Introduction to general `IO#timeout` and `IO#timeout=` for blocking operations.

03/14/2022 02:43 AM - ioquatix (Samuel Williams)

**Status:** Open

**Priority:** Normal

**Assignee:**

**Target version:**

**Description**

I would like us to consider introducing a general IO timeout for all (non-)blocking operations, specified per-IO instance. It's useful for ensuring programs don't stop responding or spend an unreasonable amount of time waiting for IO operations.

There are effectively two kinds of interfaces that we need to address:

- Those that already have a timeout argument (e.g. wait_readable) and we follow the existing semantics.
- Those that don't have a timeout argument or timeout semantics (e.g. puts, gets), and thus probably need to raise an exception on timeout.

We have three possible kinds of exceptions we could raise:

- `Errno::ETIMEDOUT`
- `Timeout::Error` (from timeout.rb)
- Introduce `IO::Timeout` or something similar.

Timeout isn't necessarily an error condition. There are different arguments for whether we should define:

```ruby
class IO::Timeout < Exception
end
```

# or

```ruby
class IO::Timeout < StandardError
end
```

I believe the latter (StandardError) is more practical but I'm open to either option. I might have more specific arguments later why one is better than the other after testing in a practical system.

There is already a PR to try it out: [https://github.com/ruby/ruby/pull/5653](https://github.com/ruby/ruby/pull/5653)

**History**

#1 - 03/14/2022 02:43 AM - ioquatix (Samuel Williams)
- Description updated

#2 - 03/14/2022 02:44 AM - ioquatix (Samuel Williams)
- Description updated

#3 - 03/14/2022 04:28 AM - matz (Yukihiro Matsumoto)
I am positive about introducing safer timeout feature. But I have several concerns:

- time-out may happen from I/O blocking operations or other CPU bound operation. This proposal only covers I/O blocking. Is it OK?
- time-out may not be caused by a single I/O operation, but by multiple operations. This proposal should be updated to address it (maybe specifying time-bound by the specific time point).
- some I/O operation methods takes timeout keyword argument. We need to confirm they are consistent with this proposal.

Matz.

#4 - 03/14/2022 09:05 AM - Eregon (Benoit Daloze)
I'm not sure a timeout per IO instance makes sense, some IO operations might take longer e.g. reading many bytes at once and so it seems unclear whether any timeout value would be sensible there.
The proposal should also mention this can only work for non-blocking IOs (and maybe raise if called on a blocking IO?)

I thought Timeout.timeout is already good enough when there is a scheduler and has similar semantics, why would we need this? Is there a concrete example?

#5 - 03/14/2022 02:52 PM - ioquatix (Samuel Williams)

I'm not sure a timeout per IO instance makes sense, some IO operations might take longer e.g. reading many bytes at once and so it seems unclear whether any timeout value would be sensible there.

I think it's reasonable that no single IO operation should take more than, say, several seconds on a healthy system. However it's definitely a hard problem. I'm also considering whether we can have a general default, e.g. set by an environment variable or global within Ruby, e.g. IO.timeout. We could also consider adding keyword arguments to File.open and so on. A timeout is really a way of protecting code from hanging indefinitely, e.g. because of deadlocks or DoS hacks. A lot of programs monkey patch such functionality but none of it is compatible with each other. I think introducing a simple, standard interface here makes sense.

The proposal should also mention this can only work for non-blocking IOs (and maybe raise if called on a blocking IO?)

This only matters for general blocking Ruby. When using the fiber scheduler with io_uring, this limitation goes away.

We could fall back to the blocking Timeout.timeout semantics in non-scheduler blocking case, I just don't know if it's a good idea to over-complexify the implementation. The vast majority of IO in Ruby now is non-blocking by default. This mostly just applies to stdin, stdout, and stderr. I think we could even consider making stdin non-blocking by default.

In order to state this more clearly, we could document this limitation as "Timeouts are best effort and are not always guaranteed to be enforced or accurate," which is totally reasonable in my mind given the nature of timeouts.

I thought Timeout.timeout is already good enough when there is a scheduler and has similar semantics, why would we need this? Is there a concrete example?

Timeout.timeout is hard to implement and I'm not sure there is any general easy implementation. There is a fiber scheduler hook for Timeout.timeout which can be a little bit safer in practice at the expense of only interrupting non-blocking IO.

IO#timeout is more like the default timeout to use when internal wait mechanisms are invoked, like nogvl_wait_for or rb_io_wait. This is much more predictable and robust.

