

Ruby - Feature #6762

Control interrupt timing

07/21/2012 04:11 AM - ko1 (Koichi Sasada)

Status:	Closed	
Priority:	Normal	
Assignee:	ko1 (Koichi Sasada)	
Target version:	2.0.0	

Description

=begin
= Abstract

Add asynchronous interrupt timing control feature. Control with the following three modes:

- immediate: process interrupt immediately
- never: never process interrupt
- on_blocking: delay interrupt until blocking operation

example

```
th = Thread.new do
  Thread.control_interrupt(RuntimeError => :never) do
    # in this block, thrown RuntimeError doesn't occur
  end
  ... # raise thrown RuntimeError
end
...
th.raise "foo"
```

= Background

== Terminology

- Interrupt: asynchronous interrupt and corresponding procedures
 - Thread#raise and occurring exception
 - signal and corresponding trap
 - Thread#kill and thread termination
 - Main thread termination and thread termination
(after main thread termination, all threads exit themselves)
- Interrupt checking: check interrupt
- Blocking operation: Possible to block the current thread such as IO read/write. In CRuby implementation, it is nearly equals to tasks without GVL

== Current use-cases of Interrupt

There are several `Interrupt` in Ruby.

Example 1

```
th = Thread.new{
  begin
  ...
  rescue FooError
  ...
  end
}
th.raise(FooError) #=> Raise FooError on thread `th`
```

Example 2

```
q = Queue.new
th1 = Thread.new{
  q << calc_in_algorithm1
}
th2 = Thread.new{
  q << calc_in_algorithm2
}
result = q.pop
th1.raise(TerminateCalcError)
th2.raise(TerminateCalcError)
```

Run two algorithms simultaneously.

If we get an answer from one algorithm,

kill them with TerminateCalcError

In this case, it is also okay with Thread#kill

Example 3

```
trap(SIGINT){
  # do something
  # maybe termination process
}
trap(SIGHUP){
  # do something
  # maybe reloading configuration process
}
server_exec # server main process
```

In such interrupts are checked at several points such as:

- method invocation timing
- method returning timing
- move program counter
- before and after block operation

== Problem

Interrupt causes the following problems because we can't control occurring timing.

- Un-safe ensure clause: Generally, ensure clause should not interrupt because it contains important tasks such as freeing resources.
- Un-safe resource allocation: If interrupt occurs between resource allocation and assign it to the variable, we can't free this object (however, this problem not too big because we have a gc and appropriate finalizer can free it).
- (other problems? please complement me)

I show an example below.

Example 4

this method is similar implementation of timeout()

```
def timeout(sec)
  timer_thread = Thread.new(Thread.current){|parent|
    sleep(sec)
    parent.raise(TimeoutError)
  }
end
```

```

begin
yield
ensure
timer_thread.stop # close thread
end
end
timeout(3){
begin
f = # point (a)
open(...) # of course, there are no problem with open(...){f| ...}
# but it is an example to show the problem
...
ensure
... # point (b)
f.close
end
}

```

On example 4, there are two problems.

Point (b) is easy to understand. If interrupt was thrown at point (b), then `f.close()` isn't called. It is problem.

On the point (a), it is a position between resource allocation (open()) and assignment `f = '. It is very rare, but it is possible. If we get interrupt before assignment, then we can't free resources (can't call f.close()) in ensure clause. It is also problem.

The problem is we can't control interrupt timing.

= Proposal

Adding interrupt timing control feature to Thread. Introduce two methods to Thread class.

- Thread.control_interrupt
- Thread.check_interrupt

Rdoc documents are:

Thread.control_interrupt():

call-seq:

Thread.control_interrupt(hash) { ... } -> result of the block

Thread.control_interrupt controls interrupt timing.

interrupt means asynchronous event and corresponding procedure by Thread#raise, Thread#kill, signal trap (not supported yet) and main thread termination (if main thread terminates, then all other thread will be killed).

hash has pairs of ExceptionClass and TimingSymbol. TimingSymbol is one of them:

- :immediate Invoke interrupt immediately.
- :on_blocking Invoke interrupt while *BlockingOperation*.
- :never Never invoke interrupt.

BlockingOperation means that the operation will block the calling thread, such as read and write. On CRuby implementation, *BlockingOperation* is operation executed without GVL.

Masked interrupts are delayed until they are enabled.
This method is similar to sigprocmask(3).

TODO (DOC): control_interrupt is stacked.

TODO (DOC): check_ancestors.

TODO (DOC): to prevent all interrupt, {Object => :never} works.

NOTE: Asynchronous interrupts are difficult to use.

If you need to communicate between threads,
please consider to use another way such as Queue.
Or use them with deep understanding about this method.

```
# example: Guard from Thread#raise
th = Thread.new do
  Thread.control_interrupt(RuntimeError => :never) {
    begin
      # Thread#raise doesn't interrupt here.
      # You can write resource allocation code safely.
      Thread.control_interrupt(RuntimeError => :immediate) {
        # ...
        # It is possible to be interrupted by Thread#raise.
      }
    end
    ensure
      # Thread#raise doesn't interrupt here.
      # You can write resource deallocation code safely.
    end
  }
end
Thread.pass
# ...
th.raise "stop"
```

```
# example: Guard from TimeoutError
require 'timeout'
Thread.control_interrupt(TimeoutError => :never) {
  timeout(10){
    # TimeoutError doesn't occur here
    Thread.control_interrupt(TimeoutError => :on_blocking) {
      # possible to be killed by TimeoutError
      # while blocking operation
    }
    # TimeoutError doesn't occur here
  }
}
```

```
# example: Stack control settings
Thread.control_interrupt(FooError => :never) {
  Thread.control_interrupt(BarError => :never) {
    # FooError and BarError are prohibited.
  }
}
```

```
# example: check ancestors
Thread.control_interrupt(Exception => :never) {
  # all exceptions inherited from Exception are prohibited.
}
```

Thread.check_interrupt():

call-seq:

Thread.check_interrupt() -> nil

Check queued interrupts.

If there are queued interrupts, process respective procedures.

