

## Ruby master - Bug #4283

### Timeout.timeout may cause application exit unintentionally

01/17/2011 01:53 PM - kosaki (Motohiro KOSAKI)

<b>Status:</b> Closed	
<b>Priority:</b> Normal	
<b>Assignee:</b> matz (Yukihiro Matsumoto)	
<b>Target version:</b> 1.9.3	
<b>ruby -v:</b> ruby 1.9.3dev (2010-12-22 trunk 30291) [x86_64-linux]	<b>Backport:</b>

#### Description

=begin  
This issue was discovered during [Bug#4266] discussion.  
Current timeout is racy.

Now, timeout module has following code.

```
def timeout()
begin
x = Thread.current
y = Thread.start {
begin
sleep sec
rescue => e
x.raise e
else
x.raise exception, "execution expired" if x.alive?
end
}
return yield(sec)
rescue exception => e
raise Error, e.message, e.backtrace
ensure
if y and y.alive?
// (1)
y.kill
y.join # make sure y is dead.
end
end
end
```

Then, A following race can occur.

CPU0(thread x)	CPU1(thread y)	remark
----------------	----------------	--------

```
begin
Thread.start
sleep sec
evaluate [user-defined-block]
y.alive?           return true
wake up from sleep
x.raise
```

Now, x is running at (1). Then ExitException which y raised can't be handled above rescue block. Then eventually, ExitException leak to caller and makes application exit.

=end

#### Related issues:

## Associated revisions

---

### Revision da3d9e99 - 05/18/2011 11:31 AM - kosaki (Motohiro KOSAKI)

- lib/timeout.rb (Timeout#timeout): don't leak "execution expired" exception. [Bug #4283] [ruby-core:34534].

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@31623 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

### Revision 31623 - 05/18/2011 11:31 AM - kosaki (Motohiro KOSAKI)

- lib/timeout.rb (Timeout#timeout): don't leak "execution expired" exception. [Bug #4283] [ruby-core:34534].

### Revision 31623 - 05/18/2011 11:31 AM - kosaki (Motohiro KOSAKI)

- lib/timeout.rb (Timeout#timeout): don't leak "execution expired" exception. [Bug #4283] [ruby-core:34534].

### Revision 31623 - 05/18/2011 11:31 AM - kosaki (Motohiro KOSAKI)

- lib/timeout.rb (Timeout#timeout): don't leak "execution expired" exception. [Bug #4283] [ruby-core:34534].

### Revision 31623 - 05/18/2011 11:31 AM - kosaki (Motohiro KOSAKI)

- lib/timeout.rb (Timeout#timeout): don't leak "execution expired" exception. [Bug #4283] [ruby-core:34534].

### Revision 31623 - 05/18/2011 11:31 AM - kosaki (Motohiro KOSAKI)

- lib/timeout.rb (Timeout#timeout): don't leak "execution expired" exception. [Bug #4283] [ruby-core:34534].

### Revision 31623 - 05/18/2011 11:31 AM - kosaki (Motohiro KOSAKI)

- lib/timeout.rb (Timeout#timeout): don't leak "execution expired" exception. [Bug #4283] [ruby-core:34534].

### Revision 67cf90f6 - 05/31/2011 12:12 AM - yugui (Yuki Sonoda)

## merges r31623 from trunk into ruby\_1\_9\_2.

- lib/timeout.rb (Timeout#timeout): don't leak "execution expired" exception. [Bug #4283] [ruby-core:34534].

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/branches/ruby\_1\_9\_2@31837 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

## History

---

### #1 - 01/17/2011 01:57 PM - kosaki (Motohiro KOSAKI)

- Assignee set to matz (Yukihiro Matsumoto)

```
=begin
Module maintainers list(http://redmine.ruby-lang.org/wiki/8/Maintainers) says
```

```
lib/timeout.rb
Yukihiro Matsumoto (matz)
```

```
Is this still correct?
=end
```

### #2 - 01/17/2011 04:50 PM - matz (Yukihiro Matsumoto)

```
=begin
Hi,
```

In message "Re: [ruby-core:34535] [Ruby 1.9-Bug#4283] Timeout.timeout may cause application exit unintetionally" on Mon, 17 Jan 2011 13:57:43 +0900, Motohiro KOSAKI [redmine@ruby-lang.org](mailto:redmine@ruby-lang.org) writes:

```
|Module maintainers list(http://redmine.ruby-lang.org/wiki/8/Maintainers) says
|
|lib/timeout.rb
```

| Yukihiro Matsumoto (matz)

|Is this still correct?

