Ruby master - Feature #14718

Use jemalloc by default?

04/27/2018 04:14 PM - mperham (Mike Perham)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

I know Sam opened #9113 4 years ago to suggest this but I'm revisiting the topic to see if there's any movement here for Ruby 2.6 or 2.7. I supply a major piece of Ruby infrastructure (Sidekiq) and I keep hearing over and over how Ruby is terrible with memory, a huge memory hog with their Rails apps. My users switch to jemalloc and a miracle occurs: their memory usage drops massively. Some data points:

https://twitter.com/brandonhilkert/status/987400365627801601
https://twitter.com/d_jones/status/989866391787335680
https://github.com/mperham/sidekiq/issues/3824#issuecomment-383072469

Redis moved to jemalloc many years ago and it solved all of their memory issues too. Their conclusion: the glibc allocator "sucks really really hard". http://oldblog.antirez.com/post/everything-about-redis-24.html

This is a real pain point for the entire Rails community and would improve Ruby's reputation immensely if we can solve this problem.

History

#1 - 04/27/2018 04:30 PM - jeremyevans0 (Jeremy Evans)

If Ruby decides to ship with jemalloc, let's make sure to be like Redis and only use it by default on Linux (as mentioned in the Redis 2.4 release announcement).

#2 - 04/27/2018 04:38 PM - mperham (Mike Perham)

It's also quite possible that jemalloc will give us a small performance increase. Another data point:

https://medium.com/@adrienjarthon/ruby-jemalloc-results-for-updown-io-d3fb2d32f67f

#3 - 04/27/2018 06:13 PM - normalperson (Eric Wong)

mperham@gmail.com wrote:

> https://bugs.ruby-lang.org/issues/14718

Agreed. I think it's acceptable to enable jemalloc by default short-term.

Long-term (unlikely this year) I hope to work with glibc team to improve malloc on their end. I have ideas which require LGPL code (wfcqueue) to implement, so not doable with jemalloc.

#4 - 04/27/2018 06:22 PM - ko1 (Koichi Sasada)

On 2018/04/28 3:12, Eric Wong wrote:

> Long-term (unlikely this year) I hope to work with glibc team to improve malloc on their end. I have ideas which require LGPL code (wfcqueue) to implement, so not doable with jemalloc.

super cool!!

--
// SASADA Koichi at atdot dot net

#5 - 04/27/2018 06:50 PM - shevegen (Robert A. Heiler)

I know way too little about the topic at hand, so I can not really comment on it, the usability, gains and so forth.

I had a look at the discussion at #9113, just due to my curiosity alone.

It seems as if matz has not yet commented on it; perhaps this issue has not been mentioned yet?

If this is the case, then perhaps the issue here may be a good candidate for the next ruby-developer meeting. Matz has the "3x as fast" goal, so this may perhaps add another +0.5% or so? :D

I guess what may be useful to know is if there is some summary or overview of trade offs for jemalloc by default, like pros and cons. I guess the pros have been mentioned already (less problems with memory) - are there cons that speak against the move? For example, by default, linux users using ruby, would they have to re-compile their ruby to benefit from this? Are there commandline switches? Questions like that. I am mostly asking so that information about it, also "trickles downwards" into more casual ruby hackers too.

Ruby is used by people who don't know that much about the internals and may not know the differences between glibc and jemalloc. I myself know what glibc is but I never really heard of jemalloc before. It's great if they improved on something though. Are there really no disadvantages?

(On a side note, glibc seems to be evolving quite slowly ... reminds me a bit of GCC versus LLVM but I may be wrong.)

#6 - 04/27/2018 07:28 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

The compile option to use jemalloc is already provided by Ruby from my understanding, this issue is about making it the default in Linux, where it seems to matter most.

#7 - 04/27/2018 07:59 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Regarding performance, it's subjective as always. What are your server's specifications? Usually RAM is not cheap, which means people don't usually have plenty of RAM in their servers, which means that lots of RAM usage could mean that applications are swapping a lot which means they get really slow. You see, there's always trade-offs regarding CPU cycles versus RAM when we talk about performance, because the surrounding environment matters a lot, so I find it really weird to set a goal for a generic programming language to become 3x faster as it doesn't mean anything to me... A benchmark could finish 3 times faster but that doesn't mean some application could become 3 times faster after switching to Ruby 3, for example. It could make no difference at all. However, if jemalloc allows RAM usage to be greatly reduced, then it makes all difference alone.

#8 - 04/27/2018 09:58 PM - mperham (Mike Perham)

I have a Sidekiq benchmark1 which processes 100,000 jobs through Redis, exercising networking, threading and JSON heavily. 2.5.0 on OSX takes 20.5 sec, 2.5.1 with jemalloc on OSX takes 18.3 sec, a 10% speed boost.

#9 - 04/28/2018 04:36 AM - noahgibbs (Noah Gibbs)

On Ubuntu Linux, with Rails Ruby Bench, jemalloc gives an overall speedup (not just memory, end-to-end speedup) of around 11%. Details: "http://engineering.appfolio.com/appfolio-engineering/2018/2/1/benchmarking-rubys-heap-malloc-tcmalloc-jemalloc"

Rails Ruby Bench is basically a big simulated concurrent workload for Discourse. So I think it's fair to say that jemalloc speeds up Discourse by around 11%.

Some of that may be jemalloc's memory savings allowing better caching. I don't have an easy way to measure direct malloc speedup vs better caching.

#10 - 04/28/2018 07:52 PM - nobu (Nobuyoshi Nakada)

Unfortunately, I have to object it.
As functions in libc, e.g., strdup, cannot be replaced by jemalloc and continue to use the system malloc, so freeing the pointer returned from such functions by je_free causes a crash (segfault, debug break, etc).

#11 - 04/28/2018 08:33 PM - mperham (Mike Perham)

Hi Nobu, GitHub, GitLab and Discourse are all running jemalloc in production. Since MRI offers --with-jemalloc, I assume it is safe and does not free any memory returned by strdup. Are you mainly concerned with native C extensions that call strdup?

#12 - 04/29/2018 08:21 AM - shyouhei (Shyouhei Urabe)

mperham (Mike Perham) wrote:

> Since MRI offers --with-jemalloc, I assume it is safe

That's not the reason for it being safe.

> Are you mainly concerned with native C extensions that call strdup?

You can have trouble without any C extensions. See also https://bugs.ruby-lang.org/issues/13524 (which is not glibc versus jemalloc, but the situation is the same).

#13 - 04/30/2018 03:57 PM - mperham (Mike Perham)

So is it impossible for Ruby to use jemalloc by default? jemalloc reduces Ruby memory usage 50-75% on large Rails apps and many large sites are using it in production so it must be stable for many environments. 40GB to 9GB:

jemalloc.jpg
For strdup, can we use jemalloc prefix mode so Ruby uses je_malloc and je_free for most memory but libc functions can still use malloc and free?

#14 - 05/01/2018 01:46 AM - shyouhei (Shyouhei Urabe)

mperham (Mike Perham) wrote:

> For strdup, can we use jemalloc prefix mode so Ruby uses je_malloc and je_free for most memory but libc functions can still use malloc and free?

I think this can be an option. Also, because no one is seriously using non-libc jemalloc on FreeBSD (see also: https://bugs.ruby-lang.org/issues/13175), such modification if any, should be restricted to the case of glibc only.

#15 - 05/02/2018 05:31 PM - mperham (Mike Perham)

Blog post this morning on the massive improvements from jemalloc for one heavy Ruby user:

https://brandonhilkert.com/blog/reducing-sidekiq-memory-usage-with-jemalloc/

An alternative is to tune glibc by reducing the number of arenas. Call this on startup:

```
#include "malloc.h"
mallopt(M_ARENA_MAX, 2)
```

https://www.gnu.org/software/libc/manual/html_node/Malloc-Tunable-Parameters.html#Malloc-Tunable-Parameters

Several people have reported massive memory reductions:

https://twitter.com/ideasasylum/status/990922248298156033
https://twitter.com/iotr/status/989537230594215936

#16 - 05/02/2018 09:03 PM - normalperson (Eric Wong)

mperham@gmail.com wrote:

> An alternative is to tune glibc by reducing the number of arenas. Call this on startup:
>
> ```
> #include "malloc.h"
> mallopt(M_ARENA_MAX, 2)
> ```

Probably acceptable. We need to verify it doesn't crash on systems where Ruby is built against various glibc versions but somebody tests with jemalloc via LD_PRELOAD.

We will also respect getenv("MALLOC_ARENA_MAX") if set; as I prefer MALLOC_ARENA_MAX=1 for my low-priority stuff.

Btw, do you have more information on which version(s) of glibc and compile options (or distro package) used?

I would like the Ruby community to work more closely with glibc developers in the future. Thanks.

#17 - 05/08/2018 02:27 AM - davidtgoldblatt (David Goldblatt)

Hi, I'm a jemalloc dev (sorry I didn't see this earlier, I try to keep an eye out for mentions on bug trackers). I don't have any insights about our performance vs. glibc's on Ruby workloads, but as far as correctness goes, I don't think the strdup concern is correct. The linker and loader should ensure that all references to allocator functions resolve to the jemalloc versions if any do. Replacing the allocator is also a documented feature of glibc (see https://www.gnu.org/software/libc/manual/html_node/Replacing-malloc.html#Replacing-malloc), so if there *is* some sort of linking screwiness happening, I suspect that their upstream will regard it as a true bug.

#18 - 05/08/2018 03:34 AM - mame (Yusuke Endoh)

I'm neutral. I have two questions.

- What will be changed by using jemalloc by default? I understood that jemalloc is great at the present time, but we are already able to use it with configure --with-jemalloc.
- How do we implement it? On an environment where jemalloc is unavailable, what will happen?

#19 - 05/10/2018 05:11 AM - bluz71 (Dennis B)

> - What will be changed by using jemalloc by default? I understood that jemalloc is great at the present time, but we are already able to use it with configure --with-jemalloc.

jemalloc clearly has superior memory fragmentation characteristics compared with glibc malloc on Linux, this characteristic greatly benefits long-lived processes which is the a type of workload Ruby is often used for.

