https://80x24.org/spew/20180501080844.22751-5-e@80x24.org/raw
> >
> > @@ -288,8 +294,17 @@ rb_mutex_lock(VALUE self)
> >
> > I can't understand why GC at acquiring (and restarting) timing is needed.
> > Why?
> >
> > For other functions, I have a same question.happen.
>
> For mutex_lock, it only does GC if it can't acquire immediately.
> Since mutex_lock cannot proceed, it can probably do GC.

```
+       if (mutex->th == th) {
+       mutex_locked(th, self);
+       }
+       if (do_gc) {
+       /*
+        * Likely no point in checking for GVL contention here
+        * this Mutex is already contended and we just yielded
+        * above.
+        */
+       do_gc = rb_gc_step(th->ec);
+       }
```

it should be else if (do_gc), isn't?

> One problem I have now is threads in THREAD_STOPPED_FOREVER
> state cannot continuously perform GC if some other thread
> is constantly making garbage and never sleeping.
>
> ```
> nr = 100_000
> th = Thread.new do
>   File.open('/dev/urandom') do |rd|
>     nr.times { rd.read(16384) }
>   end
> end
>
> # no improvement, since it enters sleep and stays there
> th.join
>
> # instead, this works (but wastes battery if there's no garbage)
> true until th.join(0.01)
> ```

I'm not sure why it is a problem. Created thread do read and it can GC
incrementally, or if read return immediately, there are no need to
step more GC (usual GC should be enough), especially for throughput.

> So maybe we add heuristics for entering sleep for methods in
> thread.c and thread_sync.c and possibly continuing to schedule
> threads in THREAD_STOPPED_FOREVER state to enable them to
> perform cleanup. I don't think this is urgent, and we can
> ignore this case for now.

"cleanup"? do GC steps? I agree on them (requirements and immediacy).

--
// SASADA Koichi at atdot dot net

#18 - 05/02/2018 04:12 AM - normalperson (Eric Wong)

Koichi Sasada ko1@atdot.net wrote:

> OK. I assumed that this "step" API is used with "rb_gc_inprogress()". But it

> is not correct.

Right, inprogress is only a hint. I will add a comment to that effect.
As with ppoll + read, there is always a chance of work being "stolen"
by other threads :)

```
+         if (mutex->th == th) {
+         mutex_locked(th, self);
+         }
+         if (do_gc) {
+         /*
+          * Likely no point in checking for GVL contention here
+          * this Mutex is already contended and we just yielded
+          * above.
+          */
+         do_gc = rb_gc_step(th->ec);
+         }
```

> it should be else if (do_gc), isn't?

Yes, I will fix.

> > One problem I have now is threads in THREAD_STOPPED_FOREVER
> > state cannot continuously perform GC if some other thread
> > is constantly making garbage and never sleeping.
>
> ```
> nr = 100_000
> th = Thread.new do
>   File.open('/dev/urandom') do |rd|
>     nr.times { rd.read(16384) }
>   end
> end
>
> # no improvement, since it enters sleep and stays there
> th.join
>
> # instead, this works (but wastes battery if there's no garbage)
> true until th.join(0.01)
> ```
>
> I'm not sure why it is a problem. Created thread do read and it can GC
> incrementally, or if read return immediately, there are no need to step
> more GC (usual GC should be enough), especially for throughput.

I suppose.... Note: read on urandom won't hit rb_wait_for_single_fd
to trigger GC(*), but it will only trigger GC via string allocation.

(*) /dev/urandom can't return with EAGAIN, only /dev/random can

> > So maybe we add heuristics for entering sleep for methods in
> > thread.c and thread_sync.c and possibly continuing to schedule
> > threads in THREAD_STOPPED_FOREVER state to enable them to
> > perform cleanup. I don't think this is urgent, and we can
> > ignore this case for now.
>
> "cleanup"? do GC steps? I agree on them (requirements and immediacy).

Sure. Should I commit after adding "else" to mutex_lock?

#19 - 05/02/2018 04:22 AM - ko1 (Koichi Sasada)

On 2018/05/02 13:08, Eric Wong wrote:

> Sure. Should I commit after adding "else" to mutex_lock?

