Backport191 - Backport #2564

[patch] re-initialize timer_thread_{lock,cond} after fork

01/06/2010 10:19 PM - alk (Aliaksey Kandratsenka)

Status:	Closed	
Priority:	Normal	
Assignee:	shyouhei (Shyouhei Urabe)	

Description

=begin
After fork ruby clears timer_thread_id and then creates new timer thread in new process. But because new process inherits memory of old process it may see locked timer_thread_lock and well as invalid state of timer_thread_cond.

This patch simply initializes both this variables before their first use.

This bug causes lockup of Rails apps under phusion passenger on ruby 1.9.1.
=end

History

#1 - 01/07/2010 06:44 AM - hongli (Hongli Lai)

=begin
I believe you may have to unlock timer_thread_lock before reinitializing it. This reinitialization-at-fork is also done for the GIL but I believe a while ago someone reported a bug, which was fixed by unlocking the GIL before reinitializing it.
=end

#2 - 01/07/2010 06:56 PM - alk (Aliaksey Kandratsenka)

=begin
I don't think you should unlock the mutex before re-initializing it. Because a) it can be unsafe depending on threading implementation details b) pthread_mutex_init doesn't assume any previous state of mutex it inits and must not.
=end

#3 - 01/08/2010 10:49 AM - naruse (Yui NARUSE)

- Status changed from Open to Assigned

- Assignee set to nobu (Nobuyoshi Nakada)

=begin

=end

#4 - 01/26/2010 04:01 PM - alk (Aliaksey Kandratsenka)

=begin
Do you need some further comments or help ? I'd like to get this verified/applied, 'cause it seems quite trivial to me.
=end

#5 - 01/26/2010 04:51 PM - naruse (Yui NARUSE)

=begin
Can you make minimum reproducing program?
It can be either minumum C code for specific environment or Ruby code.
=end

#6 - 01/27/2010 04:57 AM - rogerdpack (Roger Pack)

=begin
I know there was some change to mutexes recently:

http://redmine.ruby-lang.org/issues/show/2394

did that help at all?
-r
=end

#7 - 01/27/2010 07:28 AM - alk (Aliaksey Kandratsenka)

=begin
Yes I can. But I'd like to avoid doing that. Reliable reproduction of such tiny races is not trivial. I have better ways to spend my time. Isn't this patch trivial ?

Your're running timer thread that sleeps most of the time with timer_thread_lock unlocked. But periodically it wakes up and runs timer_thread_function. It does it with timer_thread_lock locked. This is 100% ok. This allows you to reliably send 'please-die' signal to this thread (in native_stop_timer_thread). (I would do it without cleanup at all. It's not very useful after all. But that's your code.)

When some thread does fork timer_thread_lock will by unlocked most of the time, just because this thread sleeps most of the time. But it's possible to fork in a time when this lock is taken or is in transition to taken state. When this happens new process will simply stall trying to take this lock in rb_thread_create_timer_thread. Because this lock and associated condition is never accessed before rb_thread_create_timer_thread is called I propose to initialize both of them just before forking-off timer thread. This way original or forked process will always start with correct state of them.

Fancy locking & signaling in timer thread startup code can be removed too, but that's another story. It's not bug at all.

=end

#8 - 01/27/2010 07:30 AM - alk (Aliaksey Kandratsenka)

=begin
No. http://redmine.ruby-lang.org/issues/show/2394 is unrelated.
=end

#9 - 01/27/2010 08:27 PM - kosaki (Motohiro KOSAKI)

=begin
Hi Aliaksey,

native_stop_timer_thread() sleep until timer thread die by pthread_join().

If your patch fixes the issue. It mean
1) we don't call native_thread_join(). or
2) native_thread_join() is corrupted. or
3) non-timer thread take timer_thread_lock.

I think all above scenario are bug and we need more deep analysis. Can you please consider to spend some time to make minimum reproduce program?
=end

#10 - 01/27/2010 11:47 PM - alk (Aliaksey Kandratsenka)

=begin
It seems I'm not stating the problem clearly enough.

What happens:
a) lets suppose timer thread is doing some periodic work. This means that thread_thread_lock is taken.
b) at this instant of time we Kernel#fork. Memory is copied, but not threads.
c) Child ruby process now tries to spawn it's own timer thread
d) as part of timer thread spawn procedure it tries to take timer_thread_lock, but because it sees it in locked state it cannot succeed.
e) Child simply hangs forever.

=end

#11 - 01/28/2010 03:32 PM - naruse (Yui NARUSE)

=begin
My understanding is following:

a) lets suppose timer thread is doing some periodic work. This means that thread_thread_lock is taken.
b0) at this instant of time we Kernel#fork
b1) rb_f_fork()
b2) rb_fork()
b3) rb_fork_err()
b4) before_fork()
b5) before_exec()
b6) (rb_enable_interrupt(), (forked_child ? 0 : (rb_thread_stop_timer_thread(), 1)))
b7) rb_thread_stop_timer_thread()
b8) native_stop_timer_thread() (thread_pthread.c)
b9) stop timer thread
..
b98) fork()
b99) Memory is copied, but not threads.
c) Child ruby process now tries to spawn it's own timer thread
d) as part of timer thread spawn procedure it tries to take timer_thread_lock, and the lock is free.
=end

#12 - 01/28/2010 06:51 PM - alk (Aliaksey Kandratsenka)

=begin
Indeed. I was wrong. But there's still bug in 1.9.1. native_stop_timer_thread() doesn't wait till timer thread is dead. trunk does pthread_join (which implies that lock is free), and 1.9.1-243 does not. See http://redmine.ruby-lang.org/repositories/entry/ruby-191/thread_pthread.c line 813.

You should merge r25629 to 1.9.1. My patch is not necessary. And it doesn't solve issue completely as I see now. 'cause child can see some bad state left by running timer_thread_function.

BTW, you can simplify rb_thread_create_timer_thread. There's no need to hold lock around pthread_create and there's no need to signal timer thread condition there. timer thread observes system_working in a reliable way, so there's no race with immediate native_stop_timer_thread and there's no gain in that for rb_thread_create_timer_thread.

=end

#13 - 02/04/2010 04:42 PM - alk (Aliaksey Kandratsenka)

=begin
I'm sorry to bother you again, but merging r25629 to 1.9.1 should be really trivial. Without that fix there's obvious race in native_stop_timer_thread that causes new process to see locked timer_thread_lock.

=end

#14 - 02/04/2010 06:02 PM - naruse (Yui NARUSE)

- Category set to core

- Assignee changed from nobu (Nobuyoshi Nakada) to shyouhei (Shyouhei Urabe)

=begin
backport r25629.
=end

#15 - 02/28/2010 11:07 PM - alk (Aliaksey Kandratsenka)

=begin
I have to admit my mistake again. There is a call to native_thread_join in 1.9.1, it's in rb_thread_stop_timer_thread. So there's some other reason for the bug that we got. Maybe OS bug, maybe something else. I'll ask the guys who suffer this bug for more data. It's unlikely that we'll be able to produce small test program to reproduce the bug.

=end

#16 - 03/18/2010 06:11 PM - alk (Aliaksey Kandratsenka)

=begin
The folks who experienced this bug do not experience it anymore. I assume it was kernel bug. This ticket can be closed now.
=end

#17 - 03/20/2010 12:47 AM - rogerdpack (Roger Pack)

- Status changed from Assigned to Closed

=begin
Thanks for the feedback. Closing.
=end

Files

0001-re-initialize-timer_thread_-lock-cond-after-fork.patch	902 Bytes	01/06/2010	alk (Aliaksey Kandratsenka)