This method can be defined as the following Ruby code:

```
def Thread.check_interrupt
  Thread.control_interrupt(Object => :immediate) {
    Thread.pass
  }
end
```

Examples:

```
th = Thread.new{
  Thread.control_interrupt(RuntimeError => :on_blocking){
    while true
      ...
      # reach safe point to invoke interrupt
      Thread.check_interrupt
      ...
    end
  }
}
...
th.raise # stop thread
```

NOTE: This example can be described by the another code.
You need to keep to avoid asynchronous interrupts.

```
flag = true
th = Thread.new{
  Thread.control_interrupt(RuntimeError => :on_blocking){
    while true
      ...
      # reach safe point to invoke interrupt
      break if flag == false
      ...
    end
  }
}
...
flag = false # stop thread
```

I have already commit-ed these methods into trunk.
Please try it and discuss.

This commit is easy to revert :)

Naming is also problem as usual. Good naming is also welcome.

= Acknowledgment

The base of this proposal is a discussion[1].

[1] Akira Tanaka "Re: Thread#raise, Thread#kill, and timeout.rb are unsafe" ruty-talk (2008.3) <http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/294917>

Many dev-people help me to make up this proposal.

=end

Related issues:

Related to Ruby - Bug #6174: Fix collision of ConditionVariable#wait timeout ...	Rejected	03/19/2012
Related to Ruby - Feature #7505: Mutex#owned? [] [] [] [] [] []	Closed	12/04/2012
Has duplicate Ruby - Bug #4285: Ruby don't have asynchronous exception safe ...	Closed	01/17/2011

Associated revisions

Revision 719e0cd530dd7187c716c103876fb3a112cbd30c - 11/30/2012 09:28 AM - ko1 (Koichi Sasada)

- thread.c: rename Thread.control_interrupt to Thread.async_interrupt_timing.
The option name :never' is also changed to :defer'.
[ruby-core:50375] [ruby-trunk - Feature #6762]
- thread.c: remove Thread.check_interrupt.
This method is difficult to understand by name.
- thraed.c: add Thread.async_interrupted?.
This method check any deferred async interrupts.
- test/ruby/test_thread.rb: change tests for above.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@38046 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 719e0cd5 - 11/30/2012 09:28 AM - ko1 (Koichi Sasada)

- thread.c: rename Thread.control_interrupt to Thread.async_interrupt_timing.
The option name :never' is also changed to :defer'.
[ruby-core:50375] [ruby-trunk - Feature #6762]
- thread.c: remove Thread.check_interrupt.
This method is difficult to understand by name.
- thread.c: add Thread.async_interrupted?.
This method check any deferred async interrupts.
- test/ruby/test_thread.rb: change tests for above.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@38046 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 0f9b33c793f225c1b817d73e5c915050c429edc4 - 12/23/2012 10:18 AM - ko1 (Koichi Sasada)

- thread.c: rename methods:
from Thread.async_interrupt_timing to Thread.handle_interrupt,
from Thread.async_interrupted? to Thread.pending_interrupt?.
Also rename option from defer' to never'.
[ruby-core:51074] [ruby-trunk - Feature #6762]
- vm_core.c, thread.c: rename functions and data structure
async_errinfo' to pending_interrupt'.
- thread.c: add global variables sym_immediate, sym_on_blocking and
sym_never.
- cont.c, process.c, vm.c, signal.c: ditto.
- lib/sync.rb, lib/thread.rb: catch up this renaming.
- test/ruby/test_thread.rb: ditto.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@38577 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 0f9b33c7 - 12/23/2012 10:18 AM - ko1 (Koichi Sasada)

- thread.c: rename methods:
from Thread.async_interrupt_timing to Thread.handle_interrupt,
from Thread.async_interrupted? to Thread.pending_interrupt?.
Also rename option from defer' to never'.
[ruby-core:51074] [ruby-trunk - Feature #6762]
- vm_core.c, thread.c: rename functions and data structure
async_errinfo' to pending_interrupt'.
- thread.c: add global variables sym_immediate, sym_on_blocking and
sym_never.
- cont.c, process.c, vm.c, signal.c: ditto.
- lib/sync.rb, lib/thread.rb: catch up this renaming.
- test/ruby/test_thread.rb: ditto.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@38577 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

History

#1 - 07/21/2012 04:17 AM - ko1 (Koichi Sasada)

- Description updated

#2 - 07/21/2012 05:23 AM - ko1 (Koichi Sasada)

(2012/07/21 4:48), Eric Wong wrote:

"ko1 (Koichi Sasada)" redmine@ruby-lang.org wrote:

- Un-safe ensure clause: Generally, ensure clause should not interrupt because it contains important tasks such as freeing resources.

Thank you for addressing this issue. ensure clause behaving properly is most important to me.

```
# example: Guard from Thread#raise
th = Thread.new do
  Thread.control_interrupt(RuntimeError => :never) {
    begin
      # Thread#raise doesn't interrupt here.
      # You can write resource allocation code safely.
```

```

Thread.control_interrupt(RuntimeError => :immediate) {
  # ...
  # It is possible to be interrupted by Thread#raise.
}
ensure
  # Thread#raise doesn't interrupt here.
  # You can write resource deallocation code safely.
end
}
end

```

I like the above is now possible and safe, but I think having Thread.control_interrupt twice in above example is repetitive and error-prone.

How about having something like at_exit, but local to the current scope:

I understand what you want.

In my ticket, I proposed two things.

- (1) Introducing the concept to "interrupt control"
- (2) Introducing primitives to achieve (1)

Maybe you proposed

- (3) Extra APIs to use (1) and (2) in easy way

(or (2)?)

I want to make clear and fix the (1) and (2) before (3).
How about it?

How about having something like at_exit, but local to the current scope:

```

def something
  at_scope_exit do
    # Thread#raise doesn't interrupt here.
    # do what you would normally do in ensure clause
    # deallocate resource if allocated
    res.release if res
  end

  # It is possible to be interrupted by Thread#raise.
  # You can write resource allocation code safely because
  # at_scope_exit already registered deallocation code
  res = Foo.acquire
  ...
end

```

at_scope_exit could probably take the same args as Thread.control_interrupt, too:

```
at_scope_exit(TimeoutError => :never) { ... }
```

We need more extra primitive to hook block ending to implement it.
It seems hard task...