Not speaking for Mike, but I assume he is asking why not make it the default now/soon? Some in the Ruby community have been using jemalloc for years with great benefit and no issues. They'd like to see it widely deployed for all Ruby users to gain this benefit not just the select few who know how to enable the obscure option.

> - How do we implement it? On an environment where jemalloc is unavailable, what will happen?

I would suggest shipping a minimal jemalloc sourc with Ruby. It is quite likely that the jemalloc crew (Jeremey and David) would help in regards to what source and version to use and ship.

Note, Golang uses and ships with their own tcmalloc derived allocator (see: https://golang.org/src/runtime/malloc.go).

Likewise, the Rust Language uses jemalloc (see: https://twitter.com/rustlang/status/740299354888536064).

Some programming languages do ship with the own allocator source. If Ruby did the same with jemalloc it would not be out of the ordinary. Personally I would want to enable it by default for all Unix-like system (not just Linux).

#20 - 05/10/2018 05:37 AM - bluz71 (Dennis B)

normalperson (Eric Wong) wrote:

> mperham@gmail.com wrote:
>
> > https://bugs.ruby-lang.org/issues/14718
>
> Agreed. I think it's acceptable to enable jemalloc by default short-term.

I actually believe if enabled it would for the long-term.

> Long-term (unlikely this year) I hope to work with glibc team to
> improve malloc on their end. I have ideas which require LGPL
> code (wfcqueue) to implement, so not doable with jemalloc.

This is admirable, but we are still talking about years away. 2021 before in a Linux LTS release?

The fundamentals of the glibc allocator derive from the ptmalloc2 allocator of the 1990s, which in turn builds upon the work of Doug Lea's allocators from the late 80s and early 90s. The fundamental core of glibc's allocator is decades old. It is not surprising that even today it still has real problems with memory fragmentation, it is fundamentally old.

I doubt whether the glibc allocator could compete with jemalloc in regards to memory fragmentation in combination with multiple-Guilds.

P.S. I do use jemalloc with Ruby and the results mirror what Mike has seen, it is a Ruby game-changer; probably more-so than even a JIT.

#21 - 05/10/2018 05:54 AM - jeremyevans0 (Jeremy Evans)

bluz71 (Dennis B) wrote:

> Some programming languages do ship with the own allocator source. If Ruby did the same with jemalloc it would not be out of the ordinary. Personally I would want to enable it by default for all Unix-like system (not just Linux).

It should definitely not be the default for all Unix-like systems. On OpenBSD using jemalloc by default would be considered an exploit mitigation countermeasure. jemalloc may have better performance, but different operating systems have different priorities, and certainly on OpenBSD we would want to use the system malloc to benefit from security features like guard pages, canaries, free checking, and free unmapping.

#22 - 05/10/2018 06:26 AM - bluz71 (Dennis B)

jeremyevans0 (Jeremy Evans) wrote:

> It should definitely not be the default for all Unix-like systems. On OpenBSD using jemalloc by default would be considered an exploit mitigation countermeasure. jemalloc may have better performance, but different operating systems have different priorities, and certainly on OpenBSD we would want to use the system malloc to benefit from security features like guard pages, canaries, free checking, and free unmapping.

Fair point Jeremy. At the end of the day we are primarily talking about a specific Linux glibc malloc characteristic we are wanting to avoid. I am fine with the limits you describe.

Question, do you know what Golang and Rust do on those same platforms? They both appear to ship their own allocators; do they pick-and-choose as you describe? At the end of the day, probably does not matter for us Rubyists (I am just curious).

Lastly, thanks for jemalloc. In a past life (pre-Ruby) I also used it with great success with a custom NoSQL database server.

#23 - 05/10/2018 05:20 PM - jeremyevans0 (Jeremy Evans)

bluz71 (Dennis B) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > It should definitely not be the default for all Unix-like systems. On OpenBSD using jemalloc by default would be considered an exploit mitigation countermeasure. jemalloc may have better performance, but different operating systems have different priorities, and certainly on OpenBSD we would want to use the system malloc to benefit from security features like guard pages, canaries, free checking, and free unmapping.
>
> Fair point Jeremy. At the end of the day we are primarily talking about a specific Linux glibc malloc characteristic we are wanting to avoid. I am fine with the limits you describe.
>
> Question, do you know what Golang and Rust do on those same platforms? They both appear to ship their own allocators; do they pick-and-choose as you describe? At the end of the day, probably does not matter for us Rubyists (I am just curious).

On OpenBSD, I don't think we modify either. I think Golang uses a custom allocator based on tcmalloc, and Rust is still apparently using the jemalloc allocator.

> Lastly, thanks for jemalloc. In a past life (pre-Ruby) I also used it with great success with a custom NoSQL database server.

I think you have me confused with Jason Evans, the author of jemalloc. :)

#24 - 05/11/2018 01:31 AM - shyouhei (Shyouhei Urabe)

So, I think it's clear people are interested in replacing glibc malloc, not everything that exist in the whole universe.

Let's discuss how we achieve that. There can be several ways:

- Just enable --with-jemalloc default on, only for Linux.
  - pro: This is the easiest to implement.
  - pro: Arguably works well. Already field proven.
  - con: Mandates runtime dependency for libjemalloc on those systems.
- Detect glibc on startup and try linking jemalloc then.
  - pro: Even works on systems without jemalloc.
  - con: Sacrifices process bootup time, which is a bad thing.
  - con: Tricky to implement, prone to bug.
- Bundle jemalloc and link it statically.
  - pro: No runtime hell.
  - con: Bloats source distribution. Costs non-glibc users.
  - con: Also costs the core devs because they have to sync the bundled jemalloc with the upstream.

Any opinions? Or any other ways?

#25 - 05/11/2018 02:22 AM - normalperson (Eric Wong)

shyouhei@ruby-lang.org wrote:

> So, I think it's clear people are interested in replacing glibc malloc, not everything that exist in the whole universe.
>
> Let's discuss how we achieve that. There can be several ways:
>
> - Just enable --with-jemalloc default on, only for Linux.
>   - pro: This is the easiest to implement.
>   - pro: Arguably works well. Already field proven.

    - con: Mandates runtime dependency for libjemalloc on those systems.

How about only link against it by default if detected. It will
be like our optional dependency on GMP. We will depend on
distros to enable/disable the dependency on it.

I'm not 100% sure jemalloc is the best solution today, nor will
be in the future. With some of my projects, glibc malloc is
often slightly smaller and faster. But I realize typical Ruby
code is not written with low memory usage in mind :<

jemalloc itself is a LOT of code for a malloc implementation,
so the icache footprint is non-trivial and that impacts
startup time (but that is currently overshadowed by Rubygems).

- Detect glibc on startup and try linking jemalloc then.
    - pro: Even works on systems without jemalloc.
    - con: Sacrifices process bootup time, which is a bad thing.
    - con: Tricky to implement, prone to bug.

I'm not sure how that would work since malloc seems to get
called before main() by the linker.

Perhaps we can try mallopt(M_ARENA_MAX, 2) for glibc by
default; along with auto-enabling jemalloc by iff installed.

- Bundle jemalloc and link it statically.

NAK. The cons your list are huge and makes life difficult for
distros and code auditing.

```
- pro: No runtime hell.
- con: Bloats source distribution.  Costs non-glibc users.
- con: Also costs the core devs because they have to sync the bundled jemalloc with the upstream.
```

#26 - 05/11/2018 03:01 AM - mame (Yusuke Endoh)

normalperson (Eric Wong) wrote:

> We will depend on distros to enable/disable the dependency on it.

Yes. This is what I wanted to ask.
Since almost all people use distros' package of Ruby, the final decision is left to distros.
This is because I'm unsure what is changed by this feature.
Why don't you ask distro maintainers to use --with-jemalloc for their packages?

(Don't get me wrong: I'm never against jemalloc nor this feature. I'm purely unsure.)

#27 - 05/11/2018 03:28 AM - bluz71 (Dennis B)

- Just enable --with-jemalloc default on, only for Linux.

This is my close 2nd favoured option.

- Detect glibc on startup and try linking jemalloc then.

No, too complicated. I don't think anyone wants this.

- Bundle jemalloc and link it statically.

My favoured, but I understand if this is not desired. Similar to shipping and linking in a big JIT library is not desired.

Just enabling --with-jemalloc would satisfy pretty much most (myself included). On Linux it would be one extra dependency at build time (for distros, rvm/rbenv/ruby-install). Dependency management on the main Linux distros is quite easy these days; heck Redis already has such a dependency.

#28 - 05/11/2018 03:38 AM - bluz71 (Dennis B)

mame (Yusuke Endoh) wrote:

> (Don't get me wrong: I'm never against jemalloc nor this feature. I'm purely unsure.)

It is right to be unsure and sceptical.

But in this case we do have years worth of experience with the benefits being clear and tangible for multiple users, not just Mike Perham (or Sam Saffron).

If this were to happen a --without-jemalloc flag should be provided.

#29 - 05/11/2018 04:00 AM - bluz71 (Dennis B)

jeremyevans0 (Jeremy Evans) wrote:

> I think you have me confused with Jason Evans, the author of jemalloc. :)

Yes indeed I did, please accept my apologies, I did believe you were the JE in jemalloc.

#30 - 05/11/2018 04:06 AM - bluz71 (Dennis B)

bluz71 (Dennis B) wrote:

> Just enabling --with-jemalloc would satisfy pretty much most (myself included). On Linux it would be one extra dependency at build time (for distros, rvm/rbenv/ruby-install). Dependency management on the main Linux distros is quite easy these days; heck Redis already has such a dependency.

No, actually Redis ships with their own jemalloc, it is not a build dependency:

http://oldblog.antirez.com/post/everything-about-redis-24.html

"Including jemalloc inside of Redis (no need to have it installed in your computer, just download the Redis tarball as usually and type make) was a huge win. Every single case of fragmentation in real world systems was fixed by this change, and also the amount of memory used dropped a bit."

I am not saying that Ruby should do as Redis did, I was just correcting my incorrect assumption that Redis linked against the system jemalloc library (it does not).