Yes, just because no one was willing to take over.

```
matz.
```

=end

### #3 - 01/18/2011 05:14 PM - kosaki (Motohiro KOSAKI)

=begin

|Is this still correct?

Yes, just because no one was willing to take over.

Oh, sad. Unfortunately I can't take it over too because my skill is not enough.  
However, I plan to watch thread and timeout related bug report and try to fix it awhile.

I'm feeling current ruby has too many pending concurrent issue.

Thanks.

=end

### #4 - 01/19/2011 06:37 AM - headius (Charles Nutter)

=begin

I tried to patch this many times, and it's very difficult. The problem here is that kill is inherently broken in the presence of an ensure block. I've discussed this on ruby-core emails in the past and on my blog here: <http://blog.headius.com/2008/02/rubys-threadraise-threadkill-timeoutrb.html>

You can improve the situation slightly by unconditionally killing the child thread once the user-defined block has completed, ignoring any errors. It is not, however, perfect: the child thread may still wake up between the end of the user-defined block and the call to kill (depending on green/GIL-threaded context switch boundaries and granularity).

This affected JRuby even more, since we have threads actually running concurrently and could not guarantee that only one thread would reach "kill" first or in what order they would reach it when run time and timeout time are very close.

Our solution was to move timeout.rb entirely into Java code, using java.util.concurrent's timed thread-pooled executor. In this model, we set the child thread (the killer) as a job for the executor to run after the specified amount of time. If that time expires, it does a normal Ruby kill in the running code. If the running code completes, we attempt to cancel the kill job; that will either succeed, if the job has not been submitted, or fail, if it has already begun to execute. In the latter case, we simply raise the timeout error.

The implementation is here: <https://github.com/jruby/jruby/blob/master/src/org/jruby/ext/Timeout.java#L71>

In this way, only one thread will successfully "kill" the other. I think the only way to ensure timeout behaves properly is to make it native, so it's in full control of the context-switching at a VM level.

=end

### #5 - 01/19/2011 06:44 AM - headius (Charles Nutter)

=begin

We also have a "load test" for Timeout here: [https://github.com/jruby/jruby/blob/master/test/load/load\\_timeout.rb](https://github.com/jruby/jruby/blob/master/test/load/load_timeout.rb)

This should run to completion, only ever printing "ok" or "timeout". Instead, all versions of MRI I tested will eventually let a timeout error "escape". The failure can be accelerated by adjusting the timeout and sleep times closer together.

=end

### #6 - 01/19/2011 01:13 PM - mame (Yusuke Endoh)

=begin

Hi,

2011/1/19 Charles Nutter [redmine@ruby-lang.org](mailto:redmine@ruby-lang.org):

I tried to patch this many times, and it's very difficult. The problem here is that kill is inherently broken in the presence of an ensure block. I've discussed this on ruby-core emails in the past and on my blog here: <http://blog.headius.com/2008/02/rubys-threadraise-threadkill-timeoutrb.html>

This is not just a problem of Thread#raise. Asynchronous signals (such as Interrupt caused by Ctrl+C) have the same problem. Of course, we cannot remove Ctrl+C.

Thus, to address this problem faithfully, we should provide a mechanism to safely handle asynchronous exceptions. Lobbying to eliminate only Thread#raise (and #kill) is not facing the reality.

Fortunately, there are some ancient wisdoms:

- "cancellation points" of pthread  
<http://www.kernel.org/doc/man-pages/online/pages/man7/pthreads.7.html>
- Asynchronous Exceptions in Haskell  
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.1040>

These two are based on a very similar idea: providing a feature to control whether asynchronous exceptions may be raised or may not. In fact, the latter is referred in comments of your blog article. But it was rejected as:

there are a lot of additional problems when implementing it in an environment that isn't as functionally pure as Haskell

I guess that this is misinterpretation. It is very similar to the former (cancellation points), and can be implemented even in imperative programming language, as pthread does. I don't know that they are compatible with Java (and/or JRuby) threads, though.

On a separate note, I'm not against deprecating Thread#raise. It is indeed too difficult to use correctly. Just eliminating it, however, is not enough.

the child thread may still wake up between the end of the user-defined block and the call to kill

Yes, it may occur. But does it cause any actual problem in the case of timeout.rb? Kosaki's patch seems to me good.

