[PATCH] set M_ARENA_MAX for glibc malloc

05/15/2018 08:25 AM - normalperson (Eric Wong)

Status: Open
Priority: Normal
Assignee:
Target version:

Description
Not everybody benefits from jemalloc and the extra download+install time is not always worth it. Let's make the user experience for glibc malloc users better, too.

Personally, I prefer using M_ARENA_MAX=1 (via MALLOC_ARENA_MAX env) myself, but there is currently a performance penalty for that.

gc.c (Init_GC): set M_ARENA_MAX=2 for glibc malloc

glibc malloc creates too many arenas and leads to fragmentation. Given the existence of the GVL, clamping to two arenas seems to be a reasonable trade-off for performance and memory usage.

Some users (including myself for several years, now) prefer only one arena, now, so continue to respect users' wishes when MALLOC_ARENA_MAX is set.

Thanks to Mike Perham for the reminder [ruby-core:86843]

This doesn't seem to conflict with jemalloc, so it should be safe for all glibc-using systems.

Related issues:
Related to Ruby master - Feature #15667: Introduce malloc_trim(0) in full gc ...

History
#1 - 05/16/2018 05:03 AM - bluz71 (Dennis B)

normalperson (Eric Wong) wrote:

Personally, I prefer using M_ARENA_MAX=1 (via MALLOC_ARENA_MAX env) myself, but there is currently a performance penalty for that.

gc.c (Init_GC): set M_ARENA_MAX=2 for glibc malloc

This is not desirable in the longer term.

CRuby will likely get true concurrency in the future via ko1's Guild proposal. Reducing arenas will create new contention and serialisation at the memory allocator level thus negating the full benefits of Guilds.

Debate is currently occurring in feature $14718 about using jemalloc to solve Ruby’s memory fragmentation issue on Linux. Resolution of that (one way or the other) should inform what to do here.

I believe this should be paused until #14718 is clarified.

#2 - 05/16/2018 05:32 AM - normalperson (Eric Wong)
dennisb55@hotmail.com wrote:

This is not desirable in the longer term.

I already had the following comment in the proposed patch:
I believe this should be paused until #14718 is clarified.

I think there's room for both. This patch is a no-op when jemalloc is linked.

#3 - 05/19/2018 03:04 AM - bluz71 (Dennis B)
As discussed in #14718 I am now a strong supporter of this change.

My prime concern, for the future, will be what to do when Guilds land?

What do you think Eric? What should happen when Guilds become available?

My thinking for arena count is maybe max (2 || 0.5 * core-count). But that is just speculation. I think quite a bit of testing would need to occur with a proper Guild'ed Ruby benchmark.

Also on the side (as you suggest) is continue to strive to improve glibc memory fragmentation behaviour (I assume by test cases and issue trackers and pull-requests)?

#4 - 05/19/2018 02:03 PM - mame (Yusuke Endoh)
I tried to change Mike's script to use I/O, and I've created a script that works best with glibc with no MALLOC_ARENA_MAX specified.

```ruby
# glibc (default)
$ time ./miniruby frag2.rb
VmRSS:  852648 kB
real   0m26.191s

# glibc with MALLOC_ARENA_MAX=2
$ time MALLOC_ARENA_MAX=2 ./miniruby frag2.rb
VmRSS:  1261032 kB
real   0m29.072s

# jemalloc 3.6.0 (shipped with Ubuntu)
$ time LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libjemalloc.so ./miniruby frag2.rb
VmRSS:  366624 kB
real   1m6.730s

$ ./miniruby -v
ruby 2.6.0dev (2018-05-19) [x86_64-linux]
```

As you see, default glibc is the fastest and most memory-efficient. Comparing to that, glibc with MALLOC_ARENA_MAX=2 uses 1.5x memory, and jemalloc 3.6.0 is twice slow.