#31 - 05/11/2018 04:23 PM - mperham (Mike Perham)

```
Bundle jemalloc and link it statically.

    pro: No runtime hell.
    con: Bloats source distribution. Costs non-glibc users.
    con: Also costs the core devs because they have to sync the bundled jemalloc with the upstream.
```

This would be my choice as a maintainer and I trust antirez knows what he's doing when he made the same choice for Redis. You control the exact version and any resulting bugs. If you depend on the distro's package, you will integrate with a variety of versions, all with their own set of platform-specific bugs and crash reports. Ubuntu 14 might have jemalloc 3.6.0, 16 might have 4.2.0, 18 might have 5.0.1, etc. Dealing with those platform issues will also cost the core devs.

Linux/glibc is the platform for 90% of Ruby users in production. It's ok to support FreeBSD and OpenBSD but they are very small in terms of actual usage. If you want to make things better for the vast majority of the community, focus on Linux.

#32 - 05/12/2018 12:58 AM - bluz71 (Dennis B)

mperham (Mike Perham) wrote:

> ```
> Bundle jemalloc and link it statically.
>
> This would be my choice as a maintainer and I trust antirez knows what he's doing when he made the same ch
> oice for Redis.  You control the exact version and any resulting bugs.  If you depend on the distro's pack
> age, you will integrate with a variety of versions, all with their own set of platform-specific bugs and c
> rash reports.  Ubuntu 14 might have jemalloc 3.6.0, 16 might have 4.2.0, 18 might have 5.0.1, etc.  Dealin
> g with those platform issues will also cost the core devs.
> ```

This is a very strong point.

Legacy versions of jemalloc are solid as a rock. Newer versions have a higher chance of unexpected quirks. For instance Sam Saffron had unusual results with jemalloc 4.3.0:

https://github.com/SamSaffron/allocator_bench

Redis ships jemalloc 4.0.3 (or near to) as seen here:

https://github.com/antirez/redis/tree/unstable/deps/jemalloc

If Ruby was to ship jemalloc (also my preference), why not just take the same version as Redis? It has been battle-tested for seven years. We could also chat with Sam Saffron and the jemalloc maintainers for their thoughts about a preferred jemalloc version for Ruby. Ruby could ship and use that jemalloc and lock it in at a known and stable version.

> Linux/glibc is the platform for 90% of Ruby users in production. It's ok to support FreeBSD and OpenBSD but they are very small in terms of actual usage. If you want to make things better for the vast majority of the community, focus on Linux.

The FreeBSD system allocator already is jemalloc based. I am lead to believe that was thee first usage of Jason Evans memory allocator.

Replacing the OpenBSD memory allocator has been explained as an anti-pattern by Jeremy a few posts up. I would not replace it.

I tend to agree now with limiting any jemalloc changes just to the Linux platform where the memory fragmentation problem exists.

I'd still keep the --with-jemalloc option; this would always build in an externally provided jemalloc library (even on Linux, e.g a Linux or FreeBSD user wishing to use jemalloc 5.1.0). A new --without-jemalloc option would be nice to provide which would be a no-op on most platforms and on Linux it would force usage of glibc malloc (when desired).

Lastly, Ruby would not be on the bleeding edge if it adopted jemalloc (on Linux), this allocator is already used by default in the following:

- Redis database
- MarkLogic database
- Firefox browser
- Rust language
- Facebook

P.S. I have no links with the jemalloc project, but I do use it now with Ruby and I also did successfully use it for many years in my previous database server life.

#33 - 05/14/2018 12:09 AM - shyouhei (Shyouhei Urabe)

mperham (Mike Perham) wrote:

> You control the exact version and any resulting bugs.

This literally means we have to bug fix jemalloc. Please no.

> Linux/glibc is the platform for 90% of Ruby users in production. It's ok to support FreeBSD and OpenBSD but they are very small in terms of actual usage. If you want to make things better for the vast majority of the community, focus on Linux.

Be warned; you are entering a field of persecuting minorities. Please respect others. This is not a matter of majority versus minority. If you insist it is, I have to stand for diversity.

Everyone are equally important in our community. Thank you.

#34 - 05/14/2018 02:14 AM - bluz71 (Dennis B)

shyouhei (Shyouhei Urabe) wrote:

> This literally means we have to bug fix jemalloc. Please no.

It's a judgement call of which is more likely, theoretical jemalloc bugs or Ruby-on-Linux memory fragmentation?

Carrying an extra dependency is a burden for the Ruby maintainers, yes.
But likewise, memory fragmentation on the Linux platform for long-lived Ruby processes is a real burden some users carry.

The Firefox browser (for a decade), Redis (for seven years) and the Rust programming language all considered Linux memory fragmentation to be a bug which they fixed by switching to jemalloc.

Legacy jemalloc versions (say 3.6 to around 4.1) are battle-tested and rock-solid with Ruby (and many other technologies).

> Be warned; you are entering a field of persecuting minorities. Please respect others. This is not a matter of majority versus minority. If you insist it is, I have to stand for diversity.
>
> Everyone are equally important in our community. Thank you.

You are taking offence where no offence was intended. All Ruby users matters including those that experience memory fragmentation on the Linux platform.

This graph explains the situation with clarity:

jemalloc.jpg

#35 - 05/14/2018 03:16 AM - shyouhei (Shyouhei Urabe)

You can explicitly specify --with-jemalloc, for a long time. I don't see any practical reason why that's insufficient. I can turn it default on. Anything beyond that requires a good reason to do so, not just "everybody else is jumping off the ridge" thing.

Please. I want to make the situation better. Why do we have to bundle 3rd-party source codes?

#36 - 05/14/2018 06:42 AM - bluz71 (Dennis B)

shyouhei (Shyouhei Urabe) wrote:

> You can explicitly specify --with-jemalloc, for a long time. I don't see any practical reason why that's insufficient. I can turn it default on. Anything beyond that requires a good reason to do so, not just "everybody else is jumping off the ridge" thing.

Turning on --with-jemalloc by default for the Linux platform achieves the desired result we want.

I will be more than pleased if this is the final result. Can you do this?

Note, this will be a new build time dependency on Linux which should not be an issue since all major Linux distributions will provide the jemalloc library and development packages in their base repositories (libjemalloc-dev for Debian-based systems).

> Please. I want to make the situation better. Why do we have to bundle 3rd-party source codes?

Correct, no bundling is strictly required, it just offered a known-quantity since different jemalloc versions will be shipped with the various Linux distributions.

I still believe a --without-jemalloc option should be provided as an escape hatch.

Also, if this change happens then we should contact the various Ruby build/version systems (rvm, rbenv and chruby/ruby-install) about this since they should change their auto-install-dependency engines to also include jemalloc (on Linux). I can do all that if the default changes.

If this change happens for Ruby 2.6 I genuinely believe that very many long-lived Ruby-based applications will benefit from this change.

#37 - 05/14/2018 07:33 AM - sam.saffron (Sam Saffron)

> You can explicitly specify --with-jemalloc, for a long time. I don't see any practical reason why that's insufficient.

I think this is a very legitimate question and worth talking through.

Vast majority of consumers of Ruby use a tool of sorts to get Ruby installed. This could be docker, rbenv, rvm or distro packaged rubies (and dozens of other options).

If Ruby does not make a "policy" in the the source code, convincing the 50 different places that deal with building Ruby to move to jemalloc/tcmalloc is something very tricky. A mandate from above that bloats our MRI source a tiny bit for the good of Linux will force a various packagers in the 2.6 timeframe to follow the best practice.

mperham (Mike Perham) I strongly recommend that regardless of what happens here you open tickets on docker-ruby/rbenv/rvm/ubuntu/redhat and debian to move to jemalloc or tcmalloc by default. That may push the scale a bit.

shyouhei (Shyouhei Urabe) if I were BDFL I would go with statically including 3.6.0 jemalloc or tcmalloc for Linux x64, but given the stability of tcmalloc over the past few years I would probably lean towards doing what golang do and picking tcmalloc cause being stuck with old jemalloc is not ideal and it will be harder to get support from the jemalloc team if we are not on 5.0.1.

That said I get why the core team is so nervous here, there is another option here that does not bloat source:

- On Linux X64 default to downloading a validated (using SHA1) tcmalloc/jemalloc from a particular location, allow for a flag to bypass this behavior

I know why everyone is so nervous here, but the bad behavior of glibc malloc on Linux is hurting Ruby's reputation a lot.

#38 - 05/14/2018 07:53 AM - shyouhei (Shyouhei Urabe)

Review wanted:

```
Index: trunk/configure.ac
===================================================================
--- trunk/configure.ac  (revision 63416)
+++ trunk/configure.ac  (working copy)
@@ -1014,7 +1014,7 @@

 AC_ARG_WITH([jemalloc],
   [AS_HELP_STRING([--with-jemalloc],[use jemalloc allocator])],
-  [with_jemalloc=$withval], [with_jemalloc=no])
+  [with_jemalloc=$withval], [with_jemalloc=yes])
 AS_IF([test "x$with_jemalloc" = xyes],[
   AC_SEARCH_LIBS([malloc_conf], [jemalloc],
     [AC_DEFINE(HAVE_LIBJEMALLOC, 1)], [with_jemalloc=no])
```

#39 - 05/14/2018 08:30 AM - bluz71 (Dennis B)

sam.saffron (Sam Saffron) wrote:

- On Linux X64 default to downloading a validated (using SHA1) tcmalloc/jemalloc from a particular location, allow for a flag to bypass this behavior

I have experimented with both tcmalloc and jemalloc and I am very strongly of the opinion that jemalloc is the appropriate allocator, out of the two, for Ruby (on Linux). I found tcmalloc to be a little bit more performant than jemalloc but at the expense of clearly higher memory utilisation. The main Ruby need is long-lived memory stability and low utilisation (jemalloc excels at both). I've also found jemalloc (like tcmalloc) to be more performant than glibc (a bonus win).

> I know why everyone is so nervous here, but the bad behavior of glibc malloc on Linux is hurting Ruby's reputation a lot.

This is a genuine pain point no doubt.

It is good to see shyouhei's proposed change just above. Even just linking against a system supplied jemalloc library, by default, will be a great step forward.