#6 - 03/17/2022 02:08 AM - ioquatix (Samuel Williams)

@matz (Yukihiro Matsumoto) thanks for your comments.

time-out may happen from I/O blocking operations or other CPU bound operation. This proposal only covers I/O blocking. Is it OK?

I believe it's acceptable, since it only impacts IO operations.

time-out may not be caused by a single I/O operation, but by multiple operations. This proposal should be updated to address it (maybe specifying time-bound by the specific time point).

We still retain Timeout.timeout which is sufficient for these cases.

some I/O operation methods takes timeout keyword argument. We need to confirm they are consistent with this proposal.

I agree to introduce consistency here where possible. Some method like gets and puts would be hard to update, but others like read and write should be possible to add individual timeouts.

Later we could consider adding IO#read_timeout and IO#write_timeout if there is a valid use case.

#7 - 03/17/2022 12:06 PM - Eregon (Benoit Daloze)

This part wasn't answered:

Is there a concrete/practical example (for this new timeout)?

I'm not sure it's valuable to have per-IO timeout. It's kind of global state, especially if it's set on some globally-shared IO like STDIN/STDERR/STDOUT. Also this approach can't work when creating an IO, e.g. creating a new TCPSocket connecting to some address+port.
I think this might be redundant and add additional complexity and global state compared to a new way to do timeout that only interrupts IO and blocking operations (like Queue#pop, Mutex#lock, sleep, etc).
i.e., same as Timeout.timeout except it would only interrupt blocking operations (blocking IO is considered a blocking operation too), not Thread#raise on some random Ruby line.
So we'd have Timeout.blocking_timeout(5) {...} or so, and that's no global state, more general and more useful, isn't it?
And that should of course interrupt blocking IO via SIGVTALRM (and interruptible blocking regions in C exts) as already done for other functionality like Timeout.timeout.

#8 - 03/17/2022 03:42 PM - iequatix (Samuel Williams)

I'm not sure it's valuable to have per-IO timeout.
async-io has used it for years successfully as a protection against slowloris attacks. It ensures that no matter who calls the IO operations, some timeout is associated with it.
There is a difference between IO which is usually externally facing and things like Queue, Thread which are internal. I personally have no problem adding timeouts to all those interfaces, and even adding Thread::Queue#timeout if that makes sense.

However, this PR is mostly just addressing the issue of making non-blocking IO safer in the presence of external malicious actors.

#9 - 03/21/2022 03:52 PM - byroot (Jean Boussier)

I'm not sure a timeout per IO instance makes sense, I think it does for various network clients. See for example net-protocol's BufferedIO class which backs Net::HTTP and others.
If you want to have a timeout on your network client, you are basically to read_nonblock / write_nonblock, so for most line based protocols (such as HTTP but also memcached, redis, etc) you can't use IO#gets. This means you need to buffer your reads to find the newline, so now you have the Ruby internal 16KiB IO buffer, plus the BufferedIO 16KiB buffer, and you end up calling String#slice! repeatedly on that Ruby String which is atrocious for performance.
I recently had to do the same thing for a new Redis client I'm working on, but to save on performance I don't slice! the buffer, but keep a numeric @offset alongside.
If I understand this proposal correctly, it would allow me to directly call IO#gets with the timeout I desire, which would allow me to not have my own buffer and solely rely on the internal one.
That being said, if there was some kind of IO.timeout(timeout) {} it would work too. I guess both have their pros and cons.

#10 - 03/21/2022 04:59 PM - Eregon (Benoit Daloze)

I'm still confused, in which cases does this timeout apply?
Only non-blocking IO, right?
The issue title says:
Introduce general IO#timeout and IO#timeout= for all (non-)blocking operations.
The description says:
I would like us to consider introducing a general timeout for all blocking operations.
And the PR says:
https://github.com/ruby/ruby/pull/5653/files#diff-92194f057884b3287a3a6bf84e6e9b2b433a556b68562799252a091744e7854R857

- Set the internal timeout to the specified duration or nil. The timeout
- applies to all non-blocking operations unless otherwise specified.
- This affects the following methods (but is not limited to): #gets, #puts,
  #read, #write, #wait_readable and #wait_writable. This also affects
  non-blocking socket operations like Socket#accept and Socket#connect.

But those are not "non-blocking methods" or "non-blocking operations", or at least that's not intuitive.
Is the actual condition depending on IO#nonblock?, if true possible to timeout all IO operations on it, if false none of them?
I guess one way to express it is this new IO#timeout would apply to: blocking IO methods on non-blocking IO instances.