I'm unsure what happens. I guess it is difficult to discuss this issue without glibc/jemalloc hackers.

```ruby
# frag2.rb
THREAD_COUNT = (ARGV[0] || "10").to_i

File.write("tmp.txt", "x" * 1024 * 64)

Threads = []

THREAD_COUNT.times do
  Threads << Thread.new do
    a = []
    100_000.times do
      a << open("tmp.txt") { |f| f.read } if a.size >= 1600
    end
  end
end

Threads.each { |th| th.join }

IO.foreach("/proc/#{$$}/status") do |line|
  print line if line =~ /VmRSS/ if RUBY_PLATFORM =~ /linux/
```
Yusuke, your script doesn't create any memory fragmentation, it throws away everything after 1600 and reads the exact same amount of data each time. I don't believe this is how Rails apps behave; they fragment over time. My script creates random sized data and holds onto 10% of the data to create "holes" in the heap and fragment memory quickly. I believe this better represents normal app conditions. I've edited your script slightly to randomly keep some data; it better matches the results I posted earlier. I think changing the IO to read random sizes would also exhibit worse memory:

```ruby
$ time MALLOC_ARENA_MAX=2 /ruby/2.5.1/bin/ruby frag2.rb
VmRSS: 1620356 kB
real 1m20.755s
user 0m38.057s
sys 1m2.881s

$ time /ruby/2.5.1/bin/ruby frag2.rb
VmRSS: 1857284 kB
real 1m19.642s
user 0m36.645s
sys 1m4.480s

$ more frag2.rb
THREAD_COUNT = (ARGV[0] || "10").to_i
File.write("/tmp/tmp.txt", "x" * 1024 * 64)
srand(1234)
Threads = []
Save = []

THREAD_COUNT.times do
  Threads << Thread.new do
    a = []
    100_000.times do
      a = open("/tmp/tmp.txt") { |f| f.read }
      Save << a if rand(100_000) < 1600
    end
  end
end

Threads.each{|th| th.join}
GC.start

IO.foreach("/proc/#{$$}/status") do |line|
  print line if line =~ /VmRSS/ if RUBY_PLATFORM =~ /linux/
end
```

Mike,

Yusuke script is still interesting for the datum that a Ruby script with MALLOC_ARENA_MAX=2 consumed more memory than a case using the default arena count (usually 32 on 4 core machines).

Here are my results of Yusuke's script on my 4-core machine (Intel i5-4590 quad-core, 16GB RAM, Linux Mint 18.3 with kernel 4.15.0).

% time ruby frag2.rb
VmRSS: 1,238,108 kB
real 0m38.792s

% time MALLOC_ARENA_MAX=2 ruby frag2.rb
VmRSS: 1,561,624 kB
real 0m39.002s

% time MALLOC_ARENA_MAX=4 ruby frag2.rb
VmRSS: 1,516,216 kB
real 0m36.614s

% time MALLOC_ARENA_MAX=32 ruby frag2.rb
VmRSS: 1,218,180 kB
real 0m36.857s

This is perplexing. Clearly Ruby should not be changing defaults until we understand results like this.
Here are jemalloc results (3.6.0 first and 5.0.1 second):

```bash
% time LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libjemalloc.so ruby frag2.rb
VmRSS:  933,328 kB
real    1m33.660s

% time LD_PRELOAD=/home/bluz71/.linuxbrew/Cellar/jemalloc/5.0.1/lib/libjemalloc.so ruby frag2.rb
VmRSS:  1,613,252 kB
real    0m27.530s
```
jemalloc 3.6.0 is extremely slow but with very low RSS.
jemalloc 5.0.1 is very fast (much faster than glibc) but also has the highest RSS.

Ruby can not be linked against a system supplied jemalloc since different jemalloc versions are extremely different; jemalloc 3.6 and 5.0 are basically different allocators that share the same name.

But we need to always remember that long lived Ruby processes on Linux have a very bad memory fragmentation as seen here:

jemalloc.jpg
What can the Ruby maintainers do?

I am less certain now what they should do than a week ago. But I still want to see the memory fragmentation issue reduced for default Ruby.

#7 - 05/21/2018 06:32 AM - normalperson (Eric Wong)

mame@ruby-lang.org wrote:

> I tried to change Mike's script to use I/O, and I've created a script that works best with glibc with no MALLOC_ARENA_MAX specified.

Interesting, you found a corner case of some fixed sizes where the glibc default appears the best.

I tested 16K instead of 64K since my computer is too slow and 16K is the default buffer size for IO.copy_stream, net/protocol, etc...) and the default was still best in that case.

So, I wonder if there is a trim threshold where this happens and multiple arenas tickles the threshold more frequently.

