

Ruby - Feature #8823

Run trap handler in an independent thread called "Signal thread"

08/27/2013 08:13 PM - ko1 (Koichi Sasada)

Status:	Rejected	
Priority:	Normal	
Assignee:	ko1 (Koichi Sasada)	
Target version:	2.6	
Description		
= Abstract		
How about to make an "Signal thread" to run trap handler?		
= Problem		
Now, all of thread synchronization methods are not permitted because there is a possibility of deadlock between trap handler and main thraed.		
For example:		
<pre>m = Mutex.new trap(:INT){ m.synchronization{...} } m.synchronization{ # recv SIGINT, and invoke trap handler ... }</pre>		
In this case, trap handler (a block passed to trap method) is run on the main thread.		
= Proposal		
Make a signal handler independent from main thread. If main thread and trap handler run in different threads, there are no such problem.		
= Implementation		
Don't create signal handler at first. But the first time we need a signal handler, signal handler is created by main thread.		
See timing chart (PDF) I attached.		
= Discussion		
== Advantage:		
<ul style="list-style-type: none">• Signal thread is independent on main thread, this means that you can use thread synchronization between trap handler and main thread. In other words, you can run any program in trap handler.• Simplify a path from sighandler to trap invocation thread (after creation of a signal thread)• Doesn't need a difficult implementation (modify is limited).		
== Disadvantage:		
<ul style="list-style-type: none">• There is a small compatibility issue because "Thread.current" on a trap handler is not a main thread.		
== Other Discussion:		
<ul style="list-style-type: none">• Create signal thread at first like timer thread is high cost. Without `trap`, we don't need a signal thread any more.• In signal handler and timer thread, we can't make a signal thread because creating "Ruby thread" (== signal thread) needs GVL. So the process path from timer thread to main thread is remained.		
== Other thought		

I know a philosophy that `trap` should run only a tiny program, without synchronization and so on. I agree with this philosophy. Current behaviour which prohibits synchronization features helps this philosophy. But I'm not sure which is good way for Ruby.

History

#1 - 08/27/2013 08:52 PM - ko1 (Koichi Sasada)

- File data-signal-2.pdf added

I add an additional implementation plan (called plan2) in attached PDF.

#2 - 08/27/2013 08:53 PM - akr (Akira Tanaka)

2013/8/27 ko1 (Koichi Sasada) redmine@ruby-lang.org:

Feature #8823: Run trap handler in an independent thread called "Signal thread"
<https://bugs.ruby-lang.org/issues/8823>

How about to make an "Signal thread" to run trap handler?

I think it is a right direction.

However, I know trap is used to interrupt a thread.
For example, irb raises an exception in a trap handler.

lib/irb.rb:

```
trap("SIGINT") do
  irb.signal_handle
end
...
def signal_handle
  ...
  case @signal_status
  when :IN_INPUT
    ...
    raise RubyLex::TerminateLineInput
  when :IN_EVAL
    IRB.irb_abort(self)
  ...
end
end
```

I suspect "raise RubyLex::TerminateLineInput" doesn't work as expected after your proposal is implemented because the current thread will be the signal thread.
It is described as the disadvantage:

== Disadvantage:

- There is a small compatibility issue because "Thread.current" on a trap handler is not a main thread.

Maybe, you should update irb (and possibly other libraries) as well as your signal thread implementation.

Tanaka Akira

#3 - 08/27/2013 08:53 PM - ko1 (Koichi Sasada)

(2013/08/27 20:45), Tanaka Akira wrote:

I suspect "raise RubyLex::TerminateLineInput" doesn't work as expected after your proposal is implemented because the current thread will be the signal thread.

I wrote a countermeasure for this issue only in PDF.

All of exceptions raised in trap handler, will be sent to the main threads (like: main_thread#kill(raise_exception)).

I think this spec reduce incompatibility problems.
What do you think about it?