A Trivial point. `res` in block is not a variable (it parsed as method) because the assignment of res (res = ...) is placed after the block.

One idea is extending ensure semantics.
I'm not sure how to design it...
We need more ideas.

--
// SASADA Koichi at atdot dot net

#3 - 07/21/2012 07:53 AM - normalperson (Eric Wong)

SASADA Koichi ko1@atdot.net wrote:

I understand what you want.

In my ticket, I proposed two things.

- (1) Introducing the concept to "interrupt control"
- (2) Introducing primitives to achieve (1)

Maybe you proposed

- (3) Extra APIs to use (1) and (2) in easy way

(or (2)?)

I want to make clear and fix the (1) and (2) before (3).
How about it?

I agree, I want (3) :)

I'm not sure if the current primitives make it possible to implement (3)

A Trivial point. `res` in block is not a variable (it parsed as method)
because the assignment of res (res = ...) is placed after the block.

Oops, yes, I often forget to declare variables :x

One idea is extending ensure semantics.
I'm not sure how to design it....
We need more ideas.

What if ensure is made to support parameters?

```
begin
ensure Exception => :never
end
```

#4 - 07/21/2012 03:53 PM - kosaki (Motohiro KOSAKI)

- Thread.control_interrupt
- Thread.check_interrupt

EEK. Please don't use 'interrupt' word. It makes a lot of confusing to
Unix programmer.

Rdoc documents are:

call-seq:
Thread.control_interrupt(hash) { ... } -> result of the block

Thread.control_interrupt controls interrupt timing.

interrupt means asynchronous event and corresponding procedure
by Thread#raise, Thread#kill, signal trap (not supported yet)
and main thread termination (if main thread terminates, then all
other thread will be killed).

No. control_interrupt should NOT inhibit running trap procedure. Because of,
Thread.control_interrupt() is per-thread, but trap is not per-thread.

btw, Probably it should be per-fiber instead of per-thread.

hash has pairs of ExceptionClass and TimingSymbol. TimingSymbol
is one of them:

- :immediate Invoke interrupt immediately.
- :on_blocking Invoke interrupt while *BlockingOperation*.

I don't think 'on_blocking' is good name. Example, pthread cancel have a 'cancellation point' concept and many blocking functions is defined as cancellation point. but a few non blocking functions also defined as cancellation point. So, this on_blocking should have a name as concept likes interruptible point. (but again, i don't like interrupt word)

- :never Never invoke interrupt.

#5 - 07/25/2012 06:23 PM - ko1 (Koichi Sasada)

(2012/07/21 15:41), KOSAKI Motohiro wrote:

- Thread.control_interrupt
- Thread.check_interrupt

EEK. Please don't use 'interrupt' word. It makes a lot of confusing to Unix programmer.

Okay. Give us a good name.

Rdoc documents are:

call-seq:

Thread.control_interrupt(hash) { ... } -> result of the block

Thread.control_interrupt controls interrupt timing.

interrupt means asynchronous event and corresponding procedure by Thread#raise, Thread#kill, signal trap (not supported yet) and main thread termination (if main thread terminates, then all other thread will be killed).

No. control_interrupt should NOT inhibit running trap procedure. Because of, Thread.control_interrupt() is per-thread, but trap is not per-thread.

It is not reason. trap handler and an exception from trap handler interrupt (ah, you don't like this word) ensure clause. It is problem.

Kosaki-san and me talked about it at IRC. Kosaki-san proposed that an exception from trap handler should be caused as `Thread#raise'. It can protect from this feature. It seems good. I think mask signal trap with this feature is more simple.

Any other ideas or comments?

btw, Probably it should be per-fiber instead of per-thread.

Okay.

hash has pairs of ExceptionClass and TimingSymbol. TimingSymbol is one of them:

- :immediate Invoke interrupt immediately.
- :on_blocking Invoke interrupt while *BlockingOperation*.

I don't think 'on_blocking' is good name. Example, pthread cancel have a 'cancellation point' concept and many blocking functions is defined as cancellation point. but a few non blocking functions also defined as cancellation point. So, this on_blocking should have a name as concept likes interruptible point. (but again, i don't like interrupt word)

Akr-san proposed `on_blockable'. Any other ideas?

--

#6 - 07/25/2012 06:53 PM - ko1 (Koichi Sasada)

(2012/07/21 7:51), Eric Wong wrote:

I want to make clear and fix the (1) and (2) before (3).
How about it?

I agree, I want (3) :)
I'm not sure if the current primitives make it possible to implement (3)

A Trivial point. `res` in block is not a variable (it parsed as method)
because the assignment of res (res = ...) is placed after the block.

Oops, yes, I often forget to declare variables :x

One idea is extending ensure semantics.
I'm not sure how to design it....
We need more ideas.

What if ensure is made to support parameters?

```
begin
  ensure Exception => :never
end
```

Introduce new syntax?
It seems difficult to talk matz to introduce it.

But I think it is good syntax (+1).

Implementation note:
Current ensure clause is very light weight.

```
begin
  foo
ensure
  bar
end
```

is same performance as

```
foo
bar
```

if foo doesn't raise any exceptions.

Compiler make duplicated code (bar) like (pseudo-code):

```
begin
  foo
  bar
rescue all exception
  bar
  raise # propagate an exception
end
```

After introducing new syntax, then it will be compiled to:

```
begin
  foo
  control_interrupt(...) {
    bar
  }
rescue
  control_interrupt(...) {
    bar
  }
  raise
end
```

end

--

// SASADA Koichi at atdot dot net

#7 - 10/27/2012 07:26 AM - ko1 (Koichi Sasada)

- Assignee set to ko1 (Koichi Sasada)

It still remains naming consideration.
Comments are welcome.

And we need `signal handling` feature.
Kosaki-san, could we discuss about it at RubyConf next week?
Eric (Wong), will you attend RubyConf?

#8 - 10/27/2012 11:23 AM - normalperson (Eric Wong)

"ko1 (Koichi Sasada)" redmine@ruby-lang.org wrote:

Eric (Wong), will you attend RubyConf?

No.

#9 - 10/27/2012 11:23 AM - ko1 (Koichi Sasada)

(2012/10/27 11:10), Eric Wong wrote:

Eric (Wong), will you attend RubyConf?