I want to ask to introduce "disable" macro (like USE_RGENGC) to measure
impact on this technique (and disable to separate issues). Please name
it as your favorite.

Thanks,
Koichi

--
// SASADA Koichi at atdot dot net

#20 - 05/02/2018 05:04 AM - normalperson (Eric Wong)

Koichi Sasada ko1@atdot.net wrote:

> On 2018/05/02 13:08, Eric Wong wrote:
>
> > Sure. Should I commit after adding "else" to mutex_lock?
>
> I want to ask to introduce "disable" macro (like USE_RGENGC) to measure
> impact on this technique (and disable to separate issues). Please name it as
> your favorite.

OK, I added RUBY_GC_SLEEPY_SWEEP and RUBY_GC_SLEEPY_MARK macros:

[PATCH 6/4] gc.c: allow disabling sleepy GC
https://80x24.org/spew/20180502045248.GA3949@80x24.org/raw

And missing "else":

[PATCH 5/4] thread_sync.c (mutex_lock): add missing else
https://80x24.org/spew/20180502044255.GA30679@80x24.org/raw

I also added some benchmarks, but I'm not sure if dependency on
/dev/urandom is good because performance across machines and
kernel configuration can be very different.

https://80x24.org/spew/20180502045714.GA5427@whir/raw

I need something which:
a) doesn't compete for GVL
b) takes a while

Perhaps depending on fork() is fine, since it's just as
unportable as /dev/urandom is.

#21 - 05/02/2018 05:19 AM - sam.saffron (Sam Saffron)

I can confirm this has a MAJOR benefit for particular workloads with the pg gem. In particular if you are using async_exec (which most of us should)

```
require 'pg'
require 'benchmark/ips'

$conn = PG.connect(dbname: 'postgres')


Benchmark.ips do |b|
  b.config(time: 10, warmup: 3)

  b.report("exec") do
    $conn.exec("SELECT generate_series(1,10000)").to_a
  end
  b.report("async exec") do
    $conn.async_exec("SELECT generate_series(1,10000)").to_a
  end
end
```

Before:

```
sam@ubuntu pg_perf % ruby test.rb
Warming up --------------------------------------
                exec    20.000  i/100ms
          async exec    21.000  i/100ms
Calculating -------------------------------------
                exec    212.760  (± 1.4%) i/s -      2.140k in  10.060122s
          async exec    214.570  (± 1.9%) i/s -      2.163k in  10.084992s
sam@ubuntu pg_perf % ruby test.rb
Warming up --------------------------------------
```

```
                exec    19.000  i/100ms
          async exec    20.000  i/100ms
Calculating -------------------------------------
                exec    202.603  (± 5.9%) i/s -      2.033k in  10.072578s
          async exec    201.516  (± 6.0%) i/s -      2.020k in  10.062116s
```

After:

```
sam@ubuntu pg_perf % ruby test.rb
Warming up --------------------------------------
                exec    21.000  i/100ms
          async exec    23.000  i/100ms
Calculating -------------------------------------
                exec    211.320  (± 2.8%) i/s -      2.121k in  10.044445s
          async exec    240.188  (± 1.7%) i/s -      2.415k in  10.057509s
sam@ubuntu pg_perf % ruby test.rb
Warming up --------------------------------------
                exec    20.000  i/100ms
          async exec    23.000  i/100ms
Calculating -------------------------------------
                exec    209.644  (± 1.4%) i/s -      2.100k in  10.018850s
          async exec    237.100  (± 2.1%) i/s -      2.392k in  10.092435s
```

So this moves us from 200-210 ops/s to 240 ops/s. This is a major perf boost, still to see if it holds on the full Discourse bench, but I expect major improvements cause waiting for SQL is very very very common in web apps.

I do not expect too much benefit in concurrent puma workloads, but for us in unicorn we should have a pretty nice boost.

#22 - 05/02/2018 06:33 AM - ko1 (Koichi Sasada)

On 2018/05/02 14:00, Eric Wong wrote:

> I also added some benchmarks, but I'm not sure if dependency on
> /dev/urandom is good because performance across machines and
> kernel configuration can be very different.

Sam's report?
Sam, could you try discourse benchmark?

I'm not sure pg test on [ruby-core:86820] is suitable or not.