--
// SASADA Koichi at atdot dot net

#4 - 08/27/2013 09:23 PM - akr (Akira Tanaka)

2013/8/27 SASADA Koichi ko1@atdot.net:

I think this spec reduce incompatibility problems.
What do you think about it?

I suspect that it causes race conditions.

The exception may be reached to the main thread with unexpected delay.

Tanaka Akira

#5 - 08/28/2013 04:53 AM - normalperson (Eric Wong)

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

== Disadvantage:

- There is a small compatibility issue because "Thread.current" on a trap handler is not a main thread.

This adds race conditions in existing code.

There is no easy way for existing code to be safe with both your proposed implementation and the current trap implementation.
I can't safely add mutex.synchronize{} in existing trap handler.

Perhaps this version should be something like:

```
trap_with_thread(:INT) { ... }
```

== Other thought

I know a philosophy that 'trap' should run only a tiny program, without synchronization and so on. I agree with this philosophy. Current behaviour which prohibits synchronization features helps this philosophy. But I'm not sure which is good way for Ruby.

Alternatively, queue trap handler execution in VM and make them uninterruptible (at Ruby-level) when running. This is less likely incompatible, but can slow down execution of (crazy) apps which disagree with this philosophy and do too much in trap.

#6 - 08/29/2013 03:28 AM - kosaki (Motohiro KOSAKI)

Alternatively, queue trap handler execution in VM and make them uninterruptible (at Ruby-level) when running. This is less likely incompatible, but can slow down execution of (crazy) apps which disagree with this philosophy and do too much in trap.

Could you elaborate a bit more?
If you mean synchronize{} automatically disable interrupt, it doesn't work when using cond-wait.

```
mutex.lock  
cond.wait <- thread is here. and interrupt never be happen.  
mutex.unlock
```

If you mean, trap handler automatically disable nested trap, we already do since 1.9.3.

#7 - 08/29/2013 04:53 AM - kosaki (Motohiro KOSAKI)

If you mean, trap handler automatically disable nested trap, we already do since 1.9.3.

That's what I meant, oops :x

Yay, I was cool! :)

#8 - 08/29/2013 07:07 PM - ko1 (Koichi Sasada)

- Status changed from Open to Feedback

After discussion, I decide to reject this feature.

How about to permit Queue operation in trap handler with [ruby-trunk - Feature [#3620](#)], and use Queue#push to synchronize/communicate between other threads. This is Ruby-level alternative of pipe hack.

Comments are welcome.

#9 - 08/30/2013 10:53 AM - akr (Akira Tanaka)

2013/8/29 ko1 (Koichi Sasada) redmine@ruby-lang.org:

How about to permit Queue operation in trap handler with [ruby-trunk - Feature [#3620](#)], and use Queue#push to synchronize/communicate between other threads. This is Ruby-level alternative of pipe hack.

I'm not sure how to use it.

The pipe hack is usable because we can use select/poll system call to multiplex file descriptors.

How can an application wait multiple queues or a queue and other blocking operations?
polling, threads, or other?

I think you should write toy example application.

Tanaka Akira

#10 - 08/30/2013 11:53 AM - ko1 (Koichi Sasada)

(2013/08/30 10:45), Tanaka Akira wrote:

I think you should write toy example application.

For example:

```
def trap_in_thread(sig, &b)
  q = Queue.new
  Thread.new{
    q.pop
    b.call
  }
  trap(sig){
    q.push true
  }
end
```

--

// SASADA Koichi at atdot dot net

#11 - 09/30/2013 08:13 PM - ko1 (Koichi Sasada)

- Target version changed from 2.1.0 to 2.6

Now, Queue is trap-safe.

#12 - 10/10/2013 06:12 PM - ko1 (Koichi Sasada)

- Status changed from Feedback to Rejected

Files

data-signal.pdf	179 KB	08/27/2013	ko1 (Koichi Sasada)
data-signal-2.pdf	186 KB	08/27/2013	ko1 (Koichi Sasada)