--  
Yusuke Endoh [mame@tsg.ne.jp](mailto:mame@tsg.ne.jp)

=end

#### #7 - 01/19/2011 03:21 PM - headius (Charles Nutter)

=begin  
On Tue, Jan 18, 2011 at 10:00 PM, Yusuke ENDOH [mame@tsg.ne.jp](mailto:mame@tsg.ne.jp) wrote:

This is not just a problem of Thread#raise. Asynchronous signals (such as Interrupt caused by Ctrl+C) have the same problem. Of course, we cannot remove Ctrl+C.

The JVM approach may be of interest here. JVMs spin up a signal-handling thread independent of the main thread. Signal-handlers are then triggered on this thread rather than on whatever thread happens to be running at any given time. Registering a signal handler gives it to that thread to be run when the signal fires (only via unofficial APIs, but the JVMs all work pretty similarly).

Ctrl+C is, by default, handled as a VM-level shutdown hook. All threads are terminated immediately, without any code continuing to run. Contrast this with MRI, where Ctrl+C triggers an Interrupt exception to be raised:

```
~/projects/jruby [] ruby -e 'begin; sleep; ensure; puts "I should be dead!"; end'  
^C should be dead!  
-e:1:in `sleep': Interrupt  
from -e:1
```

This decision means that Ctrl+C triggers finally blocks to run, which can then continue to block:

```
~/projects/jruby [] ruby -e 'begin; sleep; rescue Interrupt; puts "I do not want to die!"; begin; sleep; ensure; puts "Ok fine, I will die."; end; end'  
^C do not want to die!
```

<sup>COK</sup> fine, I will die.

-e:1:in `sleep': Interrupt  
from -e:1

This of course has identical problems to raising in another thread that might be running ensure logic, but Ctrl+C is a special case; it's probably valid to do almost anything, including a hard shutdown without running ensures at all.

I'm not familiar with how userland signal-handlers fire, but if they were guaranteed to run on a separate thread, the problems of interrupting a thread in an ensure would not be a problem.

Thus, to address this problem faithfully, we should provide a mechanism to safely handle asynchronous exceptions. Lobbying to eliminate only Thread#raise (and #kill) is not facing the reality.

Eliminating them (and fixing signal handling) eliminates the problem :) But I recognize that they're never going to go away. I've tried for years to make it happen.

Fortunately, there are some ancient wisdoms:

- "cancellation points" of pthread  
<http://www.kernel.org/doc/man-pages/online/pages/man7/pthreads.7.html>

- Asynchronous Exceptions in Haskell  
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.1040>

These two are based on a very similar idea: providing a feature to control whether asynchronous exceptions may be raised or may not. In fact, the latter is referred in comments of your blog article. But it was rejected as:

there are a lot of additional problems when implementing it in an environment that isn't as functionally pure as Haskell

I guess that this is misinterpretation. It is very similar to the former (cancellation points), and can be implemented even in imperative programming language, as pthread does. I don't know that they are compatible with Java (and/or JRuby) threads, though.

JRuby implements Thread#raise and #kill using exactly this mechanism. On boundaries roughly equivalent to MRI 1.8's thread context switches, we ping thread state to see if a kill or raise event has been sent to the thread. If so, we propagate that event. I documented these on a now-shutdown wiki, but you can figure them out by looking for places 1.8 checked "thread variables" to see if it should attempt a context switch. They are places like literal nil, newlines, every 256 calls (we do it every 1024, I believe), and so on.

In this way, we're able to implement #raise and #kill without using unsafe JVM thread operations like java.lang.Thread.stop and java.lang.Thread.stop(Throwable).

I don't see that other concurrent-threaded Ruby implementations will have any choice but to do it the same way.

The reason we still have the problems of asynchronous raising and killing is because the cancellation points are poorly defined. Since we based ours on MRI thread context-switch boundaries, there's simply too many of them (which is why we've removed a few and increased the call-count event ping). They also do not reflect the criticality of certain sections of code, like ensure bodies and code downstream from them.

As I said in my previous comment...it's not possible to make asynchronous exceptions safe with the above semantics (the semantics of Ruby thread context-switching, currently) intact. It may be possible to make asynchronous exceptions *safer* if we more clearly define these cancellation points and disable cancellation within certain contexts.

