Ruby master - Misc #14222

Mutex.lock is not safe inside signal handler: what is?

12/22/2017 01:00 PM - hkmaly (Honza Maly)

Status:	Closed	
Priority:	Normal	
Assignee:	normalperson (Eric Wong)	

Description

As mentioned in #7917, Mutex.lock is not safe inside signal handler. As mentioned in #6416, neither is Thread.join. But there seem to be no documentation about what IS safe to do inside signal handler. In C, it is not safe to just modify variable inside signal handler without locking. Is it safe in ruby in case of 1) global variable 2) class variable 3) object variable 4) thread local variable, as in Thread.current['local_var'] ? Is there any other method of locking usable inside trap content?

Note that while response in this issue would be useful it would be even better if it appeared in official ruby documentation, if there is any. I realize it is not directly bug but it's likely going to be cause of many bugs if the answer is "no" on any part of this and only people who understand ruby core can answer this.

Associated revisions

Revision ea675ee4 - 01/28/2018 09:57 PM - normal

doc/signals.rdoc: new document work-in-progress

We need a longer document to inform users of caveats
related to Signal.trap usage. This is still incomplete,
and we can fill in and edit other bits as needed.

- doc/signals.rdoc: new document [ruby-core:85107] [Misc #14222]

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@62083 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 62083 - 01/28/2018 09:57 PM - normalperson (Eric Wong)

doc/signals.rdoc: new document work-in-progress

We need a longer document to inform users of caveats
related to Signal.trap usage. This is still incomplete,
and we can fill in and edit other bits as needed.

- doc/signals.rdoc: new document [ruby-core:85107] [Misc #14222]

Revision 62083 - 01/28/2018 09:57 PM - normal

doc/signals.rdoc: new document work-in-progress

We need a longer document to inform users of caveats
related to Signal.trap usage. This is still incomplete,
and we can fill in and edit other bits as needed.

- doc/signals.rdoc: new document [ruby-core:85107] [Misc #14222]

History

#1 - 12/22/2017 01:11 PM - hkmaly (Honza Maly)

Few more questions: is adding key to hash safe? Pushing variable to array as used in
http://www.mikeperham.com/2013/02/23/signal-handling-with-ruby/ ?

#2 - 12/23/2017 01:08 AM - normalperson (Eric Wong)

hkmaly@matfyz.cz wrote:

> As mentioned in #7917, Mutex.lock is not safe inside signal
> handler. As mentioned in #6416, neither is Thread.join. But
> there seem to be no documentation about what IS safe to do
> inside signal handler. In C, it is not safe to just modify
> variable inside signal handler without locking. Is it safe in
> ruby in case of 1) global variable 2) class variable 3) object
> variable 4) thread local variable, as in
> Thread.current['local_var'] ? Is there any other method of

> locking usable inside trap content?

Actually, in C it is safe to set variables with the "sig_atomic_t" type without locking. Atomic instructions (C11 or __sync_* in gcc) are also safe. In fact, it is never safe to use locking such as pthread_mutex_lock inside a signal handler.

I have limited experience with the following, but I believe Thread.handle_interrupt in Ruby 2.0+ can be used for this:

```
trap(:INT) do
  Thread.handle_interrupt(Interrupt => :never) do
    # anything goes
  end
end
```

I often deal with legacy projects which still run on 1.9.3 (or even 1.8 :x), so I've done some wacky stuff:

```
trap(:INT) do
  # fire-and-forget thread
  Thread.new { something_which_locks_a_mutex }
end
```

But usually, I stick to the basic stuff which doesn't take locks: IO#syswrite, or IO#write_nonblock where IO#sync=true. Buffered I/O is not allowed since it takes locks, same with most stdio stuff in C. I also know that Array#<<, Hash#[]= are alright at least in CRuby. Thread#[]= is probably alright, too

For ruby, assigning local variables is fine, hooked variables (some globals) might not be, and maybe all class+ivars are OK. It may be Ruby implementation dependent, though...

> Note that while response in this issue would be useful it would be even better if it appeared in official ruby documentation, if there is any. I realize it is not directly bug but it's likely going to be cause of many bugs if the answer is "no" on any part of this and only people who understand ruby core can answer this.

Maybe we can mark functions with reentrancy and thread-safety levels in our RDoc like some *nix manpages do. Other Ruby implementations would need to follow if CRuby defines it as spec.

Right now my personal take is to read the source code of the methods I use and look for things it does under-the-hood; but it's probably not for everyone.

#3 - 12/23/2017 07:43 AM - nobu (Nobuyoshi Nakada)

Thread.handle_interrupt doesn't work.

You can use Queue inside trap context.

```
require 'logger'

LOG = Queue.new
Thread.start {
  log = Logger.new(STDOUT)
  log.info "Now logging!"
  nil while log.info(LOG.pop)
}

trap :INT do
  LOG << "Hello"
end

gets
```

#4 - 12/23/2017 08:08 AM - normalperson (Eric Wong)

nobu@ruby-lang.org wrote:

> Thread.handle_interrupt doesn't work.

Oops, the main thread may already have a Mutex locked, right?

> You can use Queue inside trap context.

Since 2.1 when Queue was reimplemented in C. Old (pure-Ruby)
Queue used Mutex, but I guess those versions are no longer
supported (and I still have old crap on 1.9.3 :<)

#5 - 12/25/2017 10:41 PM - kirs (Kir Shatrov)

normalperson (Eric Wong) wrote:

> In fact, it is never safe to
> use locking such as pthread_mutex_lock inside a signal handler.

Do you think it's likely that your work towards Mutex that does not rely on threads (r13517) would allow us to use Ruby Mutex from a signal trap? As far as I understand, that would mean getting rid of pthread_mutex_lock on CRuby.