No.

:(

--

// SASADA Koichi at atdot dot net

#10 - 11/03/2012 08:33 AM - kosaki (Motohiro KOSAKI)

- Thread.control_interrupt
- Thread.check_interrupt

EEK. Please don't use 'interrupt' word. It makes a lot of confusing to
Unix programmer.

Okay. Give us a good name.

How's this?

control_interrupt() => defer_async_raise() or defer_unwinding()

"control" is unclear and don't explain what action does. Actually this procedure block provide a defer way.
and "interrupt" is often used for other meanings then I think to avoid it is better.

never => end

"never" seems a word to drop an exception. I like to say "defer to end of block"

below is a rough and draft documentation idea.

a exception is queueed and exception raising will be deferred to an end of control_async_raise block.
a queueed exception never be lost.

:on_blocking => raise_point

I prefer to use foo_point rather than "block" because unblocking built-in may allow an exception raise in future.

example, recently we decided to release in zlib even though it doesn't take an IO. I except zlib and other long calculation method also prefer to allow to raise exceptions.

below is a rough and draft documentation idea.

an exception will be only raised on implicit or explicit on raise point.
almost IO and built-in blockable operation provide implicit raise point
and Thread.may_raise() provide explicit one.

Thread.check_interrupt => Thread.may_raise

"check" is also unclear word to me. and It seems to return boolean value of checking result. I think "check" is what does. and people want to know what's happen. then I propose may_raise.

#11 - 11/03/2012 08:36 AM - kosaki (Motohiro KOSAKI)

It still remains naming consideration.
Comments are welcome.

And we need 'signal handling' feature.
Kosaki-san, could we discuss about it at RubyConf next week?

After while thinking, I prefer to suggest this control_interrupt() method doesn't support to mask trap handler.
because of,

- people only want to defer Interrupt exception. trap handler control is close but different topic.
- signal and exception have different semantics. async exception is queued and never be lost. but signal may be lost.

#12 - 11/03/2012 09:07 AM - kosaki (Motohiro KOSAKI)

and maybe undeferred is better than immediate. :)
because 'immediate' can't guarantee an interrupt fire immediately when running heavy weight C code.

#13 - 11/24/2012 12:15 PM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Priority changed from Normal to 5

Ko1 said, this issue is requiring just the name.
Please decide the name and commit it before preview2 (1 Dec.).
Otherwise, I'll postpone this ticket to next minor.

--

Yusuke Endoh mame@tsg.ne.jp

#14 - 11/26/2012 09:53 AM - ko1 (Koichi Sasada)

Okay mame-san.

Eric, could you advice a good name?

Kosaki-san's points:
[\[ruby-core:48769\]](#) [ruby-trunk - Feature [#6762](#)] Control interrupt timing

(2012/11/24 12:15), mame (Yusuke Endoh) wrote:

Issue [#6762](#) has been updated by mame (Yusuke Endoh).

Status changed from Open to Assigned
Priority changed from Normal to High

Ko1 said, this issue is requiring just the name.
Please decide the name and commit it before preview2 (1 Dec.).
Otherwise, I'll postpone this ticket to next minor.

--

// SASADA Koichi at atdot dot net

#15 - 11/26/2012 09:53 AM - ko1 (Koichi Sasada)

(2012/11/03 8:33), kosaki (Motohiro KOSAKI) wrote:

Okay. Give us a good name.

How's this?

`control_interrupt() => defer_async_raise() or defer_unwinding()`

"control" is unclear and don't explain what action does. Actually this procedure block provide a defer way.
and "interrupt" is often used for other meanings then I think to avoid it is better.

'immediate' does not defer. I can't accept the name `defer_*`.

`never => end`

"never" seems a word to drop an exception. I like to say "defer to end of block"

'end' is also unclear. how about 'defer'?

below is a rough and draft documentation idea.

a exception is queueed and exception raising will be deferred to an end of `control_async_raise` block.
a queued exception never be lost.

`:on_blocking => raise_point`

I prefer to use `foo_point` rather than "block" because unblocking built-in may allow an exception raise in future.
example, recently we decided to release in `zlib` even though it doesn't take an IO. I except `zlib` and other long calculation method also prefer to allow to raise exceptions.

below is a rough and draft documentation idea.

an exception will be only raised on implicit or explicit on raise point.
almost IO and built-in blockable operation provide implicit raise point
and `Thread.may_raise()` provide explicit one.

How about 'blockable_point' ?

`Thread.check_interrupt => Thread.may_raise`

"check" is also unclear word to me. and It seems to return boolean value of checking result. I think "check" is what does.
and people want to know what's happen. then I propose `may_raise`.

It inspired from `CHECK_INTS` from C source code. I feel `may_raise` is strange.

--

// SASADA Koichi at atdot dot net

#16 - 11/26/2012 10:29 AM - normalperson (Eric Wong)

SASADA Koichi ko1@atdot.net wrote:

Okay mame-san.

Eric, could you advice a good name?

Really, naming is hard :<

I think `Thread.may_raise` is fine, however...

I prefer not to introduce new methods at all, but instead overload the existing `begin/ensure` syntax.

```
begin async_raise: false
  # code in this section will not allow exceptions raised by other threads
```

```

...
end

begin
  ...
  ensure async_raise: false
  # code in this section will not allow exceptions raised by other threads
  ...
end

```

(similar to: <http://mid.gmane.org/20120720225131.GA15737@dcvr.yhbt.net>)

Can matz provide feedback on this syntax?

I don't know the Ruby parser very well, but I think it is doable without breaking existing code...

Kosaki-san's points:

[\[ruby-core:48769\]](#) [ruby-trunk - Feature [#6762](#)] Control interrupt timing

If we use Thread.may_raise; how about adding "Thread.test_raise" ?
Thread.test_raise would work similar to pthread_testcancel().

I think Thread.test_raise may even just be implemented as:

```

def Thread.test_raise
  prev = Thread.may_raise
  Thread.may_raise = true # will raise here on pending exceptions
  Thread.may_raise = prev
end

```

#17 - 11/26/2012 01:53 PM - kosaki (Motohiro KOSAKI)

For the records.