Sorry if the description is confusing. This is a general timeout for all non-blocking IO operations, specified per IO instance.


However, how should we handle Errno::ETIMEDOUT which can also occur in some system calls.

Should we map all Errno::ETIMEDOUT -> IO::Timeout?

The current implementation just uses Errno::ETIMEDOUT which is the smallest most compatible change (since technically people should already be expecting IO operations to potentially return ETIMEDOUT), e.g. from recv man page:

```
ETIMEDOUT
The connection timed out during connection establishment, or due to a transmission timeout on active connection.
```

A quick glance at Node.JS seems to be they just use ETIMEDOUT.

For me, the biggest reason to use a different class would be because we want a more consistent handling of timeouts, across all blocking operations (not just IO ones). It sounds like maybe that ship has sailed because we now have several different exceptions for timeouts, and no common root AFAIK.

Maybe we should have:

```ruby
class TimeoutError < StandardError # or Exception?
end

class Regexp::TimeoutError < ::TimeoutError
end

class IO::TimeoutError :: TimeoutError
end
etc.
```

I hesitate to make this < StandardError as I feel that timeouts are often not strictly "errors" although a timeout can cause an error.

Basically all exception classes end with Exception or Error or some other indication of it (see ri Exception).

So I think IO::Timeout is not a good fit, because from the name it's unclear it's an exception class.
And the other two timeout exception classes have Timeout + Error in their full name, so IO::TimeoutError seems an obvious fit.

ETIMEDOUT is not e.g. in read/select/epoll_wait/... man pages as far as I can see locally and I don't even see it in the my local recv man page. It seems specific to socket connections, but the timeout we are talking about is more general (e.g., applies to pipes).
I think ETIMEDOUT is a bad fit, because it is not what is actually returned by the syscalls to which we apply the timeout, isn't it?
Also I'd consider Errno::E* to be low-level exceptions, and not something we'd want for higher-level functionality.

For me, the biggest reason to use a different class would be because we want a more consistent handling of timeouts, across all blocking operations (not just IO ones).
If that is useful (so far it seems not to me), that could always be done by a module included by all timeout exception classes.

I hesitate to make this < StandardError as I feel that timeouts are often not strictly "errors" although a timeout can cause an error.

StandardError is not about being "an error" or not (that's just consistent naming for exception classes), it's whether it's something that is sensible to rescue "as a general failure" / to recover from or not.

It is sensible to rescue IO::TimeoutError as a general "something failed during the operation" (at least in some cases) and keep going, while it is e.g. never (or almost never) sensible to rescue (without reraising them) NoMemoryError/SystemStackError.

Maybe we should have: class TimeoutError

That's a deprecated alias for Timeout::Error:

```ruby
$ ruby -w -rtimeout -e 'p TimeoutError'
e:e:1: warning: constant ::TimeoutError is deprecated
```

> #17 - 03/31/2022 11:25 AM - ioquatix (Samuel Williams)

I am fine with IO::TimeoutError but again, I'm not sure it's strictly an error. Not all exceptions or errors have those naming conventions, Errno::E* being an obvious one, but there are several others:

```ruby
> Exception.subclasses
=> [IRB::Abort,
ErrorHighlight::Spotter::NonAscii,
SystemStackError,
NoMemoryError,
SecurityError,
ScriptError,
StandardError,
SignalException,
fatal,
SystemExit]

> SignalException.subclasses
=> [Interrupt]

> StandardError.subclasses.map(&:to_s).grep_v(/Error/)
=> ["IRB::IllegalRCGenerator",
"IRB::UndefinedPromptMode",
"IRB::CantChangeBinding",
"IRB::CantShiftToMultlIrbMode",
"IRB::NoSuchJob",
"IRB::IrbSwitchedToCurrentThread",
"IRB::IrbAlreadyDead",
"IRB::IllegalParameter",
"IRB::CantReturnToNormalMode",
"IRB::UnrecognizedSwitch",
"RubyLex::TerminateLineInput",
"Gem::TSort::Cyclic"]
```

Basically, I think there are two questions:

1. Should it be called IO::Timeout or IO::TimeoutError.
2. Should it be < Exception or < StandardError.

I don't have strong opinion about either.

For (1) I guess it's matter of taste and opinion whether a timeout is an error or not. To me, whether it's an error depends on context. But for most cases, it could be considered exceptional except when a timeout is a reasonable possibility, read with timeout is exceptional, but wait_readable with timeout is not, since it has clear existing semantics for dealing with "exceeding the given timeout". So, with that in mind, it probably makes sense that in most cases, the timeout would be unexpected and thus an error.