--
// SASADA Koichi at atdot dot net

#23 - 05/02/2018 08:22 AM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> require 'benchmark/ips'
>
> So this moves us from 200-210 ops/s to 240 ops/s. This is a
> major perf boost, still to see if it holds on the full
> Discourse bench, but I expect major improvements cause waiting
> for SQL is very very very common in web apps.

Thanks! I wasn't even aiming for a speed improvement.
Any memory measurements?
I guess benchmark/ips won't show that.

> I do not expect too much benefit in concurrent puma workloads,
> but for us in unicorn we should have a pretty nice boost.

It really depends on CPU usage, I don't think it's common for
any server to be using all available CPU at all times; so
Ruby should be able to get background work done during
wait states.

One difference in MT is cross-thread malloc/free (malloc returns
a pointer to be freed in another thread) isn't too great in most
malloc implementations I've studied.

Though Ruby sometimes hits cross-thread malloc with or without

sleepy GC, it may be more common with sleepy GC. Before sleepy GC, free() happens most in threads which malloc() most, so it gets returned to the correct arena/cache most often.

Haven't checked jemalloc in a while, but I remember cross-thread was weak there in the 4.x days; maybe it's improved. glibc wasn't terrible there and (I think it was) DJ Delorie was taking it into account in his updates; but I haven't kept up with that work. Forcing fewer arenas via MALLOC_ARENA_MAX also mitigates this problem.

I seem to recall Lockless Inc. malloc being REALLY good at cross-thread malloc/free, but used too much memory overall in my experience. cross-thread malloc/free can be a common pattern for message-passing systems

Anyways, maybe this will encourage me to try getting wfcqueue into glibc malloc as I threatened to do in ruby-core:86731

#24 - 05/03/2018 03:12 AM - noahgibbs (Noah Gibbs)

I checked today's head-of-master with Rails Ruby Bench. The first run suggests a noticeable drop in performance between 2.6 preview 1 and head-of-master. It's not guaranteed that the drop is because of this change. I'll try to repro with more runs first, then see if this change seems responsible -- could be something else in 2.6. But I'm seeing a drop in RRB throughput from around 179 req/sec to around 170 req/sec, and a significant increase in variance between runs (from about 2.6 to about 11.9). This is with only 20 runs, though. I'll definitely get a lot more datapoints before I'm sure. But it's a large enough drop that it's probably *not* random noise.

#25 - 05/03/2018 03:17 AM - noahgibbs (Noah Gibbs)

Ah, never mind. It looks like the Ruby I tested doesn't have the sleepy GC changes! So it's slower, but that's not the fault of this patch. Great. I'll check this patch against that as a baseline.

#26 - 05/03/2018 05:43 AM - sam.saffron (Sam Saffron)

From my testing on Discourse bench ... the difference is pretty much not that measurable

Before patch

```
Unicorn: (workers: 3)
Include env: false
Iterations: 200, Best of: 1
Concurrency: 1

---
categories:
  50: 58
  75: 65
  90: 73
  99: 123
home:
  50: 62
  75: 70
  90: 86
  99: 139
topic:
  50: 60
  75: 65
  90: 72
  99: 117
categories_admin:
  50: 101
  75: 106
  90: 115
  99: 210
home_admin:
  50: 107
  75: 114
  90: 132
  99: 211
topic_admin:
  50: 115
  75: 123
  90: 134
  99: 201
timings:
```

```
  load_rails: 5444
ruby-version: 2.6.0-p-1
rss_kb: 196444
pss_kb: 139514
memorysize: 7.79 GB
virtual: vmware
architecture: amd64
operatingsystem: Ubuntu
processor0: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
physicalprocessorcount: 2
kernelversion: 4.15.0
rss_kb_23779: 309984
pss_kb_23779: 249785
rss_kb_23817: 307056
pss_kb_23817: 246738
rss_kb_23948: 304732
pss_kb_23948: 244364
```