#40 - 05/14/2018 03:58 PM - mperham (Mike Perham)

Shyouhei, I'm very happy with only the change in default. I believe that will significantly improve the runtime of 1000s of Ruby apps. Thank you!

#41 - 05/14/2018 04:31 PM - wyhaines (Kirk Haines)

mperham (Mike Perham) wrote:

> Shyouhei, I'm very happy with only the change in default. I believe that will significantly improve the runtime of 1000s of Ruby apps. Thank you!

It absolutely will. In my experience, jemalloc is a tremendous boon to long running processes. I am very happy to see this happening, finally.

#42 - 05/15/2018 03:49 AM - mame (Yusuke Endoh)

I've already said to Shyouhei, his patch has a problem: configure script fails on an environment where jemalloc is unavailable. I have hoped that those who want to make jemalloc default would found (and fix) the issue.

With the intention of including some self-discipline: If you want to change anything, we should not just repeat "I want! It should be! Trust me!". Instead, we should pay for it. Please act seriously. In this case, please consider and clarify the goal, propose and compare multiple concrete plans for the goal, and write/review/test a patch.

#43 - 05/15/2018 07:17 AM - sam.saffron (Sam Saffron)

mame (Yusuke Endoh) I agree this is a problem it makes it slightly more complex to install Ruby. Ideally the build process could default to "trying to download" a specific version of jemalloc and building against it if it is unacceptable to include jemalloc in the source. I think it is important for Ruby to set proper defaults... and the default of "whatever jemalloc" is not a good default.

I think short term just bundling 3.6.0 is the safest thing to do and then running with_jemalloc **only** on x64 Linux.

#44 - 05/15/2018 08:43 AM - normalperson (Eric Wong)

> mperham@gmail.com wrote:
>
> > An alternative is to tune glibc by reducing the number of arenas. Call this on startup:

```
#include "malloc.h"
mallopt(M_ARENA_MAX, 2)
```

Btw, I created https://bugs.ruby-lang.org/issues/14759 for this.

Anyways, I hope Ruby can be used to foster competition amongst
malloc developers and ALL malloc implementations can improve.

#45 - 05/15/2018 11:16 PM - mperham (Mike Perham)

Hi Yusuke, I'm sorry if this hasn't been approached the right way.

The Problem: Many large Rails apps have a memory fragmentation problem on 64-bit Linux. I hear reports of excessive memory consumption from users every day, I linked some of those tweets above.

The Immediate Solution: Tune arenas, see #14759. This seems to greatly reduce memory fragmentation. This is a glibc developer's recommendation:

> Therefore the solution to a program with lots of threads is to limit the arenas as a trade-off for memory.

I'm working on a script which reproduces the memory fragmentation but it requires a dedicated large 64-bit Linux machine instance, with many gigabytes of RAM.

Jemalloc might be a good long-term solution; I will try to get you a script so we can show you the memory improvement, not just say it. :-)

#46 - 05/16/2018 05:11 AM - bluz71 (Dennis B)

mperham (Mike Perham),

Noah Gibbs has a Discourse based benchmark that could be useful here in regards to producing a test-case to highlight memory fragmentation and utilisation.

He has already benchmarked the raw speed of tcmalloc & jemalloc vs glibc in this post. Side note, jemalloc provides about a 10% speedup with these Discourse-based tests over glibc.

Now we need to run that same test but with memory utilisation metrics over an extended time-frame; many hours (not minutes).

Should we bring in Noah into this?

#47 - 05/16/2018 05:14 AM - bluz71 (Dennis B)

mperham (Mike Perham) wrote:

> An alternative is to tune glibc by reducing the number of arenas. Call this on startup:
>
> ```
> #include "malloc.h"
> mallopt(M_ARENA_MAX, 2)
> ```

I am not favourable to this.

Less arenas means greater contention & serialisation at the memory allocator level. Guilds will be a pathway to true concurrency for CRuby, the above setting will negate those benefits. This will create an undesirable bottleneck once Guilds become a reality.

#48 - 05/16/2018 05:29 AM - mperham (Mike Perham)

Dennis, my focus for this issue is fixing the memory bloat problem that plagues 1000s of Rails apps today, not a hypothetical performance problem that might affect Guilds months or years from now.

In the future Ruby could increase arenas if additional Guilds beyond the first are allocated but I believe that M_ARENA_MAX=2 would be a good, useful solution today for non-Guild apps that upgrade to Ruby 2.6.

#49 - 05/16/2018 05:54 AM - bluz71 (Dennis B)

mperham (Mike Perham) wrote:

> Dennis, my focus for this issue is fixing the memory bloat problem that plagues 1000s of Rails apps today, not a hypothetical performance problem that might affect Guilds months or years from now.
>
> In the future Ruby could increase arenas if additional Guilds beyond the first are allocated but I believe that M_ARENA_MAX=2 would be a good, useful solution today for non-Guild apps that upgrade to Ruby 2.6.

Sure, it is a trade-off.

All options have pros and cons; including that one. I was just making the point that M_ARENA_MAX=2 is not a simple free and easy win, it too has negative consequences.

Personally I don't like the prospect of a hard-coded arena count deep in the Ruby source code. It reeks of a code-smell. How was 2 decided upon, where are results to back it up? Why note 1 or 3 or 4? What happens when Ruby is run on a 20-core machine when Guilds arrive? etc.

#50 - 05/16/2018 09:13 AM - normalperson (Eric Wong)

dennisb55@hotmail.com wrote:

> Personally I don't like the prospect of a hard-coded arena
> count deep in the Ruby source code. It reeks of a code-smell.

I don't like it, either; but it's good enough today for enough users. It is far better than massive fragmentation and RSS usage, and doesn't cost users extra download+install time.

> How was 2 decided upon, where are results to back it up? Why
> note 1 or 3 or 4? What happens when Ruby is run on a 20-core
> machine when Guilds arrive? etc.

As you know, GVL is currently the major bottleneck, so malloc contention is rare for current Ruby programs and thus one arena is often sufficient. That said, having one extra arena (2 total) seemed to help throughput with multithreaded apps when I experimented years ago. More arenas did not seem to help. Mike's current messages and citations seem to reinforce what I remember from years ago.

Of course, Ruby is currently slow for many other reasons, too; and part of that can be helped by reducing calls to malloc entirely, and perhaps reducing native thread contention by reintroducing lightweight green threads.

#51 - 05/16/2018 06:45 PM - mperham (Mike Perham)

I've created a script which attempts to reproduce memory fragmentation.

```
=begin
This script attempts to reproduce poor glibc allocator behavior within Ruby, leading
to extreme memory fragmentation and process RSS bloat.

glibc allocates memory using per-thread "arenas".  These blocks can easily fragment when
some objects are free'd and others are long-lived.

Our script runs multiple threads, all allocating randomly sized "large" Strings between 4,000
and 40,000 bytes in size.  This simulates Rails views with ERB creating large chunks of HTML
to output to the browser. Some of these strings are kept around and some are discarded.

With the builds below and the frag.rb script, jemalloc and MALLOC_ARENA_MAX=2 both show a noticeable reduction
 in RSS.
=end
```

Results, it shows a significant reduction in RSS when run with jemalloc or MALLOC_ARENA_MAX=2.

```
> MALLOC_ARENA_MAX=32 /root/versions/2.5.1/bin/ruby -v frag.rb

ruby 2.5.1p57 (2018-03-29 revision 63029) [x86_64-linux]
 '--disable-install-doc' '--prefix=/root/versions/2.5.1'
Total string size: 1903MB
VmRSS:   2831832 kB

> MALLOC_ARENA_MAX=2 /root/versions/2.5.1/bin/ruby -v frag.rb

ruby 2.5.1p57 (2018-03-29 revision 63029) [x86_64-linux]
 '--disable-install-doc' '--prefix=/root/versions/2.5.1'
Total string size: 1917MB
VmRSS:   2311052 kB

> /root/versions/2.5.1j/bin/ruby -v frag.rb
```

```
ruby 2.5.1p57 (2018-03-29 revision 63029) [x86_64-linux]
 '--with-jemalloc' '--disable-install-doc' '--prefix=/root/versions/2.5.1j'
Total string size: 1908MB
VmRSS:   2306372 kB
```

https://gist.github.com/mperham/ac1585ba0b43863dfdb0bf3d54b4098e

#52 - 05/16/2018 07:22 PM - jeremyevans0 (Jeremy Evans)

mperham (Mike Perham) wrote:

> Results, it shows a significant reduction in RSS when run with jemalloc or MALLOC_ARENA_MAX=2.
>
> ```
> > MALLOC_ARENA_MAX=2 /root/versions/2.5.1/bin/ruby -v frag.rb
>
> ruby 2.5.1p57 (2018-03-29 revision 63029) [x86_64-linux]
>  '--disable-install-doc' '--prefix=/root/versions/2.5.1'
> Total string size: 1917MB
> VmRSS:     2311052 kB
>
> > /root/versions/2.5.1j/bin/ruby -v frag.rb
>
> ruby 2.5.1p57 (2018-03-29 revision 63029) [x86_64-linux]
>  '--with-jemalloc' '--disable-install-doc' '--prefix=/root/versions/2.5.1j'
> Total string size: 1908MB
> VmRSS:     2306372 kB
> ```

I'm not sure what the performance differences are, but looking purely at the memory usage, a 0.2% difference in memory between jemalloc and glibc with MALLOC_ARENA_MAX=2 seems insubstantial. Certainly if the memory difference is that small, absent significant performance advantages, arguments for bundling jemalloc with ruby or downloading it at configure/make time don't make much sense, compared to just setting M_ARENA_MAX=2 if glibc is being used.

#53 - 05/16/2018 08:12 PM - mperham (Mike Perham)

Yeah, it seems that way. Those results are with jemalloc 3.6.0 (which is what Ubuntu ships), a version 4 years old. jemalloc 5.1.0 is newest and might have space and time improvements. The arena tweak appears to solve much of the immediate fragmentation issue.

#54 - 05/16/2018 11:54 PM - mame (Yusuke Endoh)

mperham (Mike Perham) wrote:

> I've created a script which attempts to reproduce memory fragmentation.