I must reiterate, however, that this is only a band-aid. Because Ruby is not "functionally pure", a top-level method might wrap everything in an ensure block. Should the entire application disable kill/raise? This would make many applications' threads totally immune to cancellation. Should leaving an ensure block by making a call re-enable cancellation? This hardly seems practical, since nearly everything in Ruby is a call...including the code the user wanted to "ensure" would run.

I don't want to be difficult. I would very much like to see a more formal thread event model put in place to "help" this issue, since it would make our job on JRuby (with concurrent threads) much easier.

On a separate note, I'm not against deprecating Thread#raise. It is indeed too difficult to use correctly. Just eliminating it, however, is not enough.

But it would be so nice for those of us implementing Ruby with real concurrent threads :) I've spent a lot of time thinking about these issues, and they're really hard to solve.

I do hope that any ideas going forward will keep concurrent-threaded implementations like JRuby in mind. We have worked very hard to emulate Ruby, and it would be very unfortunate if this work went in a direction impossible to implement safely or efficiently with concurrent threads.

the child thread may still wake up between the end of the user-defined block and the call to kill

Yes, it may occur. But does it cause any actual problem in the case of timeout.rb? Kosaki's patch seems to me good.

Yes, it should fix this particular problem. And it works because in this case, we actually *want* the asynchronous raise to interrupt the ensure block. My comments about the difficulty of fixing it were more appropriate for your other bug on fixing asynchronous exceptions.

This is also similar to what we do internally in JRuby, but we disable Ruby's normal #raise cancellation points entirely for one of the two threads. The patch would not guarantee safety on JRuby because thread context switches can occur at any time, which means the two threads could attempt to kill each other at exactly the same time. But it should work on non-concurrent implementations, I think.

- Charlie

=end

**#8 - 01/19/2011 07:34 PM - mame (Yusuke Endoh)**

=begin

Hi,

2011/1/19 Charles Oliver Nutter [headius@headius.com](mailto:headius@headius.com):

JRuby implements Thread#raise and #kill using exactly this mechanism. On boundaries roughly equivalent to MRI 1.8's thread context switches, we ping thread state to see if a kill or raise event has been sent to the thread.

I guess that you misunderstand cancellation points. They do not mean context-switch boundaries.

Usually, an operation that may suspend the execution is defined as cancellation points, such as blocking I/O, synchronization and sleep. In terms of Ruby, they roughly mean any operation that may change Thread#status to "sleep", such as IO#read, Mutex#lock, and sleep.

It is important for a programmer to be able to control whether his own program contains cancellation points or not.

I must reiterate, however, that this is only a band-aid. Because Ruby is not "functionally pure", a top-level method might wrap everything in an ensure block. Should the entire application disable kill/raise?

No, not at all.

According to that mechanism, asynchronous exceptions are raised when you run any cancellation point operation, even when asynchronous exceptions are prohibited.

```
begin
ensure

foo.bar # an Interrupt may not be raised
      # (unless these methods do not contain cancellation points)

sleep 1 # cancellation point; an Interrupt may be raised here

baz.qux # an Interrupt may not be raised

end

--
Yusuke Endoh mame@tsg.ne.jp

=end
```

#### #9 - 01/20/2011 06:58 AM - headius (Charles Nutter)

=begin  
On Wed, Jan 19, 2011 at 4:32 AM, Yusuke ENDOH [mame@tsg.ne.jp](mailto:mame@tsg.ne.jp) wrote:

I guess that you misunderstand cancellation points. They do not mean context-switch boundaries.

I think you misunderstood me. I know what cancellation points are. I just meant that in JRuby the cancellation points are roughly the same places in code where MRI has context switches. We designed it this way because MRI hands off async thread events by context-switching. This seemed like the closest analog.

For example, MRI 1.8 will check for a context switch when evaluating a literal "false". In JRuby, evaluating a literal "false" will ping for a raise or kill event on the current thread; in essence, literal "false" is a cancellation point in JRuby.

There are numerous other places during code evaluation and method invocation where we ping for thread events: IO operations, every N calls, after thread-related operations, and so on.

So I repeat: JRuby implements kill and raise using a mechanism similar to cancellation points, and those points occur at roughly the same places during execution that MRI 1.8 would context switch between threads.

Usually, an operation that may suspend the execution is defined as cancellation points, such as blocking I/O, synchronization and sleep. In terms of Ruby, they roughly mean any operation that may change Thread#status to "sleep", such as IO#read, Mutex#lock, and sleep.