After patch:

```
Iterations: 200, Best of: 1
Concurrency: 1

---
categories:
  50: 56
  75: 61
  90: 70
  99: 116
home:
  50: 63
  75: 70
  90: 77
  99: 170
topic:
  50: 61
  75: 68
  90: 77
  99: 96
categories_admin:
  50: 102
  75: 111
  90: 121
  99: 182
home_admin:
  50: 96
  75: 102
  90: 108
  99: 205
topic_admin:
  50: 109
  75: 118
  90: 130
  99: 192
timings:
  load_rails: 4987
ruby-version: 2.6.0-p-1
rss_kb: 196004
pss_kb: 137541
memorysize: 7.79 GB
virtual: vmware
architecture: amd64
operatingsystem: Ubuntu
processor0: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
physicalprocessorcount: 2
kernelversion: 4.15.0
rss_kb_16393: 306312
pss_kb_16393: 244353
rss_kb_16438: 307052
pss_kb_16438: 244942
rss_kb_16555: 305092
pss_kb_16555: 242997
```

Nothing really sticks out as absolutely an improvement across the board though some of the benches are a bit faster, memory is almost not impacted. It is no worse than head, but it is also not easy to measure how much better it is, we may need to repeat with significantly more iterations to remove

noise.

I do want to review Discourse carefully to ensure we are using async_exec everywhere... will do so later today.

Eric if you feel like trying out the bench, clone: https://github.com/discourse/discourse.git and run ruby script/bench.rb

I also have some allocator benches you can play with at: https://github.com/SamSaffron/allocator_bench.git

#27 - 05/03/2018 06:07 AM - sam.saffron (Sam Saffron)

I found one place where we were not using async_exec so I changed it to use async_exec... this is revised numbers:

Pre patch:

```
categories:
  50: 53
  75: 59
  90: 63
  99: 76
home:
  50: 57
  75: 64
  90: 68
  99: 136
topic:
  50: 58
  75: 61
  90: 68
  99: 110
categories_admin:
  50: 96
  75: 102
  90: 108
  99: 184
home_admin:
  50: 104
  75: 112
  90: 122
  99: 213
topic_admin:
  50: 115
  75: 121
  90: 139
  99: 184
timings:
  load_rails: 4936
ruby-version: 2.6.0-p-1
rss_kb: 193500
pss_kb: 134214
memorysize: 7.79 GB
virtual: vmware
architecture: amd64
operatingsystem: Ubuntu
processor0: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
physicalprocessorcount: 2
kernelversion: 4.15.0
rss_kb_21961: 305616
pss_kb_21961: 243099
rss_kb_22009: 304644
pss_kb_22009: 241972
rss_kb_22133: 304108
pss_kb_22133: 241388
```

Post patch:

```
Your Results: (note for timings- percentile is first, duration is second in millisecs)
Unicorn: (workers: 3)
Include env: false
Iterations: 200, Best of: 1
Concurrency: 1

---
categories:
  50: 54
  75: 59
```

```
  90: 66
  99: 84
home:
  50: 57
  75: 62
  90: 65
  99: 139
topic:
  50: 56
  75: 61
  90: 67
  99: 104
categories_admin:
  50: 95
  75: 99
  90: 106
  99: 179
home_admin:
  50: 99
  75: 103
  90: 106
  99: 195
topic_admin:
  50: 109
  75: 114
  90: 118
  99: 163
timings:
  load_rails: 4851
ruby-version: 2.6.0-p-1
rss_kb: 195164
pss_kb: 136384
memorysize: 7.79 GB
virtual: vmware
architecture: amd64
operatingsystem: Ubuntu
processor0: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
physicalprocessorcount: 2
kernelversion: 4.15.0
rss_kb_19222: 305328
pss_kb_19222: 243213
rss_kb_19267: 303188
pss_kb_19267: 240952
rss_kb_19384: 307992
pss_kb_19384: 245778
```

perf change seems hard to pin down properly.

#28 - 05/03/2018 07:13 AM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> perf change seems a tiny bit more noticable.

Thanks for benchmarking! Disappointing results, though.

Is this is with my latest updates up thread with do_select
and gc_*_continue functions?

Can you try #define RUBY_GC_SLEEPY_MARK 0 in gc.h to disable
incremental marking on sleep?

I wonder if incremental marking is causing too many objects
to be marked when it is triggered deep in the stack.

Marking is best done when the stack is shallow (where unicorn
calls IO.select), but could be harmful when the stack is deep
(where Pg calls rb_thread_fd_select).