Great! I could reproduce the issue so easily.

But I can't understand. This program does not use I/O. Due to GVL, it is virtually a single-thread program. I think that malloc lock contention rarely happens. Why does the bloat occur?

#55 - 05/17/2018 05:31 PM - mperham (Mike Perham)

Yusuke, I'm not sure, I can't explain that. It does get worse as the machine gets larger. A machine with more cores will see larger bloat, which is what that graph above shows (36 cores, 40GB -> 9GB); the GVL does not appear to help. The glibc memory allocation internals are documented here:

https://sourceware.org/glibc/wiki/MallocInternals

#56 - 05/18/2018 03:10 AM - bluz71 (Dennis B)

mperham (Mike Perham), mame (Yusuke Endoh), normalperson (Eric Wong),

glibc version 2.10 (2009) malloc was changed to favour scalibility in preference to memory size as noted here. Basically Red Hat and Ulrich Drepper did this to favour large clients (big customers, government agencies) where RAM is plentiful and thread-contention was a real problem.

Prior to 2.10 it seems that MAX_ARENA_MAX=1.
After 2.10 for 32-bit systems it changed to 2 x core-count and for 64-bit systems it changed to 8 x core-count.

This change has since caused a lot of grief for customers in regards to memory usage and fragmentation as noted here:

- https://sourceware.org/bugzilla/show_bug.cgi?id=11261
- https://github.com/prestodb/presto/issues/8993

This has lead to many recommendations:

- https://devcenter.heroku.com/articles/tuning-glibc-memory-behavior
- https://www.ibm.com/developerworks/community/blogs/kevgrig/entry/linux_glibc_2_10_rhel_6_malloc_may_show_excessive_virtual_memory_usage?lang=en
- https://stackoverflow.com/questions/26041117/growing-resident-memory-usage-rss-of-java-process/35610063#35610063

Languages such as Golang and Rust ship their own memory allocators (tcmalloc and jemalloc based respectively). Browsers such as Chrome and Firefox also ship and use their own allocators (tcmalloc/PartitionAlloc and jemalloc respectively). These technologies have their own reasons for doing this, not always related to memory fragmentation (though that is quite important).

Long-lived Ruby processes are a victim of this 2009 change.

#57 - 05/18/2018 03:13 AM - bluz71 (Dennis B)

I have taken Mike's script, increased the THREAD_COUNT to 20 and run it on my Linux box (Intel i5-4590 quad-core, 16GB RAM, Linux Mint 18.3 with kernel 4.15.0).

glibc malloc results:

```
% time MALLOC_ARENA_MAX=1 ruby frag.rb
VmRSS:   4,164,108 kB
real     12.001s

% time MALLOC_ARENA_MAX=2 ruby frag.rb
VmRSS:   4,362,360 kB
real     12.259s

% time MALLOC_ARENA_MAX=3 ruby frag.rb
VmRSS:   4,486,620 kB
real     12.618s

% time MALLOC_ARENA_MAX=4 ruby frag.rb
VmRSS:   4,692,404 kB
real     12.064s

% time MALLOC_ARENA_MAX=5 ruby frag.rb
VmRSS:   4,682,908 kB
real     12.364s

% time MALLOC_ARENA_MAX=6 ruby frag.rb
VmRSS:   4,914,360 kB
real     12.249s

% time MALLOC_ARENA_MAX=7 ruby frag.rb
VmRSS:   5,063,128 kB
real     12.507s

% time MALLOC_ARENA_MAX=8 ruby frag.rb
VmRSS:   5,325,748 kB
real     12.598s

% time MALLOC_ARENA_MAX=12 ruby frag.rb
VmRSS:   5,965,244 kB
real     12.321s

% time MALLOC_ARENA_MAX=16 ruby frag.rb
VmRSS:   6,565,000 kB
real     11.827s

% time MALLOC_ARENA_MAX=24 ruby frag.rb
VmRSS:   7,106,460 kB
real     12.007s

% time MALLOC_ARENA_MAX=32 ruby frag.rb
VmRSS:   7,101,580 kB
real     11.646s

% time ruby frag.rb
VmRSS:   7,241,108 kB
real     11.739s
```

And with jemalloc 3.6.0 (1st stanza) and 5.0.1 (2nd stanza):

```
% time LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libjemalloc.so ruby frag.rb
VmRSS:   4,680,260 kB
real     25.103s

% time LD_PRELOAD=/home/bluz71/.linuxbrew/Cellar/jemalloc/5.0.1/lib/libjemalloc.so ruby frag.rb
VmRSS:   7,162,508 kB
real     12.111s
```

I am not sure if this test is really showing memory fragmentation or arena count expense.

Mike's test clearly runs twice as slow via jemalloc 3.6.0. Conversely when run with jemalloc 5.0.1 the performance issue goes away but the VmRSS size matches high arena count glibc malloc.

I do use jemalloc 3.6.0 with my Rails application with great results. Noah Gibbs noted excellent results when using jemalloc with his Rails benchmark tests as noted here.

Nevertheless we should indeed be cautious. I tend to agree now with backing off using jemalloc by default since different versions have wildly different results with Mike's test.

#58 - 05/18/2018 03:19 AM - bluz71 (Dennis B)

As much as it is a code-smell I too now favour hard-coding M_ARENA_MAX=2 directly in Ruby whilst the GVL is in effect.

Once Guilds land in 2.7/2.8/3.0 (whenever) then M_ARENA_MAX=2 absolutely should be revisited since it will indeed lead to concurrency contention (less arenas leads to higher malloc contention).

But Guilds are not here now and M_ARENA_MAX=2 is an effective strategy for a very real problem.

#59 - 05/18/2018 03:49 AM - mperham (Mike Perham)

Dennis, your results match my results (Ubuntu 18.04, gcc 7.3, glibc 2.27). jemalloc 3.6 is slow but space efficient. jemalloc 5.1 is faster but almost as bad with space as untuned glibc. MALLOC_ARENA_MAX=2 is fast and space efficient (at least with my script). I would be happy to see this issue closed and ruby-core move forward with #14759.

#60 - 05/18/2018 04:10 AM - bluz71 (Dennis B)

I agree Mike.

Close this one and implement #14759 with the caveat to revisit if/when Guilds land.

Good discussion.

P.S. I doubt that glibc will ever be *fixed* since the current behaviour suits the customers Red Hat services and Red Hat themselves are the effective maintainers of glibc.

#61 - 05/18/2018 08:04 AM - normalperson (Eric Wong)

dennisb55@hotmail.com wrote:

> P.S. I doubt that glibc will ever be *fixed* since the current
> behaviour suits the customers Red Hat services and Red Hat
> themselves are the effective maintainers of glibc.

Reading some glibc mailing list (libc-alpha) posts last year,
I remember excessive RSS usage is one of the areas they're
tackling due to user complaints.

Heck, I even seem to even recall a half-hearted proposal a few
years ago to use jemalloc in glibc.

I should also note the newish glibc 2.26 has the thread-caching
malloc as a build-time option, so it should reduce contention on
arenas and allow using fewer arenas.

We also have regular contributions from Red Hat employees
to Ruby, so I think RH does care about Ruby to some degree;
and I don't think the allocation patterns for Ruby would be
too different from a lot of existing software.

(I have no affiliation with Red Hat, past or present)

#62 - 05/19/2018 02:55 AM - bluz71 (Dennis B)

normalperson (Eric Wong) wrote:

> Reading some glibc mailing list (libc-alpha) posts last year,
> I remember excessive RSS usage is one of the areas they're
> tackling due to user complaints.

Interesting, do you have links?

> I should also note the newish glibc 2.26 has the thread-caching
> malloc as a build-time option, so it should reduce contention on
> arenas and allow using fewer arenas.

Also very interesting:

- http://tukan.farm/2017/07/08/tcache/
- https://www.phoronix.com/scan.php?page=news_item&px=glibc-malloc-thread-cache

This is enabled by default; though it will be years before it will be available in LTS releases.

I wonder what the fragmentation characteristics will be?

The arena count defaults haven't changed have they? (8 times core count for x64).

> We also have regular contributions from Red Hat employees
> to Ruby, so I think RH does care about Ruby to some degree;
> and I don't think the allocation patterns for Ruby would be
> too different from a lot of existing software.
>
> (I have no affiliation with Red Hat, past or present)

Red Hat's prime focus will be their paying customers, which is the big end of town (quite rightly). Hence glibc changes tend to favour big-iron rather than Heroku-sized instances.

Hopefully the M_ARENA_MAX=2 change is a part of Ruby 2.6. Whilst GVL exists this change (for Linux only) will be a very big win with no negative consequences. Post Guild is another story.

#63 - 05/22/2018 01:22 AM - vmakarov (Vladimir Makarov)

On 05/18/2018 10:55 PM, dennisb55@hotmail.com wrote:

> Issue #14718 has been updated by bluz71 (Dennis B).
>
> Red Hat's prime focus will be their paying customers, which is the big end of town (quite rightly). Hence glibc changes tend to favour big-iron rather than Heroku-sized instances.
>
> I doubt that this accurately reflects the reality. Cloud is becoming
> very important to Red Hat. OpenShift, for example, contains a lot of
> Ruby code (e.g. Fluentd) which needs to work with very small memory
> footprint. If you check openshift issues, you can find ones aimed to
> decrease memory footprint for Ruby code.

Disclaimer: it is just my personal opinion which hardly depends on the fact that I am a Red Hat employee.

#64 - 05/22/2018 04:25 PM - mperham (Mike Perham)

Another graph from a production Rails app:

Ddv2rvfVMAAM9qu.jpg
https://twitter.com/krasnoukhov/status/998662977891913728

Edit: Not sure why the image isn't showing...

#65 - 05/22/2018 08:22 PM - mperham (Mike Perham)

If jemalloc 5.1.0 is using too much memory, you can tune its arenas in the same way as glibc:

```
MALLOC_CONF=narenas:2,print_stats:true
```

https://github.com/jemalloc/jemalloc/blob/dev/TUNING.md

#66 - 05/23/2018 12:38 AM - bluz71 (Dennis B)

mperham (Mike Perham) wrote:

> If jemalloc 5.1.0 is using too much memory, you can tune its arenas in the same way as glibc:
>
> `MALLOC_CONF=narenas:2,print_stats:true`
>
> https://github.com/jemalloc/jemalloc/blob/dev/TUNING.md

Interesting.