Ko1 suggested me defer_async_interrupt or async_interrupt_timing(...) {...}.
Both looks acceptable to me.

```

async_interrupt_timing(X => :immediate)
async_interrupt_timing(X => :on_blocking)
async_interrupt_timing(X => :defer) # instead of :never

```

The rest problem is async_interrupt_timing(X => :on_blocking) looks still strange.
because of, bignum don't have blocking point for example.

Now, two idea was raised.

```

23:14 ko1_ndk: async_interrupt_timing(Y => :checkpoint)
23:15 ko1_ndk: async_interrupt_timing(Y => :cancepoint)

```

To me, this is not unacceptable. but not fine too. ;-)
Any good name idea is highly welcome.

Eric:

Your syntax overloading looks ok to me. and I think you misunderstand what ko1's Thread.check_interrupt() does.
Current Thread.check_interrupt() doesn't have an argument nor return value. It just behave as pthread_testcancel(),
(i.e. raise an exception if any exception is queueed).
Moreover, I don't like a name of "test_foobar" because it seems to test something and return boolean value.

#18 - 11/27/2012 12:40 AM - headius (Charles Nutter)

normalperson (Eric Wong) wrote:

Really, naming is hard :<

I think Thread.may_raise is fine, however...

I prefer not to introduce new methods at all, but instead
overload the existing begin/ensure syntax.

```

begin async_raise: false
  # code in this section will not allow exceptions raised by other threads
  ...
end

```

```
begin
  ...
  ensure async_raise: false
  # code in this section will not allow exceptions raised by other threads
  ...
end
```

The examples of Thread.may_raise do not guarantee an ensure block will run, because you still have to get to the may_raise call.

```
begin
raise Something
ensure
```

context switch + async exception could happen here

```
Thread.may_raise = false
...
end
```

It's also going to be error prone because people will forget to turn async exceptions back on again. If you really want to ensure that ensure blocks don't get interrupted asynchronously, just make them that way by default, or provide syntax as Eric suggests.

Honestly, the only safe answer is to disallow asynchronous exceptions. The better long-term design would probably be to provide a built-in messaging/polling mechanism between threads that threads can opt into.

#19 - 11/27/2012 12:47 AM - kosaki (Motohiro KOSAKI)

Honestly, the only safe answer is to disallow asynchronous exceptions. The better long-term design would probably be to provide a built-in messaging/polling mechanism between threads that threads can opt into.

Thread.raise is not big problem, the most big problems are timeout module and Ctrl-C. I don't think we can disallow ctrl-c.

#20 - 11/27/2012 01:50 AM - headius (Charles Nutter)

kosaki (Motohiro KOSAKI) wrote:

Thread.raise is not big problem, the most big problems are timeout module and Ctrl-C. I don't think we can disallow ctrl-c.

The timeout module uses Thread#raise, of course.

#21 - 11/27/2012 02:10 AM - kosaki (Motohiro KOSAKI)

The timeout module uses Thread#raise, of course.

Sure. then we can't drop Thread.raise even though 99% programmers don't use it directly.

#22 - 11/27/2012 03:11 AM - headius (Charles Nutter)

kosaki (Motohiro KOSAKI) wrote:

Sure. then we can't drop Thread.raise even though 99% programmers don't use it directly.

Sure we can...drop timeout as well :)

Of course I accept that Thread#kill and raise aren't going away...I'm just saying that no amount of patching will make them safe.

#23 - 11/30/2012 05:53 PM - ko1 (Koichi Sasada)

(2012/11/26 13:53), kosaki (Motohiro KOSAKI) wrote:

Ko1 suggested me defer_async_interrupt or async_interrupt_timing(...){...}. Both looks acceptable to me.

```
async_interrupt_timing(X => :immediate)
async_interrupt_timing(X => :on_blocking)
async_interrupt_timing(X => :defer) # instead of :never
```

The rest problem is `async_interrupt_timing(X => :on_blocking)` looks still strange.
because of, bignum don't have blocking point for example.

I rename it.

Now, two idea was raised.

```
23:14 ko1_ndk: async_interrupt_timing(Y => :checkpoint)
23:15 ko1_ndk: async_interrupt_timing(Y => :cancelpoint)
```

To me, this is not unacceptable. but not fine too. ;-)
Any good name idea is highly welcome.

Highly welcome.

```
--
// SASADA Koichi at atdot dot net
```

#24 - 11/30/2012 06:28 PM - ko1 (Koichi Sasada)

- Status changed from Assigned to Closed
- % Done changed from 0 to 100

This issue was solved with changeset r38046.
Koichi, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

-
- thread.c: rename `Thread.control_interrupt` to `Thread.async_interrupt_timing`.
The option name `:never` is also changed to `:defer`.
[\[ruby-core:50375\]](#) [\[ruby-trunk - Feature #6762\]](#)
 - thread.c: remove `Thread.check_interrupt`.
This method is difficult to understand by name.
 - thread.c: add `Thread.async_interrupted?`.
This method check any deferred async interrupts.
 - test/ruby/test_thread.rb: change tests for above.

#25 - 11/30/2012 06:29 PM - brent (Brent Roman)

I like this proposal.
It looks very similar to one I made five years ago on this thread:

www.ruby-forum.com/topic/135822

The key, of course, is to associate a (hidden) queue with each thread
for incoming exceptions from other threads.
This essentially makes `Thread#raise` analogous to `Queue#push`

Kosaki,
Was my old post any sort of inspiration for this, or did you
arrive at the same solution independently? You add the ability to
assign different exception delivery policies to each subclass of
Exception.
This seems good on the surface, but won't it complicate the queue
management and make it possible for exceptions to be delivered out of
order? Have you thought about this?

Charlie,
Five years ago, after a bit of arm twisting, you admitted that this
technique could allow `thread.raise` to be used safely. Why the change of
heart now?

```
--
Posted via http://www.ruby-forum.com/.
```


#26 - 11/30/2012 06:31 PM - ko1 (Koichi Sasada)

- Status changed from Closed to Feedback

I renamed it as [\[ruby-core:50147\]](#) says.

and I also remove/add method:

```
* thread.c: remove Thread.check_interrupt.  
This method is difficult to understand by name.
```

```
* thraed.c: add Thread.async_interrupted?.  
This method check any deferred async interrupts.
```

Ah, I want to say "This method checks any deferred async interrupts are there".
It is no ambiguous, I think.