#29 - 05/04/2018 09:32 PM - normalperson (Eric Wong)

Eric Wong normalperson@yhbt.net wrote:

> Marking is best done when the stack is shallow (where unicorn

> calls IO.select), but could be harmful when the stack is deep
> (where Pg calls rb_thread_fd_select).

Also, I think we need to start GC if no sweeping/marking is
inprogress.

#30 - 05/05/2018 07:02 AM - larskanis (Lars Kanis)

I updated pg to use rb_wait_for_single_fd() instead of rb_thread_fd_select(). The change is already on the master branch: https://github.com/ged/ruby-pg . However although the speedup is measurable in micro benchmarks, it is not within a rails context.

In pg all IO bound methods release the GVL, but not all methods use rb_wait_for_single_fd() or rb_thread_fd_select() to wait for server answers. Only methods of the async API do this. This is why I proposed a change to rails, to make use of the async API only: https://github.com/rails/rails/pull/32820

Hope that helps...

#31 - 05/05/2018 02:53 PM - noahgibbs (Noah Gibbs)

For Rails Ruby Bench (large concurrent Rails benchmark based on Discourse), measuring sleepy-gc-v3 branch versus the previous commit, the difference isn't measurable. No detectable speedup. The sleepy-gc batch of runs has a higher variance in runtime, but that may just be an outlier or two - I'd need a lot more samples to see if it consistently gives higher variance. The variance is often randomly a bit different batch-to-batch.

#32 - 05/06/2018 03:33 AM - normalperson (Eric Wong)

the.codefolio.guy@gmail.com wrote:

> For Rails Ruby Bench (large concurrent Rails benchmark based
> on Discourse),

So multithreaded? Do you have any info on the amount of CPU
time was being used without these changes?

If the CPU usage was already 100% or close before the patch,
then I'd expect no benefit.

So yeah, for benchmarking, I would mainly expect it to show up
more in single-threaded benchmarks.

But for practical use outside of benchmarks, I think there'll be
a benefit in all <100% CPU usage scenarios (which is typical
of real-world traffic, but not benchmarks).

> measuring sleepy-gc-v3 branch versus the
> previous commit, the difference isn't measurable. No
> detectable speedup. The sleepy-gc batch of runs has a higher
> variance in runtime, but that may just be an outlier or two -
> I'd need a lot more samples to see if it consistently gives
> higher variance. The variance is often randomly a bit
> different batch-to-batch.

The variance might have something to do with the malloc and
settings used (arena count), especially when multithreaded.
(see what I wrote previously about cross-thread malloc/free).

I experimented with some GC-start-on-sleep the other day,
but didn't get very far as far as having a small reproducible
benchmark case.

If anybody wants to give me SSH access to a machine they run
100% Free Software benchmarks on, my public key has always been
here:

```
https://yhbt.net/id_rsa.pub
I will only use a terminal for Ruby development, no GUIs.
```

Thanks (also won't be around computers much for another day or two)

#33 - 05/07/2018 08:12 AM - ko1 (Koichi Sasada)

On 2018/05/05 6:32, Eric Wong wrote:

> Also, I think we need to start GC if no sweeping/marking is

> inprogress.

This is a problem we need to discuss.

Good: It can increase GC cleaning without additional overhead.

Bad1: However if we kick unnecessary GCs it should be huge penalty.
Bad2: Also if we run multiple Ruby processes, it can be system's
overhead which consumes CPU resources which other process can run.

--
// SASADA Koichi at atdot dot net

#34 - 05/07/2018 09:52 AM - normalperson (Eric Wong)

Koichi Sasada ko1@atdot.net wrote:

> On 2018/05/05 6:32, Eric Wong wrote:
>
> > Also, I think we need to start GC if no sweeping/marking is
> > inprogress.
>
> This is a problem we need to discuss.
>
> Good: It can increase GC cleaning without additional overhead.
>
> Bad1: However if we kick unnecessary GCs it should be huge penalty.

Right. Minor GC is still expensive, I wonder if we can make it
cheaper or semi-incremental. It can be incremental until next
newobj_of happens, at which point newobj_of must finish the
minor GC immediately. This may help some IO cases if object
creation can be avoided.