These are all places where we check thread events for kill and raise. One such example:

RubyKernel.sleep calls RubyThread.sleep:

<https://github.com/jruby/jruby/blob/master/src/org/jruby/RubyKernel.java#L826>

RubyThread.sleep polls for thread events before and after the actual sleep operation:

<https://github.com/jruby/jruby/blob/master/src/org/jruby/RubyThread.java#L759>

pollThreadEvents checks thread "mail":

<https://github.com/jruby/jruby/blob/master/src/org/jruby/RubyThread.java#L759>

checkMail looks for KILL or RAISE events and responds accordingly:

<https://github.com/jruby/jruby/blob/master/src/org/jruby/RubyThread.java#L151>

Unless a JRuby thread reaches a point that calls pollThreadEvents, asynchronous KILL and RAISE will not be triggered. I feel this is very similar to cancellation points.

It is important for a programmer to be able to control whether his own program contains cancellation points or not.

That would be wonderful. Currently, as I mentioned in previous email, there need to be too many cancellation points, since MRI will respond to asynchronous kill and raise in *many* places. If we could reduce the number of those places and make it more explicit, it would be easier to control when asynchronous thread events should be handled and when they should be ignored.

I must reiterate, however, that this is only a band-aid. Because Ruby is not "functionally pure", a top-level method might wrap everything in an ensure block. Should the entire application disable kill/raise?

No, not at all.

According to that mechanism, asynchronous exceptions are raised when you run any cancellation point operation, even when asynchronous exceptions are prohibited.

```
begin
ensure

foo.bar # an Interrupt may not be raised
  # (unless these methods do not contain cancellation points)

sleep 1 # cancellation point; an Interrupt may be raised here

baz.qux # an Interrupt may not be raised

end
```

And it's certainly possible that calls within an ensure can trigger other exceptions, so once the ensure starts to run you obviously can't assume it will complete in all cases. Currently, Ruby responds to those asynchronous events on too fine-grained of boundaries (even responding to them as a result of making N calls!). In order for asynch thread events to be made "safe", we need to coarsen these boundaries and potentially allow users to opt out of them.

- Charlie

=end

**#10 - 01/20/2011 01:02 PM - mame (Yusuke Endoh)**

=begin  
Hi,

2011/1/20 Charles Oliver Nutter [headius@headius.com](mailto:headius@headius.com):

So I repeat: JRuby implements kill and raise using a mechanism similar to cancellation points, and those points occur at roughly the same places during execution that MRI 1.8 would context switch between threads.

Ah, I misunderstood you. Excuse me.

Indeed, a context switch is considered as cancellation point in the current MRI.

And it's certainly possible that calls within an ensure can trigger other exceptions, so once the ensure starts to run you obviously can't

assume it will complete in all cases. Currently, Ruby responds to those asynchronous events on too fine-grained of boundaries (even responding to them as a result of making N calls!). In order for asynch thread events to be made "safe", we need to coarsen these boundaries and potentially allow users to opt out of them.

Agreed.

--

Yusuke Endoh [mame@tsg.ne.jp](mailto:mame@tsg.ne.jp)

=end

**#11 - 01/28/2011 04:44 PM - usa (Usaku NAKAMURA)**

- Status changed from Open to Assigned

=begin

=end

**#12 - 05/18/2011 08:31 PM - kosaki (Motohiro KOSAKI)**

- Status changed from Assigned to Closed

- % Done changed from 0 to 100

This issue was solved with changeset r31623.  
Motohiro, thank you for reporting this issue.  
Your contribution to Ruby is greatly appreciated.  
May Ruby be with you.

- 
- lib/timeout.rb (Timeout#timeout): don't leak "execution expired" exception. [Bug [#4283](#)] [ruby-core:34534].

**#13 - 05/18/2011 08:36 PM - kosaki (Motohiro KOSAKI)**

I've committed timeout-race-fix.patch as r31623 and I confirmed it fixes Charles's load test (ie [https://github.com/jruby/jruby/blob/master/test/load/load\\_timeout.rb](https://github.com/jruby/jruby/blob/master/test/load/load_timeout.rb)). I know it doesn't fix Ctrl-C issue. However I think we can discuss it separately.

## Files

---

timeout-race-fix.patch	2.94 KB	01/17/2011	kosaki (Motohiro KOSAKI)
------------------------	---------	------------	--------------------------