

Ruby - Feature #17849

Fix Timeout.timeout so that it can be used in threaded Web servers

05/05/2021 01:41 AM - duerst (Martin Dürst)

Status:	Assigned	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:		
Description Making this a separate issue from #17837 Eregon (Benoit Daloze) wrote in https://bugs.ruby-lang.org/issues/17837#note-10 (which is about timeouts for regular expressions): I think fixing Timeout.timeout might be possible. The main/major issue is it can trigger within ensure, right? Is there anything else? We could automatically mask Thread#raise within ensure so it only happens after the ensure body completes. And we could still have a larger "hard timeout" if an ensure takes way too long (shouldn't happen, but one cannot be sure). I recall discussing this with @schneems (Richard Schneeman) some time ago on Twitter.		
Related issues: Related to Ruby - Feature #17837: Add support for Regexp timeouts Closed Related to Ruby - Feature #17363: Timeouts Assigned Related to Ruby - Bug #13876: Tempfile's finalizer can be interrupted by a Ti... Closed		

History

#1 - 05/05/2021 01:41 AM - duerst (Martin Dürst)

- Related to Feature #17837: Add support for Regexp timeouts added

#2 - 05/05/2021 07:54 AM - byroot (Jean Boussier)

We could automatically mask Thread#raise within ensure so it only happens after the ensure body completes.

As said on the other issue, I don't think it would fix *all* issues, but it would certainly be a huge improvement.

#3 - 05/12/2021 03:06 PM - schneems (Richard Schneeman)

I've had a conversation with Matz about this and I've been thinking about this issue for a LONG time. The blocker to doing something like this before is that due to the halting problem we can never know if an ensure block will exit or not. In effect, if someone were to use Thread#safe_raise (or something similar) where the exception didn't fire inside an ensure block: It might accidentally end up with your program stuck in an infinite loop.

My proposal to avoid this would be to add an additional timer (yes, timers all the way down) to fire an "overtime" block. For example, you could specify the program should wait 1 second to clear all ensure blocks, if it doesn't then execute some code (such as raising an error or emitting a warning). Here's my stab at an API:

```
Timeout.safe_timeout(10, overtime: 1, on_overtime: :warn || :error) do
  # ...
end
Timeout.safe_timeout(10, overtime: 1, on_overtime: -> { puts "Overtime reached" } ) do
  # ...
end
```

Here's the twitter thread that's referenced in the prior conversation <https://mobile.twitter.com/schneems/status/1377340755878965248>.

I've written at length about the problem here <https://www.schneems.com/2017/02/21/the-oldest-bug-in-ruby-why-racktimeout-might-hose-your-server/>

#4 - 05/13/2021 10:02 AM - duerst (Martin Dürst)

- Assignee set to matz (Yukihiro Matsumoto)

Assigning to Matz because this may involve a new method.

#5 - 05/14/2021 06:15 PM - Eregon (Benoit Daloze)

Another way to express the idea is that `ensure; code(); end` would automatically behave like `Thread.handle_interrupt(Object => :never) { code }`. Since that would be directly in the interpreter and not actually call `Thread.handle_interrupt`, there is no worry that the interrupt happens during the call to `Thread.handle_interrupt` (which would be an issue when writing `ensure; Thread.handle_interrupt { ... }; end`).

byroot (Jean Boussier) wrote in [#note-12](#):

That would fix most issues, but not all. It can also trigger in places where exception are entirely unexpected, so there's just no `ensure`.

Do you have an example?

Also I'm not clear on the details, but some C extensions (often various clients) can't be interrupted by `Thread#raise`.

If that's the case I would argue it's a bug of the C extension (for blocking without an unblocking function). It seems very much expected that `Ctrl+C` always works in Ruby.

#6 - 05/14/2021 06:16 PM - Eregon (Benoit Daloze)

- Related to Feature #17363: Timeouts added

#7 - 05/14/2021 06:22 PM - Eregon (Benoit Daloze)

This is also a bit related to the proposed `Timeout.wake` in [#17363](#), but that would only interrupt blocking methods and it's unclear if that includes Regexp matching (probably not since it's CPU work, not waiting on anything external), and it would definitely not include code that does not block but runs for too long (e.g. a very long loop).