For tracking GC statistics, we should probably keep them in
rb_execution_context_t instead of current globals using atomics.
To recover the most memory from GC, we want to do gc_mark_roots

1) from the ec with the most allocations
2) when it is at the shallowest stack point

This is tricky in MT situations :<

> Bad2: Also if we run multiple Ruby processes, it can be system's overhead
> which consumes CPU resources which other process can run.

I hope this feature can reduce the use extra processes, even.
In other words, instead of having an N:1 process:core ratio, it
could become (N/2):1 or something.

Now I need sleep myself :<

#35 - 05/07/2018 03:15 PM - noahgibbs (Noah Gibbs)

normalperson (Eric Wong) wrote:

> So multithreaded? Do you have any info on the amount of CPU
> time was being used without these changes?

Highly multithreaded. Normally the CPU usage stays at nearly 100%. So I agree, this is not a great benchmark to show the benefit. The main result is that it didn't slow it down :-)

> The variance might have something to do with the malloc and
> settings used (arena count), especially when multithreaded.
> (see what I wrote previously about cross-thread malloc/free).

Yeah. I'll need to run the benchmark a lot of times to be sure. It's not a large effect, if it's real.

#36 - 05/14/2018 08:42 PM - normalperson (Eric Wong)

I wrote:

> For tracking GC statistics, we should probably keep them in
> rb_execution_context_t instead of current globals using atomics.
> To recover the most memory from GC, we want to do gc_mark_roots

That's maybe too complex for now, this patch (on top of existing
sleepy GC):

https://80x24.org/spew/20180514201509.28069-1-e@80x24.org/raw

While the effect on big Rails apps seems minimal, I think the
significant improvements for small scripts is still helpful and
we can build on top of them. I am already satisfied with the
improvement from a Net::HTTP example from the first patch:

https://80x24.org/spew/20180501080844.22751-3-e@80x24.org/raw

Since all new behavior changes can be easily disabled via gc.h,
I propose we commit the current changes to trunk for now to
gain more testing and feedback.

Current series is up to 8 patches, but I will squash
"thread_sync.c (mutex_lock): add missing else" into
"thread.c: native_sleep callers may perform GC".

The following changes since commit 6f0de6ed98e669e915455569fb4dae9022cb47b8:

error.c: check redefined backtrace result (2018-05-14 08:33:14 +0000)

are available in the Git repository at:

git://80x24.org/ruby.git sleepy-gc-v6

for you to fetch changes up to 6944014696bea793603d47db6dba0a1e83f1e430:

gc.c: enter sleepy GC start (2018-05-14 20:25:29 +0000)

Eric Wong (8):
thread.c (timeout_prepare): common function
gc: rb_wait_for_single_fd performs GC if idle (Linux)
thread.c (do_select): perform GC if idle
thread.c: native_sleep callers may perform GC
thread_sync.c (mutex_lock): add missing else
benchmark: add benchmarks for sleepy GC
gc.c: allow disabling sleepy GC
gc.c: enter sleepy GC start

```
benchmark/bm_vm3_gc_io_select.rb        |  30 +++++
benchmark/bm_vm3_gc_io_wait.rb          |  21 ++++
benchmark/bm_vm3_gc_join_timeout.rb     |  11 ++
benchmark/bm_vm3_gc_remote_free_spmc.rb |  15 +++
benchmark/bm_vm3_gc_szqueue.rb          |  14 +++
gc.c                                    |  55 +++++++++
gc.h                                    |  28 +++++
thread.c                                | 197 ++++++++++++++++++++-----------
thread_pthread.c                        |   6 +
thread_sync.c                           |  21 +++-
thread_win32.c                          |   6 +
11 files changed, 337 insertions(+), 67 deletions(-)
create mode 100644 benchmark/bm_vm3_gc_io_select.rb
create mode 100644 benchmark/bm_vm3_gc_io_wait.rb
create mode 100644 benchmark/bm_vm3_gc_join_timeout.rb
create mode 100644 benchmark/bm_vm3_gc_remote_free_spmc.rb
create mode 100644 benchmark/bm_vm3_gc_szqueue.rb
```

#37 - 05/17/2018 09:02 PM - noahgibbs (Noah Gibbs)

I've now run a lot more batches of Rails Ruby Bench - 100 batches of 10,000 HTTP requests/batch. I am *definitely* seeing lower performance and more variance with Sleepy GC. Overall, Sleepy GC gets 169.4 req/sec mean throughput with variance of 6.4, while the previous commit gets 177.0 req/sec throughput with a variance of 3.8. So Sleepy GC v3 costs about 4% performance for Rails Ruby Bench running flat-out and completely parallel.