#6 - 12/25/2017 11:42 PM - normalperson (Eric Wong)

shatrov@me.com wrote:

> normalperson (Eric Wong) wrote:
>
> > In fact, it is never safe to
> > use locking such as pthread_mutex_lock inside a signal handler.
>
> Do you think it's likely that your work towards Mutex that
> does not rely on threads (r13517) would allow us to use Ruby
> Mutex from a signal trap? As far as I understand, that would
> mean getting rid of pthread_mutex_lock on CRuby.

No, the principle for the mutex rewrite [Feature #13517] is the
same as any other simple (fast) mutex implementation, so the same
caveats apply(*).

I think what you're looking for is a reentrant (or recursive) mutex.
I haven't looked into those in many years; but from what I recall,
they were generally discouraged for being tricky to use.

Not speaking for matz and ko1, but I recall them not liking Mutexes
in the language. So reentrant mutexes would probably not be welcome.

(*) The work done on Mutex was to make them more sympathetic to
the current VM and GVL by removing redundancies. It still
relies on the GVL (which uses pthread_mutex_lock), and the
implementation will need to evolve as the VM evolves.
One side-effect of those changes is the current iteration
can be easily ported for use with green threads (Thriber)
in case that feature is accepted.

#7 - 12/27/2017 07:07 PM - Eregon (Benoit Daloze)

Could MRI simply use the self-pipe trick or similar internally, and execute signal handlers on the main thread by using Ruby's interrupts? (RUBY_VM_CHECK_INTS)
A bit like Thread#raise except it would execute the signal handler instead of throwing an exception.
That would allow to run any kind of code in signal handlers.
The timer thread could listen for signals and interrupt the main thread to execute the handler, since that code is likely not async-signal safe.

Of course, it doesn't solve the problem of calling SOME_LOCK.lock in the handler if SOME_LOCK is already locked by the main Thread.
But that seems rarely a problem, and internal locks could be reentrant (e.g. IO locks).
Having a not-well defined set of allowed operations in a Ruby block (the signal handler) seems a much larger problem worth fixing.

P.S.: That's what is done in TruffleRuby essentially.

#8 - 12/27/2017 08:08 PM - normalperson (Eric Wong)

eregontp@gmail.com wrote:

> Could MRI simply use the self-pipe trick or similar internally, and execute signal handlers on the main thread by using Ruby's interrupts? (RUBY_VM_CHECK_INTS)
> A bit like Thread#raise except it would execute the signal handler instead of throwing an exception.
> That would allow to run any kind of code in signal handlers.
> The timer thread could listen for signals and interrupt the main thread to execute the handler, since that code is likely not async-signal safe.

We already do this. If we didn't, hardly any Ruby code would be
safe inside a signal handler, not even creating a string.

> Of course, it doesn't solve the problem of calling SOME_LOCK.lock in the handler if SOME_LOCK is already locked by the main Thread.

Exactly!

> But that seems rarely a problem, and internal locks could be reentrant (e.g. IO locks).

One could say most bugs are rarely a problem... but there is no
place for optimism when the goal is robust software ;)

Reentrant locks aren't cheap, either; so I don't think they're
good for something as common as IO operations. Not to mention
they are tricky and probably lead to bad design
(require/autoload uses them, and yes, it's tricky and we've had
problems there).

> Having a not-well defined set of allowed operations in a Ruby block (the signal handler) seems a much larger problem worth fixing.

Maybe we can list out safe methods for signal handlers
somewhere. Maybe create doc/signals.rdoc? signal(7) in Linux
manpages has a similar list. I can help review/maintain it.

#9 - 01/25/2018 10:00 AM - Eregon (Benoit Daloze)

- Assignee changed from ruby-core to normalperson (Eric Wong)

normalperson (Eric Wong) wrote:

> > Having a not-well defined set of allowed operations in a Ruby block (the signal handler) seems a much larger problem worth fixing.
>
> Maybe we can list out safe methods for signal handlers
> somewhere. Maybe create doc/signals.rdoc? signal(7) in Linux
> manpages has a similar list. I can help review/maintain it.

I think that would be very helpful and help to discuss possible improvements.
Also explaining why Mutex is problematic (the signal handler can be run between any 2 lines of code, and Mutex is not re-entrant).
Can I assign this to you to create the document?

#10 - 01/27/2018 09:41 AM - normalperson (Eric Wong)

eregontp@gmail.com wrote:

> I think that would be very helpful and help to discuss possible improvements.
> Also explaining why Mutex is problematic (the signal handler can be run between any 2 lines of code, and Mutex is not re-entrant).
> Can I assign this to you to create the document?

Sure; quite a bit of work and yes, I'm noticing some ickiness
along the way...

Ugh, I still hate how big rb_io_t is and the presence of write_lock +
rb_mutex_allow_trap does not make me comfortable(*) :<
(Fortunately pipes and sockets default to IO#sync=true)

I know some code uses Mutex#synchronize to cover a huge scope
nowadays, but maybe disabling Signal.trap blocks from firing
while any Mutex is locked gets rid of a huge chunk of problems
(at the expensive of new ones)

Or we could've had lock-free operations from the start and never
had a Mutex class in the first place :>

#11 - 01/28/2018 09:57 PM - Anonymous

- Status changed from Open to Closed

Applied in changeset trunk|r62083.

---

doc/signals.rdoc: new document work-in-progress

We need a longer document to inform users of caveats
related to Signal.trap usage. This is still incomplete,
and we can fill in and edit other bits as needed.

- doc/signals.rdoc: new document [ruby-core:85107] [Misc #14222]

#12 - 01/28/2018 10:03 PM - normalperson (Eric Wong)

Definitely needs work, but r62083 is a start for now.