Feedback is welcome.

Thanks,
Koichi

#27 - 11/30/2012 07:23 PM - brent (Brent Roman)

Koichi,

Sorry about misspelling your name in my previous post. :-(

In reference to:

- thraed.c: add Thread.async_interrupted?.
This method check any deferred async interrupts.

Ah, I want to say "This method checks any deferred async interrupts are there".

It is no ambiguous, I think.

I would suggest:

- thread.c: add Thread.exceptions_pending?
This method checks for any deferred exceptions.

"This method returns true if there are exceptions pending"

I don't see why this is a class method. What's wrong with allowing a thread to check whether another has pending exceptions?

I, too, think that "interrupt" is a term so loaded with hardware connotations that it should be avoided in this context. Also, async is sort of redundant, as all exceptions you can defer will be asynchronous ones.

Do you propose allowing a thread to defer delivery of exceptions to itself?

What happens if one writes:

```
Thread.current.raise Exception.new
```

Can this get deferred, in your new scheme?
Can the delivery of:

```
Kernel.raise Exception.new
```

be deferred?

Personally, I think that Thread.current.raise might get deferred, but Kernel.raise should bypass the queuing mechanism entirely.

- brent

--

Posted via <http://www.ruby-forum.com/>.

#28 - 11/30/2012 07:23 PM - brent (Brent Roman)

How about:

```
Thread.allow_exception(RuntimeError => :on_blocking) {  
  while true
```

...

- brent

--

Posted via <http://www.ruby-forum.com/>.

#29 - 11/30/2012 07:23 PM - ko1 (Koichi Sasada)

(2012/11/30 18:59), Brent Roman wrote:

Koichi,

Sorry about misspelling your name in my previous post. :-(

Kosaki-san is also a member of this discussion.

In reference to:

- `Thread.async_interrupted?`.
This method check any deferred async interrupts.

Ah, I want to say "This method checks any deferred async interrupts are there".

It is no ambiguous, I think.

I would suggest:

- `Thread.exceptions_pending?`
This method checks for any deferred exceptions.

"This method returns true if there are exceptions pending"

Now, I unify the name "async_interrupt". For example, "Interrupt" exception by C-c should be included (but now, it not supported).

I want to separate normal (sync) exception and async exception.

I don't see why this is a class method. What's wrong with allowing a thread to check whether another has pending exceptions?

Any use case?

Do you propose allowing a thread to defer delivery of exceptions to itself?

What happens if one writes:

```
Thread.current.raise Exception.new
```

Can this get deferred, in your new scheme?

Can the delivery of:

```
Kernel.raise Exception.new
```

be deferred?

No.

Personally, I think that `Thread.current.raise` might get deferred, but `Kernel.raise` should bypass the queuing mechanism entirely.

Now, Thread.current.raise is not async.
Because there is a test case it expect it is synchronous.
(No strong reason)

Thank you for your comment. But I can't accept the method name with simple *exception*. I want to emphasize it is *asynchronous* and very internal method.

In fact, Kosaki-san also don't like *async_interrupt*. We are very welcome to propose good name.

But I feel "async_interrupt_timing" and "async_interrupted" is good name. Of course, it is not a best name, I agree.

Thanks,
Koichi

--
// SASADA Koichi at atdot dot net

#30 - 12/01/2012 04:23 AM - brent (Brent Roman)

OK. I see the logic in using the term "interrupt" if you are actually trying to unify exceptions from other threads with handling of OS signals. However, both of these are generally thought of as being asynchronous events.

Try googling (with the quotes):

"asynchronous interrupt*" => 1,130,000 results
"synchronous interrupt*" => 180,000 results

If you insist on the `async_*` prefix, you should apply it consistently.
But, `Thread.control_async_interrupt` is getting quite cumbersome, no?

As someone who was writing ISRs for Intel 8080's and Zilog Z-80's in the late 1970's, here are my suggestions for more conventional vocabulary:

`Thread.control_interrupt` becomes `Thread.interruptible`
alternatives would be:
`Thread.allow_interrupt` or `Thread.enable_interrupt`
Any of these read better (to a native English speaker).
I like `interruptible` because it is a *property* of the thread being assigned by the construct. After all, nothing actually happens when this construct is executed. It affects what (might) happen later:

```
th = Thread.new{
  Thread.interruptible(RuntimeError => :on_blocking) {
    ...
  }
}
```

In General:
Code within the block passed to the `Thread.interruptible` method may or may not be interrupted according to the specification passed as its Hash argument.

In the example above, within the block passed to `Thread.interruptible`, the thread becomes interruptible by any `RuntimeError` when/if it waits for I/O or stops.

=====

The method `:async_interrupted?` would be better named:
`:interrupts_pending?`
A thread is not interrupted if it has interrupts being deferred.
The accepted idiom for this is to say the thread has interrupts pending for it.

The use case for defining `interrupts_pending?` method as Thread instance method is summarized on one word: debugging!

If you have a complex application that has threads which seem to be unresponsive, you'll want some way to tell whether those threads are ignoring pending interrupts, or whether they are not even getting interrupts delivered to them.

I'd also suggest adding another method:

```
Thread#interrupts_pending #without the question mark
```

This would return the number of pending interrupts for the thread. A thread might normally have 0, 1 or 2 pending interrupts. Seeing dozens pending would indicate a performance problem. This would be very useful information for debugging and optimization. A thread might even decide to take some drastic action to if it discovers that it has too many interrupts pending for itself.

Making Thread.current.raise act like sending exceptions to any other thread seemed more consistent to me because the method's behavior then has no special case for Thread.current. I have written low level code what processed a hardware interrupt, but then decided it must defer it for later and accomplished this by making the interrupt pending again, in the controller chip, but masked the interrupt in the CPU. However, I can see where this might break existing code that currently relies on Thread.current#raise being exactly synonymous with Kernel#raise Either behavior is workable.

- brent

--

Posted via <http://www.ruby-forum.com/>.