Long story short, this proposal would be general, `Timeout.wake` would not and would only address blocking methods.

#8 - 05/14/2021 06:28 PM - byroot (Jean Boussier)

Do you have an example?

I don't have a specific example in mind, but for instance things like:

```
module Namespace
  extend self

  def clear_something
    @something, old_something = Something.new, @something
    @something.clear # if we raise here we might leak or whatever
  end
end
```

The `setup / yield / ensure / clean` pattern is extremely frequent and just fixing this one would be a huge improvement. I just want to make it clear that it won't be sufficient to eliminate all possible woes.

If that's the case I would argue it's a bug of the C extension

Sure. But it can similarly be argued (and I heard it before) that a request that doesn't complete in a reasonable time is a bug in the application.

But in my opinion web server timeouts are exactly that, a safety net against bugs in the application. That's exactly the kind of cases I want a timeout for.

#9 - 05/19/2021 06:13 PM - schneems (Richard Schneeman)

Assignee set to `matz` (Yukihiro Matsumoto)

Maybe we want to tag in Koichi. Last I talked to Matz (EuRuKo in 2019) I believe he suggested having Koichi take a look at the issue. I'm not familiar with the interface here I don't know if we can `@` mention someone or if we need to set them to the owner.

#10 - 05/19/2021 07:34 PM - Eregon (Benoit Daloze)

[@byroot \(Jean Boussier\)](#) For the blocking C extension code without an unblocking function case, I think there is no other way to unstuck it than to kill the entire process.

That's probably a good last measure to have, but obviously it's very costly so should only be used when there is no other way.

#11 - 05/20/2021 08:44 AM - ko1 (Koichi Sasada)

There are two issues on masking interrupts in ensure clauses:

- (1) performance: manipulating masks in each ensure clauses introduces overhead
- (2) Some applications can run their program body in ensure clauses.

Before we discussed about it, we can't decide to mask interrupts because of (2).
Such applications should allow interrupt explicitly.

#12 - 05/20/2021 08:56 AM - Eregon (Benoit Daloze)

I think (2) is rare, and IMHO such programs should be fixed.

What would happen is then those programs can't be interrupted (only by killing the process), but then that also becomes a clear motivation to fix those programs so they can be interrupted safely.

Not sure about (1), I think it's loading the current thread + saving current value of the mask, writing the interrupt mask for every ensure, and restoring it once the ensure finishes.

It needs to be done on the non-exceptional path too, so indeed it could be some overhead.

I think we need to measure it in practice here to evaluate better the overhead.

The whole strategy here relies on code in ensure having code that only takes a small amount of time.

If that doesn't hold, it will either hang (motivating those programs to be fixed),

or we could resort to the old semantics after some extra time (e.g., 10s), similar to what <https://bugs.ruby-lang.org/issues/17849#note-3> proposes, and then print a big warning that the program misbehaved (and that state might be corrupted due to that as ensure likely has been interrupted).

That could be useful notably for Ctrl+C to still work as it does currently in such situations.

#13 - 05/21/2021 06:11 AM - matz (Yukihiro Matsumoto)

I doubt if there's a clear solution.

- Timeout uses asynchronous exceptions and it is difficult to handle correctly.
- for example, if some calls Timeout.timeout from within ensure clause directly/indirectly
- or what if timeout interrupt breaks in finalizers

The perfect global timeout (that relies on asynchronous exceptions) might be theoretically possible, but practically (nearly) impossible, I believe.

Matz.

#14 - 05/21/2021 06:19 PM - mame (Yusuke Endoh)

Just FYI. According to [@ko1 \(Koichi Sasada\)](#), masking all exceptions in an ensure clause is possible in pure Ruby, as long as you use MRI.

```
begin
  ...
ensure
  Thread.handle_interrupt(Object => :never) do
    ..
  end
end
```

No interrupt checking is executed outside of handle_interrupt block. This is an implementation detail of MRI, though.

Note: we don't insist that people should write such messy code for all ensure clauses. It may be just useful to perform a proof-of-concept of this proposal, whether it is practically possible to implement a global timeout.