So can we make the observation that low arena count equals low fragmentation but higher thread/malloc contention.

Maybe jemalloc 3.6 used one (or low) arena counts. This gives low fragmentation, but also with certain scripts (such as Yusuke's frag2.rb IO program) low performance.

It may be the case that jemalloc vs glibc performance/fragmentation will not differ that much once arena count are equalised?

Ideally I would like a new Ruby runtime flag --long-lived that was tuned for long run times (e.g low malloc arena count, JIT enabled) vs script/short usages where I want maximum performance.

Sidekiq, Sinatra and Rails usually want low arena counts and JIT.
Scripts want maximum immediate performance always (max arenas and no JIT).

#67 - 05/23/2018 03:04 AM - normalperson (Eric Wong)

dennisb55@hotmail.com wrote:

> normalperson (Eric Wong) wrote:
>
> > Reading some glibc mailing list (libc-alpha) posts last year,
> > I remember excessive RSS usage is one of the areas they're
> > tackling due to user complaints.
>
> Interesting, do you have links?

Wasn't even last year.
https://marc.info/?i=a9367dd0-f130-a23c-df7b-14d50ed10cfa@redhat.com
(or https://public-inbox.org/libc-alpha/a9367dd0-f130-a23c-df7b-14d50ed10cfa@redhat.com/)

```
A high-level concrete problem today with glibc's malloc, and the
only problem be reported by our users is that it consumes too
much RSS. Solving that problem in abstract is what we should be
looking at.
- Carlos O'Donell
```

> > I should also note the newish glibc 2.26 has the thread-caching
> > malloc as a build-time option, so it should reduce contention on
> > arenas and allow using fewer arenas.
>
> I wonder what the fragmentation characteristics will be?

shrug haven't gotten around to testing

> The arena count defaults haven't changed have they? (8 times core count for x64).

Doesn't seem to have changed:

```
git clone git://sourceware.org/git/glibc.git &&
    git -C glibc grep NARENAS_FROM_NCORES
```

#68 - 05/23/2018 04:22 AM - mperham (Mike Perham)

> Ideally I would like a new Ruby runtime flag --long-lived that was tuned for long run times (e.g low malloc arena count, JIT enabled) vs script/short usages where I want maximum performance.
>
> Sidekiq, Sinatra and Rails usually want low arena counts and JIT.

> Scripts want maximum immediate performance always (max arenas and no JIT).

The JVM has had server and client VMs for two decades now? This is the eternal question of trade offs.

> It may be the case that jemalloc vs glibc performance/fragmentation will not differ that much once arena count are equalised?

glibc does not use slab allocation so it will fragment more. MALLOC_ARENA_MAX=2 seems to minimize that fragmentation but jemalloc still consumes less memory overall.

https://en.wikipedia.org/wiki/Slab_allocation

#69 - 05/29/2018 08:34 PM - Ksec (E C)

bluz71 (Dennis B) wrote:

> Redis ships jemalloc 4.0.3 (or near to) as seen here:
>
> https://github.com/antirez/redis/tree/unstable/deps/jemalloc

The latest Redis 5, ships with Jemalloc 5.1

#70 - 06/23/2018 06:39 PM - vo.x (Vit Ondruch)

Fedora and RHEL Ruby maintainer here. Just a few remarks.

mperham (Mike Perham) wrote:

```
Bundle jemalloc and link it statically.

    pro: No runtime hell.
    con: Bloats source distribution. Costs non-glibc users.
    con: Also costs the core devs because they have to sync the bundled jemalloc with the upstream.
```

> This would be my choice as a maintainer and I trust antirez knows what he's doing when he made the same choice for Redis. You control the exact version and any resulting bugs. If you depend on the distro's package, you will integrate with a variety of versions, all with their own set of platform-specific bugs and crash reports. Ubuntu 14 might have jemalloc 3.6.0, 16 might have 4.2.0, 18 might have 5.0.1, etc. Dealing with those platform issues will also cost the core devs.

We are always against bundling. You make great disservice to almost every Linux distribution if you ever suggest/consider bundling.

normalperson (Eric Wong) wrote:

> shyouhei@ruby-lang.org wrote:
>
> > So, I think it's clear people are interested in replacing glibc malloc, not everything that exist in the whole universe.
> >
> > Let's discuss how we achieve that. There can be several ways:
> >
> > - Just enable --with-jemalloc default on, only for Linux.
> >   - pro: This is the easiest to implement.
> >   - pro: Arguably works well. Already field proven.
> >   - con: Mandates runtime dependency for libjemalloc on those systems.
>
> How about only link against it by default if detected. It will
> be like our optional dependency on GMP.

This is the preferred way of course. Specifying --with-jemalloc configure option while I still have to add BuildRequires: jemalloc-devel on the top of the Ruby package just duplicates the effort.

OTOH, if somebody is building Ruby on their system, they want to be in control and be able to disable jemalloc, although the jemalloc-devel is installed. But that is not the case for building package, since in that time we start with minimal buildroot and add just what is necessary.

normalperson (Eric Wong) wrote:

> We will depend on distros to enable/disable the dependency on it.

Thx :) I think that at least in Fedora/RHEL case, the decision will be based on this discussion and on the Ruby defaults, unless some glibc maintainer chimes in and provides some great arguments against.

#71 - 06/25/2018 12:36 AM - nobu (Nobuyoshi Nakada)

vo.x (Vit Ondruch) wrote:

> We are always against bundling. You make great disservice to almost every Linux distribution if you ever suggest/consider bundling.

I think there is no chance to bundle it.

vo.x (Vit Ondruch) wrote:

> OTOH, if somebody is building Ruby on their system, they want to be in control and be able to disable jemalloc, although the jemalloc-devel is installed. But that is not the case for building package, since in that time we start with minimal buildroot and add just what is necessary.

Do you mean --without-jemalloc, or something else?

vo.x (Vit Ondruch) wrote:

> Thx :) I think that at least in Fedora/RHEL case, the decision will be based on this discussion and on the Ruby defaults, unless some glibc maintainer chimes in and provides some great arguments against.

For instance, glibc is going to merge jemalloc? :)

#72 - 07/26/2018 06:51 PM - carlos@redhat.com (Carlos O'Donell)

> vo.x (Vit Ondruch) wrote:
>
> > Thx :) I think that at least in Fedora/RHEL case, the decision will be based on this discussion and on the Ruby defaults, unless some glibc maintainer chimes in and provides some great arguments against.
>
> For instance, glibc is going to merge jemalloc? :)

I am a glibc developer, and project steward.

We have no plans to merge jemalloc.

We have plans to look into glibc's RSS usage under certain workloads including a ruby application which has issues with RSS (fluentd).

In general be careful that allocator performance may vary wildly by workload. We have used glibc in Fedora (in which I also help maintain glibc) for a long time on a wide variety of workloads without problems. There are some workloads for which the heap-based allocator (which glibc is) has RSS problems, partly due to fragmentation, maybe due to infrequent coalescing. We don't really understand the issues, and will be working to root cause the reasons why with the aid of some new tooling (heap dumping, visualization, and analysis). It is clear that page-based allocators (which tcmalloc/jemalloc are) have strong affinity for keeping RSS low and do well in key workloads that matter to modern software designs. Lastly, take care how you calculate "optimal" though since in cloud environments you pay both for CPU *and* MEMORY usage.

In glibc we have a malloc trace project and we have begun collecting trace workloads in an attempt to characterize workloads and be able to replay them in a simulator to look at fragmentation and performance issues (very hard to do, particularly for page touch heuristics). We have some collaboration here with Google's perftools team and tcmalloc, to try and make the tracer and traces generic.

In the end my one recommendation is that you should benchmark thoroughly and make an informed decision based on a corpus of workloads from your users that you have reviewed with both allocators.

I will try to keep the ruby community updated with regard to our analysis of fluentd and the ruby vm with glibc's malloc.

#73 - 07/29/2018 01:18 AM - andresakata (André Guimarães Sakata)

Hi!

I'm another Ruby user that used to have memory bloat problems and switched to jemalloc as well.

I just wrote a simple script (36 lines) that seems to reproduce the issue.

https://github.com/andresakata/ruby-memory-bloat

The script basically creates a FixedThreadPool (depends on concurrent-ruby) and initializes lots of arrays and strings.

It also logs the memory usage at the end of each thread and I'm plotting the numbers below.

Hope it can be useful for the discussion.

glibc 2.23

glibc-plot.jpeg
jemalloc 3.6

jemalloc-plot.jpeg
The data is in the GitHub.

I did the test in a **Ubuntu 16.04**.

#74 - 07/29/2018 04:27 AM - bluz71 (Dennis B)

andresakata (André Guimarães Sakata),

Which version of jemalloc?

Ubuntu 16.04 provides jemalloc 3.6.0 in the repos, is that the version you are testing?

Simple testing some of us have done indicates that newer jemalloc versions will behave **very differently** to 3.6.0. Can you test jemalloc 5.1.0, either compile it by hand or install it via linuxbrew, and post the results please.

Can you also test glibc with MALLOC_ARENA_MAX=2.

Early on with this issue I thought the solution was simple, just move across to jemalloc, but now I realise that there is no easy fix, only headaches whichever way one goes. Ultimately we need the glibc allocator to fragment less since I doubt Ruby core will switch over to jemalloc; and if I were in their shoes now I would not switch over either (just yet).

BUT, lest we sweep this under the carpet, memory fragmentation for long-lived Ruby applications (such as Rails) is a **real** problem that effects the broader Ruby community. This memory behaviour genuinely harms Ruby; but I don't know how we deal with it, maybe hard-coding MALLOC_ARENA_MAX=2 whilst Guilds don't exist yet?