#38 - 05/18/2018 07:33 AM - normalperson (Eric Wong)

the.codefolio.guy@gmail.com wrote:

> Overall, Sleepy GC gets 169.4 req/sec mean throughput with
> variance of 6.4, while the previous commit gets 177.0 req/sec
> throughput with a variance of 3.8.

Thanks for testing! I think we will need to work on increasing
granularity of the steps. The variance actually bothers me a
bit, more.

I'll have to work on increasing granularity of the marking and
sweeping (which may hurt throughput in apps without IO-wait at
all...). And I won't be around much the next few days..

Also, our malloc accounting is silly expensive(*) and I think
we can do some lazy sweeping before making big allocations

(*) https://bugs.ruby-lang.org/issues/10238

#39 - 05/18/2018 09:13 AM - normalperson (Eric Wong)

> I'll have to work on increasing granularity of the marking and
> sweeping (which may hurt throughput in apps without IO-wait at
> all...). And I won't be around much the next few days..

Maybe the unlink_limit can be lowered if we are sweeping more
frequently:

https://80x24.org/spew/20180518085819.14892-9-e@80x24.org/raw

We may also add more sweeping around more malloc() calls.

I also wonder if you can help narrow down which feature causes
the most damage to performance:

RUBY_GC_SLEEPY_SWEEP || RUBY_GC_SLEEPY_MARK || RUBY_GC_SLEEPY_START

Perhaps try defining RUBY_GC_SLEEPY_MARK and
RUBY_GC_SLEEPY_START to 0 in gc.h and see if that helps
(Originally, I only intended to try sleepy sweep)

Anyways, rebased against current-ish trunk, since I had some
fixes and minor improvements which also conflicted with this:

The following changes since commit 74724107e96228c34f92a1f210342891bb29400e:

thread.c (rb_wait_for_single_fd): do not leak EINTR on timeout (2018-05-18 08:01:07 +0000)

are available in the Git repository at:

git://80x24.org/ruby.git sleepy-gc-v7

for you to fetch changes up to f6745fe9acd3453a38eb646006a5e2703732f973:

gc.c: lower sweep unlink limit and make tunable in gc.h (2018-05-18 08:51:45 +0000)

Eric Wong (8):
thread.c (timeout_prepare): common function
gc: rb_wait_for_single_fd performs GC if idle (Linux)
thread.c (do_select): perform GC if idle
thread.c: native_sleep callers may perform GC
benchmark: add benchmarks for sleepy GC
gc.c: allow disabling sleepy GC
gc.c: enter sleepy GC start
gc.c: lower sweep unlink limit and make tunable in gc.h

benchmark/bm_vm3_gc_io_select.rb | 30 +++++
benchmark/bm_vm3_gc_io_wait.rb | 21 ++++

```
benchmark/bm_vm3_gc_join_timeout.rb     |  11 ++
benchmark/bm_vm3_gc_remote_free_spmc.rb |  15 +++
benchmark/bm_vm3_gc_szqueue.rb          |  14 +++
gc.c                                    |  57 +++++++++-
gc.h                                    |  31 ++++++
thread.c                                | 191 ++++++++++++++++++++++----------
thread_pthread.c                        |   6 +
thread_sync.c                           |  21 +++-
thread_win32.c                          |   6 +
11 files changed, 337 insertions(+), 66 deletions(-)
create mode 100644 benchmark/bm_vm3_gc_io_select.rb
create mode 100644 benchmark/bm_vm3_gc_io_wait.rb
create mode 100644 benchmark/bm_vm3_gc_join_timeout.rb
create mode 100644 benchmark/bm_vm3_gc_remote_free_spmc.rb
create mode 100644 benchmark/bm_vm3_gc_szqueue.rb
```

Files

sleepy-gc-wip-v1.diff	5.37 KB	04/29/2018	normalperson (Eric Wong)