#31 - 12/03/2012 03:53 PM - headius (Charles Nutter)

On Fri, Nov 30, 2012 at 3:27 AM, Brent Roman brent@mbari.org wrote:

Charlie,
Five years ago, after a bit of arm twisting, you admitted that this technique could allow thread.raise to be used safely. Why the change of heart now?

I had to read through it a bit. Your proposal is quite a bit different from this one, since it sets threads uninterruptible by default and you have to opt-in. It would be safe, if you have to opt-in and you know exactly what that implies (including unsafe interrupting of ensure blocks). I haven't been following closely, but the proposal on the current thread is (I believe) intended to designate in Ruby code uninterruptible blocks of code. That would be safe too, if there's no change of interrupting between the time you enter an ensure block and when the uninterruptible section has started.

My bandwidth is limited, so I've been letting this issue go round and round for a while before I jump in much.

- Charlie

#32 - 12/03/2012 04:02 PM - ko1 (Koichi Sasada)

brent (Brent Roman) wrote:

OK. I see the logic in using the term "interrupt" if you are actually trying to unify exceptions from other threads with handling of OS signals. However, both of these are generally thought of as being asynchronous events.

I agree. They are "asynchronous events".

Try googling (with the quotes):

"asynchronous interrupt" => 1,130,000 results

"synchronous interrupt" => 180,000 results

If you insist on the async_ prefix, you should apply it consistently. But, Thread.control_async_interrupt(is getting quite cumbersome, no?

You are right.

As someone who was writing ISRs for Intel 8080's and Zilog Z-80's in the

late 1970's, here are my suggestions for more conventional vocabulary:

Thread.control_interrupt becomes Thread.interruptible
alternatives would be:

Thread.allow_interrupt or Thread.enable_interrupt

Any of these read better (to a native English speaker).

I like interruptible because it is a *property* of the thread being assigned by the construct. After all, nothing actually happens when this construct is executed. It affects what (might) happen later:

```
th = Thread.new{
  Thread.interruptible(RuntimeError => :on_blocking){
    ...
  }
}
```

"interruptible" makes sense for me.

But I feel it is ambiguous that this method returns only current interruptible flags.

How about to use `async_event' ?

In General:

Code within the block passed to the Thread.interruptible method may or may not be interrupted according to the specification passed as its Hash argument.

In the example above, within the block passed to Thread.interruptible, the thread becomes interruptible by any RuntimeError when/if it waits for I/O or stops.

=====

The method :async_interrupted? would be better named:
:interrupts_pending?

A thread is not interrupted if it has interrupts being deferred.

The accepted idiom for this is to say the thread has interrupts pending for it.

As non-native English speaker, I'm not sure the difference with "pending_interrupt?". Yours is good?

The use case for defining interrupts_pending? method as Thread instance method is summarized on one word: debugging!

It makes sense.

If you have a complex application that has threads which seem to be unresponsive, you'll want some way to tell whether those threads are ignoring pending interrupts, or whether they are not even getting interrupts delivered to them.

I'd also suggest adding another method:

```
Thread#interrupts_pending #without the question mark
```

This would return the number of pending interrupts for the thread.

A thread might normally have 0, 1 or 2 pending interrupts. Seeing dozens pending would indicate a performance problem. This would be very useful information for debugging and optimization. A thread might even decide to take some drastic action to if it discovers that it has too many interrupts pending for itself.

I don't like this method.

I like Thread#interrupts_pending?(err_class) what current Thread#async_interrupt? do.

Number is important? I don't think so.

Making Thread.current.raise act like sending exceptions to any other thread seemed more consistent to me because the method's behavior then has no special case for Thread.current. I have written low level code what processed a hardware interrupt, but then decided it must defer it

for later and accomplished this by making the interrupt pending again, in the controller chip, but masked the interrupt in the CPU. However, I can see where this might break existing code that currently relies on `Thread.current#raise` being exactly synonymous with `Kernel#raise`. Either behavior is workable.

I agree with it. Non *exception* is easy to understand.

Guys: any problem on it?

#33 - 12/03/2012 11:53 PM - tarui (Masaya Tarui)

Hi,

2012/11/30 SASADA Koichi ko1@atdot.net:

(2012/11/30 18:59), Brent Roman wrote:

Do you propose allowing a thread to defer delivery of exceptions to itself?

What happens if one writes:

`Thread.current.raise Exception.new`

Can this get deferred, in your new scheme?

Can the delivery of:

`Kernel.raise Exception.new`

be deferred?

No.

Personally, I think that `Thread.current.raise` might get deferred, but `Kernel.raise` should bypass the queuing mechanism entirely.

I also agree strongly.

Now, `Thread.current.raise` is not `async`.

Because there is a test case it expect it is synchronous.

(No strong reason)

really?

```
eval <<EOT
begin
  Thread.async_interrupt_timing(Object => :defer){
    p 1; Thread.current.raise "test"; p 2 }
  rescue
    p $!
  end
EOT
result is
1
2
#<RuntimeError: test>
```

it's seems as `async`.

Masaya TARUI

No Tool, No Life.

#34 - 12/04/2012 06:53 AM - brent (Brent Roman)

Regarding ability of `Thread.current.raise` to be deferred, if it works that way now, I'd vote to keep it this way.

Best not to have a special case for `Thread.current.raise`

If an application requires the special behavior, that's easily achieved:

```
class Thread
```

```
alias_method :original_raise, :raise
def raise exc
  Kernel.raise exc if self == Thread.current
  original_raise exc
end
end
```

However, the converse is *not* easily achieved.

If Thread.current.raise cannot be deferred, the only option would be to have another thread waiting to rescue the exception and immediately raise it back to the original thread. Not at all elegant.

--

Posted via <http://www.ruby-forum.com/>.

#35 - 12/04/2012 08:23 AM - brent (Brent Roman)

I was suggesting "interruptible" as a better alternative for "async_interrupt_timing" or "control_interrupt". Can either be called without a block? If so, does it change the way subsequent interrupts are delivered?

I'd like to avoid the use of "async" because it is an abbreviation for asynchronous. Ruby core method names tend to avoid abbreviations. That helps make the language more readable.