#75 - 07/29/2018 04:31 AM - bluz71 (Dennis B)

carlos@redhat.com (Carlos O'Donell) wrote:

> I will try to keep the ruby community updated with regard to our analysis of fluentd and the ruby vm with glibc's malloc.

Please do keep us informed Carlos.

Thank you, I genuinely hope the initiative produces the desired results we all seek.

#76 - 07/29/2018 07:45 AM - fweimer (Florian Weimer)

I believe this has been reported multiple times against glibc, without ever mentioning Ruby. The most relevant upstream bug seems to be:

https://sourceware.org/bugzilla/show_bug.cgi?id=14581

The Ruby allocator calls posix_memalign (16384, 16344), and unfortunately, such allocations, when freed, can not always be reused for subsequent allocations because the glibc allocator will not search for existing unused aligned allocations of a given size.

#77 - 07/29/2018 08:38 AM - fweimer (Florian Weimer)

andresakata (André Guimarães Sakata) wrote:

> Hi!
>
> I'm another Ruby user that used to have memory bloat problems and switched to jemalloc as well.
>
> I just wrote a simple script (36 lines) that seems to reproduce the issue.
>
> https://github.com/andresakata/ruby-memory-bloat
>
> The script basically creates a FixedThreadPool (depends on concurrent-ruby) and initializes lots of arrays and strings.

Thanks. Would it be possible to eliminate the dependency on newrelic_rpm, and possibly Rails as well? We don't have a commercial newrelic_rpm license, and we are not Ruby developers, so this is not easy for us to do. (The concurrent dependency is not a problem.)

#78 - 07/29/2018 03:16 PM - andresakata (André Guimarães Sakata)

fweimer (Florian Weimer) wrote:

> Thanks. Would it be possible to eliminate the dependency on newrelic_rpm, and possibly Rails as well? We don't have a commercial newrelic_rpm license, and we are not Ruby developers, so this is not easy for us to do. (The concurrent dependency is not a problem.)

Hi @fweimer. You don't have to have a NewRelic license and it doesn't depend on Rails. I added the newrelic_rpm gem because they have a function to get the current RSS that works on different platforms. Is it a problem yet?

I'm working on a README and creating some new files to make it easier for anyone to run the test.

#79 - 07/29/2018 04:34 PM - andresakata (André Guimarães Sakata)

- File jemalloc-3-5.log added

- File glibc.log added

- File glibc-arena-2.log added

- File glibc.png added

- File glibc_arena_2.png added

- File jemalloc.png added

I removed the NewRelic dependency and it's using the get_process_mem gem instead. It should work on different platforms as well.

Now there is a README file in the repository that may help you to reproduce the tests. Let me know you have any issue.

Did some of the tests proposed by bluz71 (Dennis B) in a **Ubuntu 14.04**.

Using glibc 2.19

```
Min.   :143.3
1st Qu.:1606.7
Median :1801.5
Mean   :1677.5
3rd Qu.:1943.0
Max.   :2158.4
```

Using glibc 2.19 MALLOC_ARENA_MAX=2

```
Min.   :143.0
1st Qu.:608.3
Median :608.8
Mean   :592.5
3rd Qu.:618.3
Max.   :671.2
```

Using jemalloc 3.5

```
Min.   :113.5
1st Qu.:356.7
Median :374.8
Mean   :376.0
3rd Qu.:407.4
Max.   :645.2
```

Didn't make tests with jemalloc 5.1 yet...

#80 - 07/29/2018 05:39 PM - andresakata (André Guimarães Sakata)

bluz71 (Dennis B) wrote:

> andresakata (André Guimarães Sakata),
>
> Which version of jemalloc?
>
> Ubuntu 16.04 provides jemalloc 3.6.0 in the repos, is that the version you are testing?

I had forgotten to answer it. Yes, in that test I was using jemalloc 3.6.0.

#81 - 07/30/2018 10:22 AM - normalperson (Eric Wong)

fweimer@redhat.com wrote:

> https://sourceware.org/bugzilla/show_bug.cgi?id=14581
>
> The Ruby allocator calls posix_memalign (16384, 16344), and
> unfortunately, such allocations, when freed, can not always be
> reused for subsequent allocations because the glibc allocator

> will not search for existing unused aligned allocations of a
> given size.

Thanks for bringing this to our attention.

I wonder how beneficial it is for Ruby to free the memalign-ed
sections used for object slots. Each of those allocations
is "only" 400 or so objects and apps churn through that
quickly, so.

Testing the following patch with
"MALLOC_ARENA_MAX=1 MALLOC_ARENA_TEST=1 make gcbench-rdoc"
seems show a small improvement in VmHWM across repeated
runs, but the results aren't stable...

https://80x24.org/spew/20180730085724.29644-1-e@80x24.org/raw

The other problems are we hit malloc a lot and we use native
threads (despite the GVL) in unnecessary ways; and having a
process-wide GC means the associated free() for an allocation
can happen from a thread the allocation didn't come from.

Having multiple malloc arenas to avoid contention isn't very
useful with the GVL, either.

Anyways I'm working on several topics to reduce unnecessary
uses of native threads; and ko1 is introducing transient heap
to deal with short-lived allocations. So I'm hoping these
ideas pan out and we can put less stress the on malloc
implementation.

#82 - 07/30/2018 12:16 PM - fweimer (Florian Weimer)

I can reproduce your MALLOC_ARENA_MAX=2 number.

But my untuned glibc 2.26 numbers are slightly worse (presumably due to the thread cache, which delays coalescing even further):

```
count    5000.000000
mean     2299.963311
std       589.499758
min       143.515625
25%      2120.331055
50%      2539.085938
75%      2728.136719
max      2887.113281
```

With jemalloc 4.5, I get very similar numbers to glibc 2.26:

```
count    5000.000000
mean     2137.653509
std       397.754364
min       178.406250
25%      2053.171875
50%      2148.148438
75%      2347.949219
max      2571.218750
```

jemalloc 5.0.1 may be slightly better, but it is not a large change:

```
count    5000.000000
mean     1844.086273
std       338.717582
min       176.300781
25%      1848.468750
50%      1935.281250
75%      2011.210938
max      2158.937500
```

jemalloc 5.1 is slightly worse, it seems.

```
count    5000.000000
mean     2027.594097
std       365.946724
min       176.437500
```

```
25%      1967.921875
50%      2073.148438
75%      2245.913086
max      2508.105469
```

Based on these tests, untuned current jemalloc shows only a very moderate improvement over untuned glibc.

#83 - 07/30/2018 01:16 PM - andresakata (André Guimarães Sakata)

Hello @fewimer, very interesting.

Just to understand one thing, were you using MALLOC_ARENA_MAX=2 in your first test?

#84 - 07/30/2018 01:20 PM - fweimer (Florian Weimer)

andresakata (André Guimarães Sakata) wrote:

> Hello @fewimer, very interesting.
>
> Just to understand one thing, were you using MALLOC_ARENA_MAX=2 in your first test?

In the first test, yes, but I did not quote those numbers. The first quoted numbers are for untuned glibc 2.26.

#85 - 07/30/2018 04:16 PM - andresakata (André Guimarães Sakata)

Got it, thanks @fweimer.

I did run a test with **jemalloc 5.1** and I've got the same results as you did.

```
Min.   :242.9
1st Qu.:1593.5
Median :1737.7
Mean   :1691.5
3rd Qu.:1860.0
Max.   :2127.9
```

Remembering my previous results in the same environment below.

glibc 2.19

Sorry about the too old version here... but I've got similar differences between glibc 2.23 and jemalloc 3.6 (comparing to glibc 2.19 and jemalloc 3.5).

```
Min.   :143.3
1st Qu.:1606.7
Median :1801.5
Mean   :1677.5
3rd Qu.:1943.0
Max.   :2158.4
```

glibc 2.19 MALLOC_ARENA_MAX=2

```
Min.   :143.0
1st Qu.:608.3
Median :608.8
Mean   :592.5
3rd Qu.:618.3
Max.   :671.2
```

jemalloc 3.5

```
Min.   :113.5
1st Qu.:356.7
Median :374.8
Mean   :376.0
3rd Qu.:407.4
Max.   :645.2
```

#86 - 07/30/2018 08:27 PM - davidtgoldblatt (David Goldblatt)

I don't think this benchmark is a useful way to compare performance between versions 3 and 5 of jemalloc. In between them was the advent of time-based purging, where the allocator waits for a while (by default, around 10 seconds) before returning memory to the OS. That purging occurs (well, in the default but not recommended case where background threads are disabled) on an operation counter tick. By having low activity for a short program lifetime, it effectively disables returning memory to the OS, in a way that wouldn't actually persist if you were running a more realistic program. Even outside of that, we've started using MADV_FREE, which gives the kernel the right to reclaim memory, but in such a way that it will only

do so if the machine is actually under memory pressure; this won't show up in test runs unless you're trying to make it happen.

You could verify this by setting dirty decay and muzzy decay to 0 in the MALLOC_CONF environment variable (i.e. MALLOC_CONF="dirty_decay_ms:0,muzzy_decay_ms:0", unless I've typoed something).

In general, malloc benchmarking is notoriously difficult to do usefully (outside of ruling out certain kinds of pathological behavior, such as in an internal allocator test suite). Better would be some representative suite of allocation-heavy test programs that accomplish some real task.

#87 - 07/30/2018 10:46 PM - andresakata (André Guimarães Sakata)

davidtgoldblatt (David Goldblatt) wrote:

> I don't think this benchmark is a useful way to compare performance between versions 3 and 5 of jemalloc. In between them was the advent of time-based purging, where the allocator waits for a while (by default, around 10 seconds) before returning memory to the OS. That purging occurs (well, in the default but not recommended case where background threads are disabled) on an operation counter tick. By having low activity for a short program lifetime, it effectively disables returning memory to the OS, in a way that wouldn't actually persist if you were running a more realistic program. Even outside of that, we've started using MADV_FREE, which gives the kernel the right to reclaim memory, but in such a way that it will only do so if the machine is actually under memory pressure; this won't show up in test runs unless you're trying to make it happen.
>
> You could verify this by setting dirty decay and muzzy decay to 0 in the MALLOC_CONF environment variable (i.e. MALLOC_CONF="dirty_decay_ms:0,muzzy_decay_ms:0", unless I've typoed something).
>
> In general, malloc benchmarking is notoriously difficult to do usefully (outside of ruling out certain kinds of pathological behavior, such as in an internal allocator test suite). Better would be some representative suite of allocation-heavy test programs that accomplish some real task.