For (2) I've gone both ways on this in the past and finally settled on < StandardError being more reasonable. However it's true that in many cases, a timeout is significantly different from a normal "error". It's the reason why SystemExit is an exception and not an error, is the same motivating factor for why timeouts should be an exception and not an error.

I would personally like to hear if @matz (Yukihiro Matsumoto) has an opinion on this. Basically, I think we agree on IO::TimeoutError < StandardError being the most obvious, but I don't have a strong opinion on this... the closest analogue would be Interrupt which is also not StandardError which makes me think we need to consider this more carefully.
By the way, this still doesn't address that some operations can return Errno::ETIMEDOUT and this class of errors is practically speaking the same.

I don't have a strong opinion about whether we should force users to write rescue IO::TimeoutError, Errno::ETIMEDOUT but it seems more difficult for users.

Regarding which operations can give Errno::ETIMEDOUT you can check your local man pages: man -K ETIMEDOUT. Surprisingly macOS has a ton of hits, even some non-socket cases. I can check Linux too.

I guess we need to decide whether we try to incorporate Errno::ETIMEDOUT or ignore it. Or maybe another option is to just have a shared module which is included in all of them.

AFAIK Errno::* is only meant for what the libc/syscalls would return, nothing else, so doesn't fit here. As said above, many of functions involved above don't ever return ETIMEDOUT, so it would be incorrect/surprising to reuse it for that.

Re naming Errno::E*, the "Error" is in the name, twice, it's "Error numero::ERROR abbreviation" ;)

Most of the other exceptions have something that is closely related to error/exception in their name.

It's not because there is "Error" in the name that the exception class should be treated "as an error". There is simply no such relation, and as you can see there are exception classes with Error in the name both under StandardError and above it too. They are all exceptions, and whether it is "an error" almost always depends on the context. The only meaningful distinction AFAIK in Ruby is < StandardError or not. And since it's meant to be rescued in some cases, and for consistency will all other timeout exceptions, it should be < StandardError.

The reality is, some functions do return this and so we should consider how it impacts users who want to handle all IO timeouts. Having two distinct exceptions seems a little messy to me. I'm happy to accept these are different, but I'm also suggesting we should consider carefully the implications of this choice.

Basically, do we want users to write:

rescue IO::TimeoutError, Errno::ETIMEDOUT

In relation to #18669, this proposal needs IO#nonblock= and IO#nonblock? because IO#timeout= only applies to IO#nonblock?=true IO instances. And so to use it successfully, a user needs to ensure the relevant IO is in nonblock=true mode + some other conditions:

https://github.com/ruby/ruby/pull/5653/files#diff-92194f057884b3287a3a6b84e6a3b2bf433a556b68562799252a091744e7854R856-R867 says:

* call-seq:
* timeout = duration -> duration
* timeout = nil -> nil
* 
* Set the internal timeout to the specified duration or nil. The timeout
* applies to all blocking operations provided the IO is in non-blocking mode:
* +io.nonblock? => true+.
* 
* This affects the following methods (but is not limited to): #gets, #puts, #read, #write, #wait_readable and #wait_writable. This also affects
* blocking socket operations like Socket#accept and Socket#connect.

In #20 - 03/31/2022 10:52 PM - ioquatix (Samuel Williams) many of functions involved above don't ever return ETIMEDOUT, so it would be incorrect/surprising to reuse it for that.

In #21 - 04/21/2022 09:20 AM - Eregon (Benoit Daloze) it's like cases of rescue IO::WaitReadable, ... which often includes multiple exception types. It can be useful to distinguish both timeouts, IO::TimeoutError is the per-IO timeout and Errno::ETIMEDOUT is a per-operation timeout (only for some operations, most operations can't raise ETIMEDOUT) and the timeout durations can be different of course.
I think we can make this for blocking operations too, but the code path will be a little different if a timeout is set.

Okay, I was able to make it work:

```ruby
STDIN.timeout = 1

pp STDIN.read
```

Output:

```
/Users/samuel/Projects/ioquatix/ruby/build/../test.rb:3:in `read': Operation timed out @ io_fread - <STDIN> (Errno::ETIMEDOUT)
  from /Users/samuel/Projects/ioquatix/ruby/build/../test.rb:4:in `<main>,'
```

The final step is to decide what kind of exception we should use.