#15 - 05/21/2021 06:53 PM - schneems (Richard Schneeman)

- (2) Some applications can run their program body in ensure clauses.

[@ko1 \(Koichi Sasada\)](#) first, thank you for looking at this issue. I agree this is a problem, it is the pathological case to what I was describing by the inability to detect a halting condition. This case is the cause of my suggestion to add some sort of "overtime" timeout value. Somewhat like how you should send a process SIGTERM and let it gracefully exit before having the option of sending a SIGKILL. What do you think of that option?

[@mame \(Yusuke Endoh\)](#) "It may be possible" thank you for this idea. If exploration or PoC is possible to help move things forward that would be great. I can benchmark some real-world rails apps using https://github.com/schneems/derailed_benchmarks if we are worried about performance concerns. (Though if I am not responsive, it's because ruby-lang emails stopped coming for me. I have asked HSBT but I have no solution now)

I think that this ability to timeout arbitrary Ruby code is very useful, even if it is not perfect. I agree with [@matz \(Yukihiro Matsumoto\)](#) that such a truly "perfect" API and implementation may be impossible...I also believe that this feature is worth improving even if the final result has shortcomings.

Thank you all for your continued work with Ruby!

#16 - 05/22/2021 07:55 PM - Eregon (Benoit Daloze)

name (Yusuke Endoh) wrote in [#note-14](#):

Just FYI. According to [@ko1 \(Koichi Sasada\)](#), masking all exceptions in an ensure clause is possible in pure Ruby, as long as you use MRI.

It does not seem to be the case: <https://gist.github.com/eregon/9022f5709fa054fc4e488d7de085b254>

And I would be surprised if it was on any Ruby, there are two calls there: `handle_interrupt` on `Thread`, and `hash` on `Object`.

Actually such a pattern is a typical pitfall of using `handle_interrupt`, and `monitor.rb` had such a bug.

The correct pattern is the first one documented here: https://www.rubydoc.info/stdlib/core/Thread.handle_interrupt

Of course that's very verbose and unrealistic to be used for every ensure. And additionally it's quite a large overhead when done that way (extra calls, blocks, Hash instances, etc).

In other words, we need VM changes to experiment with this.

I think masking/disabling interrupts in ensure and in finalizers (<https://bugs.ruby-lang.org/issues/13876#change-91365>) would be a good improvement, and definitely worth experimenting with.

That won't make `Thread#raise` completely safe indeed, but much more usable, and much safer.

Of course, we will need to evaluate the performance and semantics impact of this change.

#17 - 05/22/2021 08:00 PM - Eregon (Benoit Daloze)

- Related to Bug #13876: *Tempfile's finalizer can be interrupted by a Timeout exception which can cause the process to hang added*

#18 - 10/07/2021 12:56 PM - headius (Charles Nutter)

name (Yusuke Endoh) wrote in [#note-14](#):

Just FYI. According to [@ko1 \(Koichi Sasada\)](#), masking all exceptions in an ensure clause is possible in pure Ruby, as long as you use MRI.

```
begin
  ...
ensure
  Thread.handle_interrupt(Object => :never) do
    ..
  end
end
```

This is incorrect code. The `handle_interrupt` code must go outside of the entire `begin/ensure/end` block or you still run the risk of being interrupted before the ensure runs.

`handle_interrupt` is a band-aid over the real problem, which is that code can be interrupted at any time. Unfortunately the Ruby Way to fix this is that users have to **opt in** to get safe ensures, and almost nobody gets it right. Every library in existence is broken.

And for the record, both JRuby and TruffleRuby fully support `handle_interrupt`, so you don't have to use MRI.

```
Thread.handle_interrupt(Object => :never) do
  begin
    Thread.handle_interrupt(Object => :immediate) do
      ...
    end
  ensure
    ...
  end
end
```

See <http://blog.headius.com/2008/02/ruby-threadraise-threadkill-timeoutrb.html>

`handle_interrupt` makes it possible to avoid some of these problems, but nobody is using it and the ones who are using it are using it wrong.