However, I believe Mike's script of random sizes is more representative of realistic memory use. Unfortunately, srand+rand alone is not enough to give consistently reproducible results for benchmarking with threads...

Maybe a single thread needs to generate all the random numbers and feed them round-robin to per-thread SizedQueue for deterministic results.

#8 - 08/01/2018 12:20 AM - sam.saffron (Sam Saffron)

Koichi / Nobu

Is there any chance we can merge this in and even backport to 2.4/2.5.

It is a very safe change for now and users can override the behavior if they need to using ENV (rare cases where you say host a ton of V8 runtimes)

It will give Ruby a much better reputation to have this out there and guilds are a few releases out.

#9 - 08/01/2018 09:42 AM - normalperson (Eric Wong)

https://bugs.ruby-lang.org/issues/14759#change-73263

Another followup on this, current glibc 2.28 still creates arenas without checking for contention, first. It can create (nproc * 8) arenas on 64-bit and (nproc * 2) on 32-bit.

I might be able to work on improving the situation in glibc, but the current glibc defaults makes no sense given our GVL for the
vast majority of Ruby programs.

Since glibc 2.26, tcache is also provided so that can mitigate most contention problems a Ruby program would encounter, lessening the need for multiple arenas.

#10 - 08/01/2018 10:04 AM - normalperson (Eric Wong)

https://bugs.ruby-lang.org/issues/14759#change-73263

Btw, has anybody tested this patch with various allocators to ensure it doesn’t trigger a conflict (resulting in a segfault) when using the other allocator via LD_PRELOAD? jemalloc seems fine, but I’m not sure about the others.

#11 - 08/01/2018 01:04 PM - Eregon (Benoit Daloze)
The question then becomes: What happens when Guilds land?

It will be a hard choice to have slow allocations when there are many guilds, or remove this behavior and regress on existing programs not using guilds. What does ko1 (Koichi Sasada) think about this? And any idea when guilds could land?

#12 - 08/01/2018 07:22 PM - normalperson (Eric Wong)

eregontp@gmail.com wrote:

The question then becomes: What happens when Guilds land?

We will probably match arenas to Guild count dynamically; depending on whether the program uses Guilds or not.

glibc checks arena_max whenever a new thread allocates memory for the first time, so arena_max doesn’t need to be frozen at Ruby startup.

I don’t know about ko1’s timeline, but glibc releases every 6 months (Aug and Feb); and Carlos’s willingness to accept URCU use in glibc is increasing my interest in contributing to glibc to fix malloc problems for all GNU users. :>

#13 - 08/01/2018 08:55 PM - Eregon (Benoit Daloze)

normalperson (Eric Wong) wrote:

We will probably match arenas to Guild count dynamically; depending on whether the program uses Guilds or not.

glibc checks arena_max whenever a new thread allocates memory for the first time, so arena_max doesn’t need to be frozen at Ruby startup.

Ah, great to know! I have no objection then.

I don’t know about ko1’s timeline, but glibc releases every 6 months (Aug and Feb); and Carlos’s willingness to accept URCU use in glibc is increasing my interest in contributing to glibc to fix malloc problems for all GNU users. :>

It sounds like there is lots of potential there :)

#14 - 08/08/2018 06:10 AM - ko1 (Koichi Sasada)
sam.saffron (Sam Saffron) wrote:

Koichi / Nobu

Is there any chance we can merge this in and even backport to 2.4/2.5.
It is a very safe change for now and users can override the behavior if they need to using ENV (rare cases where you say host a ton of V8 runtimes)

It will give Ruby a much better reputation to have this out there and guilds are a few releases out.

We are not maintainers for both versions, but I think we shouldn't backport it because we can change this parameter by an environment variable.

#15 - 08/08/2018 06:27 AM - ko1 (Koichi Sasada)

The question then becomes: What happens when Guilds land?

I have an optimistic opinion.
If transient heap successes, there are no big issue on it.
(too optimistic? :p)

#16 - 08/09/2018 11:35 AM - shyouhei (Shyouhei Urabe)

One question: is it possible to cancel the effect of M_ARENA_MAX? Given mame (Yusuke Endoh)'s corner case, it might be desirable for a user (or sysadmin) to be able to choose the behaviour between the proposed one and the status quo.

