Ruby master - Feature #11098

Thread-level allocation counting

04/25/2015 04:21 PM - jasonrclark (Jason Clark)

Status:	Feedback
Priority:	Normal
Assignee:	
Target version:	

Description

This patch introduces a thread-local allocation count. Today you can get a global allocation count from GC.stat, but in multi-threaded contexts that can give a muddied picture of the allocation behavior of a particular piece of code.

Usage looks like this:

```
[2] pry(main)> Thread.new do
[2] pry(main)*   1000.times do
[2] pry(main)*     Object.new
[2] pry(main)*   end
[2] pry(main)*   puts Thread.current.allocated_objects
[2] pry(main)* end
1000
```

This would be of great interest to folks profiling Ruby code in cases where we can't turn on more detailed object tracing tools. We currently use GC activity as a proxy for object allocations, but this would let us be way more precise.

Obviously performance is a big concern. Looking at GET_THREAD, this doesn't appear to have any clearly large overhead. To check this out, I ran the following benchmark:

```
require 'benchmark/ips'

Benchmark.ips do |benchmark|
  benchmark.report "Object.new" do
    Object.new
  end

  benchmark.report "Object.new" do
    Object.new
  end

  benchmark.report "Object.new" do
    Object.new
  end
end
```

Results from a few run-throughs locally:

Commit 9955bb0 on trunk:

```
Calculating -------------------------------------
          Object.new   105.244k i/100ms
          Object.new   105.814k i/100ms
          Object.new   106.579k i/100ms
-------------------------------------------------
          Object.new      4.886M (± 4.5%) i/s -     24.417M
          Object.new      4.900M (± 1.9%) i/s -     24.549M
          Object.new      4.835M (± 7.4%) i/s -     23.980M
```

With this patch:

```
Calculating -------------------------------------
          Object.new   114.248k i/100ms
          Object.new   114.508k i/100ms
          Object.new   114.472k i/100ms
-------------------------------------------------
          Object.new      4.776M (± 5.1%) i/s -     23.878M
          Object.new      4.767M (± 5.2%) i/s -     23.818M
          Object.new      4.818M (± 1.5%) i/s -     24.154M
```

I don't have a good sense of whether this is an acceptable level of change or not, but I figured without writing the code to test there was no way to know. What do you think?

History

#1 - 05/06/2015 09:18 AM - normalperson (Eric Wong)

I don't mind this patch and even see it as an opportunity to drop objspace->total_allocated_objects entirely and rely exclusively on thread-local counters for GC.

I toyed around with a similar idea last year in [ruby-core:61424] for malloc accounting but haven't gotten much further. I might investigate this again over the summer.

Anyways some minor nits inline:

```
--- a/gc.c
+++ b/gc.c
@@ -1741,6 +1741,10 @@ newobj_of(VALUE klass, VALUE flags, VALUE v1, VALUE v2, VALUE v3)
 #endif

     objspace->total_allocated_objects++;
+
+    rb_thread_t *th = GET_THREAD();
+    th->allocated_objects++;
```

That would trip -Werror=declaration-after-statement in GCC. Declare `th' earlier or avoid the local variable entirely since you're only reading that once.

```
GET_THREAD()->allocated_objects++;
```

```
--- a/thread.c
+++ b/thread.c
@@ -2568,6 +2568,14 @@ rb_thread_group(VALUE thread)
     return group;
 }

+VALUE
+rb_thread_allocated_objects(VALUE thread)
+{
+    rb_thread_t *th;
+    GetThreadPtr(thread, th);
+    return LONG2NUM(th->allocated_objects);
+}

--- a/vm_core.h
+++ b/vm_core.h
@@ -598,6 +598,7 @@ typedef struct rb_thread_struct {
     int safe_level;
     int raised_flag;
     VALUE last_status; /* $? */
+    long allocated_objects;
```

Use uint64_t to avoid overflow on 32-bit systems as this counter never resets. This should never be a signed value.

#2 - 04/27/2016 06:07 PM - jasonrclark (Jason Clark)

- File thread-local-update.patch added

So apparently I didn't have notifications turned on and lost track of this. Sorry!

I've rebased this to current trunk and modified it per your suggestions Eric. How's this look? Anything else I can do to help this along?

#3 - 04/27/2016 07:21 PM - normalperson (Eric Wong)

jclark@newrelic.com wrote:

> So apparently I didn't have notifications turned on and lost track of this. Sorry!

No worries; I wish this place had a reply-to-all convention and
encourage emailing each other directly. It would mitigate this
Redmine server as a single-point-of-failure.

And yes, I miss stuff all the time and absolutely don't mind
being emailed directly if I don't respond after a week or two.
(not speaking for the rest of ruby-core).

> I've rebased this to current trunk and modified it per your
> suggestions Eric. How's this look? Anything else I can do to
> help this along?

LONG2NUM should probably be adjusted to ULL2NUM, since "long"
on 32-bit platforms is only 32-bits, not 64. There's no exact
U64T2NUM macro/function, but "long long" is 64-bits everywhere
nowadays, I think...

Other than that, technically it's fine. I don't get to make
the final decision as far as public API changes go, though.

#4 - 04/30/2016 07:46 PM - ko1 (Koichi Sasada)

Is it acceptable to use gem for this purpose?
You can make a gem using some hooks like allocation tracer gem (https://github.com/ko1/allocation_tracer).

Disadvantages:

- we need to require them first.
- it should be more slow.

#5 - 05/17/2016 06:31 AM - ko1 (Koichi Sasada)

- Status changed from Open to Feedback

#6 - 05/18/2016 01:33 AM - shyouhei (Shyouhei Urabe)

Koichi told me that this proposed functionality can be implemented on top of what is provided now.

He also said that per-thread allocation counting must be debug purpose; no use in a production environment is expected by him. He doesn't like to introduce overheads like retrieving thread contexts every time allocation happens, especially when such feature are likely rarely used.

#7 - 05/19/2016 04:56 PM - jasonrclark (Jason Clark)

allocation_tracer is awesome for debugging, and I've happily used it a number of times. Thank you for building it Koichi!

While most people certainly wouldn't use this, I do have a case for it in production. Specifically, I work at New Relic, and I wanted this for the Ruby agent (newrelic_rpm) to read. It would be a huge benefit to our users to pinpoint specific web requests that are allocation heavy. Production allocation often differs from other environments, so seeing what's actually happening on prod is a big benefit. The current global counters are noisy in the presence of other threads, and since we can't reliably provide the information for a specific request, we don't say anything at all.

Working as a gem has the disadvantages that you list, which are real concerns for us. In our experience few users enable optional features, so we probably won't even build something for an optional approach to adding this in.

If you still feel the overhead outweighs the use case we can close this out. It would give instrumenters like myself awesome insight into one of the most common causes of Ruby app slowdown, but I understand the concerns.

#8 - 05/27/2016 09:12 AM - ko1 (Koichi Sasada)

Thank you for your explanation.

After that we need to measure the gem version of this feature.
Can you write it? or should I write?

#9 - 05/27/2016 06:39 PM - jasonrclark (Jason Clark)

I'd be happy to spin up a gem version of this and see what the difference is. I'll report back once I have some code and findings there.

#10 - 10/09/2018 01:52 AM - daniel.ferreira@sage.com (Daniel Ferreira)

Files

thread-local.patch	2.04 KB	04/25/2015	jasonrclark (Jason Clark)
thread-local-update.patch	2.05 KB	04/27/2016	jasonrclark (Jason Clark)