Hello David. The tests took more than one minute to finish. Does it change anything?

Let me know if it is useful for you to see the memory usage distribution over the time during the test. They are completely different in these versions. But as this subject is out of my domain I can't take any conclusion about it.

Also, I believe we have to be able to make some simple, reproducible and representative tests on this. Otherwise, we can't make reasonable decisions here. Until now, the comparison of jemalloc 3.x and glibc 2.x matches what we have seen in our production apps... but maybe there's more to do.

#88 - 07/31/2018 12:31 AM - bluz71 (Dennis B)

davidtgoldblatt (David Goldblatt) wrote:

> I don't think this benchmark is a useful way to compare performance between versions 3 and 5 of jemalloc. In between them was the advent of time-based purging, where the allocator waits for a while (by default, around 10 seconds) before returning memory to the OS. That purging occurs (well, in the default but not recommended case where background threads are disabled) on an operation counter tick. By having low activity for a short program lifetime, it effectively disables returning memory to the OS, in a way that wouldn't actually persist if you were running a more realistic program. Even outside of that, we've started using MADV_FREE, which gives the kernel the right to reclaim memory, but in such a way that it will only do so if the machine is actually under memory pressure; this won't show up in test runs unless you're trying to make it happen.
>
> You could verify this by setting dirty decay and muzzy decay to 0 in the MALLOC_CONF environment variable (i.e. MALLOC_CONF="dirty_decay_ms:0,muzzy_decay_ms:0", unless I've typoed something).

Very interesting indeed.

I suspect many of us were not aware of that, I certainly wasn't. All the micro-testing conducted here and at #14759 indicated a large regression in jemalloc RSS behaviour post 3.X series.

This is a difficult one. I wonder if we need to transistion across to Noah Gibbs Rails benchmark?

Nevertheless this usage graph posted by Micke Perham is indicative of what many users experience with long-lived production Ruby applications on Linux:

jemalloc.jpg

#89 - 08/01/2018 12:14 AM - sam.saffron (Sam Saffron)

After spending a bit too much time thinking about this, I would like to recommend **against** any jemalloc related changes and instead to double down on Eric's https://bugs.ruby-lang.org/issues/14759 for the next release of Ruby (which I think should be backported to 2.5/2.4)

As much as we like to think of jemalloc as a silver bullet of sorts... there are problems... In particular if you have THP enabled which most people do out of the box its behavior is not ideal despite all the fixes it has gotten over the years. My observation is that both MALLOC_ARENA_MAX=2 and tcmalloc can perform better if THP is on, both in 5.1 and 3.6.0. Getting the world to turn off THP is a hard job.

Further to this jemalloc probably has too many arenas out of the box and should only need 2 or so for optimal perf.

Since it is so hairy and so application specific my vote here is merge and backport https://bugs.ruby-lang.org/issues/14759 ASAP. We can teach people about jemalloc quirks elsewhere but Ruby does not need to go down this path. Better just work with glibc here.

7409f18667a24ed6643457377e475a738164e952.png

#90 - 08/01/2018 12:55 AM - mperham (Mike Perham)

Sam, I'm ok with your suggestion, any progress here is welcome. The main issue with tcmalloc is that Ruby doesn't support it out of the box with a --with-tcmalloc flag. Are you using LD_PRELOAD instead?

#91 - 08/01/2018 01:04 AM - normalperson (Eric Wong)

> Testing the following patch with
> "MALLOC_ARENA_MAX=1 MALLOC_ARENA_TEST=1 make gcbench-rdoc"
> seems show a small improvement in VmHWM across repeated
> runs, but the results aren't stable...
>
> ```
> https://80x24.org/spew/20180730085724.29644-1-e@80x24.org/raw
> ```

Maybe https://bugs.ruby-lang.org/issues/14955 to use MADV_FREE directly is easier to stomach for memory-constrained systems.

> The other problems are we hit malloc a lot and we use native
> threads (despite the GVL) in unnecessary ways; and having a
> process-wide GC means the associated free() for an allocation
> can happen from a thread the allocation didn't come from.

I'm not sure how much it'll help common Ruby apps with GVL;
but mwrap[1] users should see an improvement by adding wfcqueue
support to glibc malloc:

https://public-inbox.org/libc-alpha/20180731084936.g4yw6wnvt677miti@dcvr/T/

[1] git clone https://80x24.org/mwrap.git
(uses call_rcu for real free(3))

#92 - 08/01/2018 02:12 AM - sam.saffron (Sam Saffron)

> Are you using LD_PRELOAD instead?

Yes that is how we deploy PRD we compile without jemalloc and then just ld preload various libs via environment depending on what we are testing.

btw... jemalloc with THP off is really good, its just once THP is on ... stuff can get weird even on latest.

#93 - 08/01/2018 03:41 AM - bluz71 (Dennis B)

sam.saffron (Sam Saffron) wrote:

> After spending a bit too much time thinking about this, I would like to recommend **against** any jemalloc related changes and instead to double down on Eric's https://bugs.ruby-lang.org/issues/14759 for the next release of Ruby (which I think should be backported to 2.5/2.4)

I was as big a supporter of changing to jemalloc as anyone (and I still use it in production myself); but a little while ago I came to the same conclusion. Ruby should not default to using jemalloc.

Implementing #14759 is the appropriate, and safe, medium-term solution **until** Guilds land. In the meantime hopefully the glibc folks can do their analysis and improve RSS behaviour especially with the Ruby-based Fluentd application causing grief to the Red Hat folks (an excellent pain point motivator to fix glibc). If glibc were fixed then M_ARENA_MAX=2 should only be applied to older glibc versions effected, via a runtime check?

I think most of the core promoters of this issue: Mike Perham, Sam, Eric and myself would prefer that 14759 be done instead, and possibly close this one?

#94 - 08/01/2018 10:04 AM - normalperson (Eric Wong)

sam.saffron@gmail.com wrote:

> As much as we like to think of jemalloc as a silver bullet of
> sorts... there are problems... In particular if you have THP

enabled which most people do out of the box its behavior is
not ideal despite all the fixes it has gotten over the years.
My observation is that both MALLOC_ARENA_MAX=2 and tcmalloc
can perform better if THP is on, both in 5.1 and 3.6.0.
Getting the world to turn off THP is a hard job.

Ruby 2.6 will disable THP, at least:

https://bugs.ruby-lang.org/issues/14705

#95 - 08/02/2018 12:04 AM - sam.saffron (Sam Saffron)

Wow awesome news normalperson (Eric Wong) this is going to make a massive difference with Ruby 2.6!

#96 - 02/07/2019 03:53 AM - mame (Yusuke Endoh)

FYI: Rust dropped jemalloc and switched the default to the system allocator.

https://blog.rust-lang.org/2019/01/17/Rust-1.32.0.html#jemalloc-is-removed-by-default

#97 - 02/09/2019 04:42 AM - bluz71 (Dennis B)

Very interesting news indeed about Rust. Thanks for the update.

Ruby 2.6 came and went, allocator remained unchanged though two nice enhancements were incorporated, transient-heap and disabled THP. Sam Saffron tweeted the other day about improved performance of 2.6 with Discourse.

Anyway, I really hope the Red Hat / glibc crew are doing work to improve fragmentation behaviour (as noted above).

Not changing allocator for Ruby was the correct choice. I suspect this issue can be closed, it is unlikely to ever happen.

Until glibc improves I am still a proponent of #14759*set M_ARENA_MAX for glibc malloc*. Eric, are you still wanting to do that?

#98 - 03/14/2019 04:52 PM - RubyBugs (A Nonymous)

Does Hongli Lai's article What causes Ruby memory bloat? shed any potential light here?
https://www.joyfulbikeshedding.com/blog/2019-03-14-what-causes-ruby-memory-bloat.html#a-magic-trick-trimming

After investigation, he discovered that patching Ruby to add into gc_prof_timer_start a call to malloc_trim(0); demonstrated significant impact, on the same order as using jemalloc in his test application -- and with an attendant *speed up*, rather than the slow down one might anticipate, on the Rails Ruby Bench.

(I came here to post this because I've been following this issue for some time, and hoped this additional information might be helpful to others)

#99 - 03/14/2019 08:02 PM - fredngo (Fred Ngo)

Came here to find out if indeed malloc_trim() would be integrated into the garbage collector. If Hongli's findings are true, this is indeed a game changer. Looking forward to seeing the bloat in my apps disappear!

#100 - 03/14/2019 08:35 PM - sam.saffron (Sam Saffron)

I created https://bugs.ruby-lang.org/issues/15667 to track malloc_trim, please post any results you have regarding to production performance there

#101 - 03/15/2019 06:50 AM - bluz71 (Dennis B)

Outstanding research by Hongli Lai, and superbly presented in a post & video that all of us can easily digest.

Sam's new 15667 issue will be very interesting. I tend to agree, if all goes well, it should be a candidate for 2.5 & 2.6 back-ports.

The results of that should also result in the closure of this issue and the [PATCH] set M_ARENA_MAX for glibc malloc #14759 issue.

Thank you Hongli (if you are reading this).

#102 - 03/25/2020 06:38 PM - hackeron (Roman Gaufman)

I just tried this and saw dramatic reduction in memory use per time. Why isn't this the default??

Files

glibc_arena_2.png	7.23 KB	07/29/2018	andresakata (André Guimarães Sakata)
jemalloc.png	21.1 KB	07/29/2018	andresakata (André Guimarães Sakata)
glibc-arena-2.log	60.3 KB	07/29/2018	andresakata (André Guimarães Sakata)
glibc.log	62.3 KB	07/29/2018	andresakata (André Guimarães Sakata)

jemalloc-3-5.log	58.5 KB	07/29/2018	andresakata (André Guimarães Sakata)
glibc.png	9.03 KB	07/29/2018	andresakata (André Guimarães Sakata)