#17 - 08/09/2018 05:52 PM - normalperson (Eric Wong)

shyouhei@ruby-lang.org wrote:

One question: is it possible to cancel the effect of M_ARENA_MAX? Given mame (Yusuke Endoh)'s corner case, it might be desirable for a user (or sysadmin) to be able to choose the behaviour between the proposed one and the status quo.

Environment variable (MALLOC_ARENA_MAX) at startup (only), or they can use fiddle or C extension to call mallopt at anytime once a program is running.

#18 - 08/10/2018 12:51 AM - shyouhei (Shyouhei Urabe)

normalperson (Eric Wong) wrote:

shyouhei@ruby-lang.org wrote:

Yes the question is, what exactly is the value of MALLOC_ARENA_MAX that a user should specify to let malloc behave as it works in 2.5 now?

(Etc.nprocessors * 8) on 64-bit, (Etc.nprocessors * 2) on 32-bit.

#19 - 08/10/2018 01:42 AM - normalperson (Eric Wong)

shyouhei@ruby-lang.org wrote:

Yes the question is, what exactly is the value of MALLOC_ARENA_MAX that a user should specify to let malloc behave as it works in 2.5 now?
MALLOC_ARENA_MAX that a user should specify to let malloc behave as it works in 2.5 now?

(Etc.nprocessors * 8) on 64-bit, (Etc.nprocessors * 2) on 32-bit.

Hmm. Thank you. Now I am very faintly negative because
MALLOC_ARENA_MAX=2 ruby ...

is much easier than
MALLOC_ARENA_MAX=$((ls -1 /sys/bus/cpu/devices/|wc -l*8)) ruby...

#21 - 08/10/2018 02:22 AM - normalperson (Eric Wong)

shyouhei@ruby-lang.org wrote:

Hmm. Thank you. Now I am very faintly negative because
MALLOC_ARENA_MAX=2 ruby ...

is much easier than
MALLOC_ARENA_MAX=$((ls -1 /sys/bus/cpu/devices/|wc -l*8)) ruby...

nproc(1) command in GNU coreutils is handy :) 

MALLOC_ARENA_MAX=$(( $(nproc) * 8 )) ruby ...

Fwiw, a goal of glibc developers is to limit MALLOC_ARENA_MAX to nproc. It should happen when restartable sequences are available, but maybe it can happen for older kernels, too.

#22 - 08/10/2018 05:37 AM - bluz71 (Dennis B)

Ruby use generally falls into one of two categories: short-lived or very long-lived.

For short-lived Ruby scripts MALLOC_ARENA_MAX could, and maybe should, be left as is?

For long-lived Ruby processes MALLOC_ARENA_MAX absolutely should be reduced from the glibc default as noted in this request and #14718. Hopefully, in time glibc can be improved, but that may be many years away.

I wonder if a new runtime flag should be added to the Ruby executable, --long-lived? That flag could tweak internals to favour long runtimes. For the moment it could tweak MALLOC_ARENA_MAX, in the future it could tweak other aspects to favour low memory fragmentation.

Rails, Sinatra, Sidekiq users would then be encouraged to invoke Ruby via ruby --long-lived?

I suspect some/many folks would not like that since that will not be the default behaviour. But that would avoid the weird regression noted above that concerns shyouhei (Shyouhei Urabe). Ideally we don't want to penalize short-lived Ruby users for the benefit of long-lived Ruby users.

#23 - 03/15/2019 06:54 AM - bluz71 (Dennis B)

Hongli Lai's What causes Ruby memory bloat? post is most-interesting indeed.

That article has given rise to a new issue, opened by Sam Saffron, Introduce malloc_trim(0) in full gc cycles #15667.

If malloc_trim does not have any deleterious performance demons (to-be-determined) then there would be a very strong case to add a call to it in the Ruby GC hence rendering this issue completely moot.

We await real-world results of #15667

#24 - 03/15/2019 07:04 AM - duerst (Martin Dürst)

- Related to Feature #15667: Introduce malloc_trim(0) in full gc cycles added

Files

<table>
<thead>
<tr>
<th>Files</th>
<th>Size</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001-gc.c-Init_GC-set-M_ARENA_MAX-2-for-glibc-malloc.patch</td>
<td>1.46 KB</td>
<td>05/15/2018</td>
</tr>
</tbody>
</table>