In light of all the ways "async_interrupt_timing" method can be used, perhaps (even better :) alternative names would be:

```
accept_interrupt(X => :immediately)
accept_interrupt(Y => :on_blocking)
accept_interrupt(Z => :never)
```

Or:

```
handle_interrupt(X => :immediately)
handle_interrupt(Y => :on_blocking)
handle_interrupt(Z => :never)
```

Handle interrupt X immediately. Handle interrupt Y on_blocking.
Handle interrupt Z never. You could also write:

```
asynchronous_event(X => :immediate)
asynchronous_event(Y => :on_blocking)
asynchronous_event(Z => :defer)
```

Or, (but this is getting a bit too long):

```
handle_asynchronous_event(X => :immediately)
handle_asynchronous_event(Y => :on_blocking)
handle_asynchronous_event(Z => :never)
```

My vote is for handle_interrupt or asynchronous_event, but all these read as idiomatically correct English jargon. I adjusted the values in the hashes slightly when using a verb phrase for the method name to make the resulting syntax more consistent with English grammar.

The
Thread#pending_interrupt?

method name you propose is also perfectly good English.
Either name is much more descriptive than Thread#async_interrupt?

But, enough syntax. Let's move on to semantics:

It is vital that further interrupts for a thread be deferred immediately after any asynchronous exception is raised in it. There is no other way to guarantee that ensure clauses run to completion. This deferral must happen even when the delivery policy is X=>:immediate ! Charles pointed this out earlier. I just assumed this would be the case. Can you please confirm?

My five year old proposal incorporated to two similar interrupt deferral policies to deal with the above issue. One was analogous to your :immediate, the other was called :always. The :always policy operated

exactly as Ruby does today. No interrupt queues or deferral. It is intended to provide compatibility with existing Ruby code, however conceptually flawed.

This new proposal adds the ability to assign different exception delivery policies to each subclass of Exception.

This seems good on the surface, but won't it complicate the queue management and make it possible for exceptions to be delivered out of order? Have you thought about this?

Have you considered timestamping asynchronous exceptions so the application can tell how long they had been deferred, and, much more importantly, sort them to determine the actual order in which they occurred?

I would suggest that, if you don't want to timestamp exceptions, you should drop the ability to apply different delivery policies to different object classes.

Another, less important issue, is having the ability to query the number of interrupts in the deferral queue. Soft real-time systems may change their behavior depending on the perceived backlog. But, more importantly, seeing a large number of deferred interrupts for a thread is a great debugging aid. Under high loads, a boolean test would not provide the needed information, as there will usually be one or two interrupts pending.

- brent

--

Posted via <http://www.ruby-forum.com/>.

#36 - 12/04/2012 03:53 PM - kosaki (Motohiro KOSAKI)

```
handle_interrupt(X => :immediately)
handle_interrupt(Y => :on_blocking)
handle_interrupt(Z => :never)
```

Handle interrupt X immediately. Handle interrupt Y on_blocking.
Handle interrupt Z never. You could also write:

```
asynchronous_event(X => :immediate)
asynchronous_event(Y => :on_blocking)
asynchronous_event(Z => :defer)
```

Or, (but this is getting a bit too long):

```
handle_asynchronous_event(X => :immediately)
handle_asynchronous_event(Y => :on_blocking)
handle_asynchronous_event(Z => :never)
```

I'm ok both `handle_interrupt` and `handle_asynchronous_event`.
(and I also agree `:defer` should go back `:never` if we accept this name)

My vote is for `handle_interrupt` or `asynchronous_event`, but all these read as idiomatically correct English jargon. I adjusted the values in the hashes slightly when using a verb phase for the method name to make the resulting syntax more consistent with English grammar.

The
`Thread#pending_interrupt?`

method name you propose is also perfectly good English.
Either name is much more descriptive than `Thread#async_interrupt?`

I'm ok this one too.

#37 - 12/17/2012 05:47 AM - headius (Charles Nutter)

Could someone write up a summary of the current status on the wiki? I'm having trouble sorting out what has been implemented and what's just

proposed at this point.

#38 - 12/22/2012 06:59 PM - ko1 (Koichi Sasada)

Thank you and sorry for my late response.

I will change method names as your proposal with the following patch.

<http://www.atdot.net/sp/view/fwdffm/readonly>

(2012/12/04 15:43), KOSAKI Motohiro wrote:

```
handle_interrupt(X => :immediately)
handle_interrupt(Y => :on_blocking)
handle_interrupt(Z => :never)
```

Handle interrupt X immediately. Handle interrupt Y on_blocking.
Handle interrupt Z never. You could also write:

```
asynchronous_event(X => :immediate)
asynchronous_event(Y => :on_blocking)
asynchronous_event(Z => :defer)
```

Or, (but this is getting a bit too long):

```
handle_asynchronous_event(X => :immediately)
handle_asynchronous_event(Y => :on_blocking)
handle_asynchronous_event(Z => :never)
```

I'm ok both handle_interrupt and handle_asynchronous_event.
(and I also agree :defer should go back :never if we accept this name)

My vote is for handle_interrupt or asynchronous_event, but all these read as idiomatically correct English jargon. I adjusted the values in the hashes slightly when using a verb phase for the method name to make the resulting syntax more consistent with English grammar.

The
Thread#pending_interrupt?

method name you propose is also perfectly good English.
Either name is much more descriptive than Thread#async_interrupt?

I'm ok this one too.

--
// SASADA Koichi at atdot dot net

#39 - 12/23/2012 07:19 PM - ko1 (Koichi Sasada)

- Status changed from Feedback to Closed

This issue was solved with changeset r38577.
Koichi, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

-
- thread.c: rename methods:
from Thread.async_interrupt_timing to Thread.handle_interrupt,
from Thread.async_interrupted? to Thread.pending_interrupt?.
Also rename option from 'defer' to 'never'.
[\[ruby-core:51074\]](#) [ruby-trunk - Feature [#6762](#)]
 - vm_core.c, thread.c: rename functions and data structure
'async_errinfo' to 'pending_interrupt'.
 - thread.c: add global variables sym_immediate, sym_on_blocking and
sym_never.
 - cont.c, process.c, vm.c, signal.c: ditto.
 - lib/sync.rb, lib/thread.rb: catch up this renaming.
 - test/ruby/test_thread.rb: ditto.