#19 - 10/07/2021 08:04 PM - ioquatix (Samuel Williams)

The fiber scheduler redefines `Timeout` to only well defined wait points.

We could also do the same for other options likes `Thread#raise` etc.

#20 - 11/06/2021 12:03 PM - Eregon (Benoit Daloze)

Another way would be to default to `Thread.handle_interrupt(Exception => :on_blocking)` do for all threads, which is more or less what happens with the Fiber scheduler.

Then `Thread#raise` would only affect blocking calls.

OTOH it means we wouldn't be able to interrupt code which doesn't do blocking calls but e.g. loops infinitely and does a very long computation (unless it explicitly checks for interrupt but basically no code does that currently).

Regexp matching is not considered a blocking call/operation currently, but I think we could maybe change that.

#21 - 11/08/2021 08:58 AM - mame (Yusuke Endoh)

[@headius \(Charles Nutter\)](#)

headius (Charles Nutter) wrote in [#note-18](#):

This is incorrect code.

I believe that my code is correct under the current implementation of MRI. You may think that the method invocation itself `Thread.handle_interrupt(...)` do ... end could be interrupted, but it is not true. There is no point of interruption checks during this method invocation, under the current implementation of MRI. This is very subtle implementation detail of MRI, but it is as I have already stated in [#note-14](#).

Here is a test:

```
def foo
  begin
    sleep
  ensure
    Thread.handle_interrupt(Object => :never) do
      $finalized = true
    end
  end
end
```

```
Thread.report_on_exception = false
1000.times do
  $finalized = false
  th = Thread.new { foo }
  sleep 0.1
  3000.times { th.raise }
  begin
    th.join
  rescue
    end
  raise "test failed" unless $finalized
end
```

#22 - 11/08/2021 05:33 PM - Eregon (Benoit Daloze)

[@mame \(Yusuke Endoh\)](#) How do you explain the test I linked above fails (still does with latest CRuby) then?
<https://bugs.ruby-lang.org/issues/17849#note-16>

Anyway, we all agree nobody should rely on that, and it's also unrealistic to ask people to explicitly add `Thread.handle_interrupt` in every `ensure`.

#23 - 11/09/2021 02:09 AM - mame (Yusuke Endoh)

Eregon (Benoit Daloze) wrote in [#note-22](#):

[@mame \(Yusuke Endoh\)](#) How do you explain the test I linked above fails (still does with latest CRuby) then?

The exception is raised immediately after the block returns, and before `handle_interrupt` returns.

```
$ready = false
thread = Thread.new do
  loop do
    $ready = true
    begin
      Thread.pass
    rescue => e
      p e
    ensure
      Thread.handle_interrupt(Object => :never) do # backtrace reports it happens here
        begin
          Thread.pass
        rescue => e
          abort "ensure saw #{e}"
        end
        $finalized = true
        # the exception is raised immediately after this block returns to handle_interrupt
      end
    end
  end
end
```

```

end
end

$finalized = false
1000.times {
  Thread.pass until $ready
  thread.raise "timeout"
  Thread.pass
}
p $finalized #=> true

```

#24 - 04/10/2022 06:26 AM - avit (Andrew Vit)

The handling of timeouts is [documented](#), but that only shows it with a `handle_interrupt` block set up around everything, not under `ensure`. In the examples shown here, can `ensure;begin` or `ensure;Thread.handle_interrupt` be considered "atomic", or is it possible for an interrupt to happen between these lines?

(Another rubyist once recommended a [different approach using rescue/retry](#), but in my testing it looks like `handle_interrupt` is more robust.)

I have a small thought on a possible API: if it might be an improvement to pass interrupt handling options to `ensure`, similar to `rescue`:

```

begin
ensure Timeout::Error => :never
  $finalized = true
end

```

To extend the similarity with `rescue`, could it make sense to allow multiple `ensure` blocks, if different options are needed in sequence?

EDIT: It looks like my idea was previously suggested:

<https://bugs.ruby-lang.org/issues/6762#note-3>

#25 - 04/03/2024 03:50 AM - hsbt (Hiroshi SHIBATA)

- Status changed from Open to Assigned