Ruby master - Bug #16288

Segmentation fault with finalizers, threads

10/31/2019 11:02 PM - davidw (David Welton)

Status:	Open		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:	ruby 2.6.6p116 (2019-10-02 revision 67825) [x86_64-linux]	**Backport:**	2.5: UNKNOWN, 2.6: UNKNOWN

Description

Hi,

This is a tricky one and I am still working on narrowing it down, but I will report what I have so far.

I compiled a version of 2_6_6 from github: ruby 2.6.6p116 (2019-10-02 revision 67825) [x86_64-linux]

I have a minimal Rails project that uses Mongoid. It crashes with a segmentation fault when rspec runs. The concurrent ruby gem is in some way involved, and I have been posting there: https://github.com/ruby-concurrency/concurrent-ruby/issues/808

However, I think there is a deeper problem - I would not expect a user level script to cause a segmentation fault.

I have been putting a lot of debugging statements in, and turned on Thread.DEBUG, and have noticed some things. I am not experienced with Ruby's internals, so some of these bits of data might be normal or irrelevant:

- The concurrent-ruby gem uses ObjectSpace.define_finalizer to set a finalizer
- That finalizer creates a new Thread
- However, it appears as if that thread is running after the main thread is already dead, so code that expects to reference the main thread crashes, because it's a NULL reference.

I tried the following test code:

```
class Foo
  def initialize
    ObjectSpace.define_finalizer(self, proc do
                                   Foo.foo_finalizer
                                 end)
  end

  def bar
    puts 'bar'
  end

  def Foo.foo_finalizer
    puts "foo_finalizer"
    t = Thread.new do
      puts "Thread reporting for duty"
    end
    puts "foo_finalizer thread launched"
    sleep 5
  end
end

f = Foo.new
f.bar
f = nil
```

While trying to develop a simple test case to demonstrate the problem. It triggers rb_raise(rb_eThreadError, "can't alloc thread"); in thread_s_new, because it looks like the main thread has already been marked as 'killed' in this case. When I check the main thread status in thread_s_new with the above code, it reports 'dead'.

When I run my rspec code in the sample Rails project, thread_s_new shows the main thread's status as 'run' even if it should be

dead?

I have seen some debugging things that shows some exceptions and thread_join interrupts and so on.

Is it possible that something like this is happening?

Main thread starts doing a cleanup, and gets an exception or something that generates an interrupt, and its KILLED status gets reset to RUNNABLE

Then, in the finalizer, it starts creating a Thread, but at this point the main thread actually does get killed, and when that finalizer thread tries to run it runs into a null reference?

I can provide the Rails sample project if needs be.

Sorry if any of the above isn't clear; I've been staring at the C code for several hours and am a bit cross-eyed!

Thank you for any insights.

History

#1 - 11/01/2019 04:14 PM - davidw (David Welton)

A little bit more data:

I added some more debugging statements, and I found that the main thread goes from being marked with the THREAD_KILLED status to THREAD_STOPPED_FOREVER

This appears to happen in thread_join_sleep in thread.c.

After this happens, it would be possible for new threads to be started in thread_s_new, although I'm not sure why the check there happens *after* the rb_thread_alloc call.

#2 - 11/02/2019 01:22 AM - mame (Yusuke Endoh)

Thank you for the report and the great investigation! I could reproduce the issue by using your example: https://github.com/mainameiz/segfault_app

Currently, starting a thread in a finalizer is dangerous. The termination of the interpreter is: (1) kill all threads except the main thread, (2) run all finalizers, and (3) destruct all (including all mutexes, the main thread, timer thread, VM itself, etc.). If a finalizer creates a thread, it starts running during or after (3), which leads to a catastrophic situation.

An easy solution is to prohibit thread creation after the process (3) is started. The following patch fixes the segfault of your example.

```
diff --git a/thread.c b/thread.c
index eff5d39b51..fc609907ef 100644
--- a/thread.c
+++ b/thread.c
@@ -833,6 +833,11 @@ thread_create_core(VALUE thval, VALUE args, VALUE (*fn)(void *))
     rb_raise(rb_eThreadError,
          "can't start a new thread (frozen ThreadGroup)");
     }
+    if (current_th->vm->main_thread->status == THREAD_KILLED) {
+        rb_warn("can't start a new thread after the main thread has stopped");
+        rb_raise(rb_eThreadError,
+                 "can't start a new thread (the main thread has already terminated)");
+    }

     if (fn) {
         th->invoke_type = thread_invoke_type_func;
```

By this patch, your example ends gracefully.

```
$ bundle exec rspec spec/models/user_spec.rb
/home/mame/work/ruby/local/lib/ruby/gems/2.7.0/gems/mongoid-7.0.5/lib/mongoid.rb:104: warning: The last argume
nt is used as the keyword parameter
/home/mame/work/ruby/local/lib/ruby/gems/2.7.0/gems/activesupport-6.0.0/lib/active_support/core_ext/module/del
egation.rb:171: warning: for `delegate' defined here
config.eager_load is set to nil. Please update your config/environments/*.rb files accordingly:

  * development - set it to false
  * test - set it to false (unless you use a tool that preloads your test environment)
  * production - set it to true

/home/mame/work/ruby/local/lib/ruby/gems/2.7.0/gems/tzinfo-1.2.5/lib/tzinfo/ruby_core_support.rb:142: warning:
```

```
 The last argument is used as the keyword parameter
/home/mame/work/ruby/local/lib/ruby/gems/2.7.0/gems/tzinfo-1.2.5/lib/tzinfo/ruby_core_support.rb:142: warning:
 The last argument is used as the keyword parameter
/home/mame/work/ruby/local/lib/ruby/gems/2.7.0/gems/actionpack-6.0.0/lib/action_dispatch/middleware/stack.rb:3
7: warning: The last argument is used as the keyword parameter
/home/mame/work/ruby/local/lib/ruby/gems/2.7.0/gems/actionpack-6.0.0/lib/action_dispatch/middleware/static.rb:
110: warning: for `initialize' defined here
.

Finished in 0.00977 seconds (files took 1.22 seconds to load)
1 example, 0 failures

/home/mame/work/ruby/local/lib/ruby/2.7.0/timeout.rb:85: warning: can't start a new thread on a finalizer
/home/mame/work/ruby/local/lib/ruby/2.7.0/timeout.rb:85: warning: can't start a new thread on a finalizer
```

However, as the last two lines show, Timeout cannot be used safely in a finalizer. I'm unsure if it is acceptable, but to support thread creation in a finalizer, we need to revamp the termination process. ko1 (Koichi Sasada) and nobu (Nobuyoshi Nakada), what do you think?

davidw (David Welton) I could be wrong as I don't understand your statement about thread_join, but I couldn't see the behavior by running your example under gdb. Anyways, thanks for the great investigation. It is really helpful.

#3 - 11/04/2019 05:11 PM - davidw (David Welton)

mame (Yusuke Endoh) wrote:

> Thank you for the report and the great investigation! I could reproduce the issue by using your example:
> https://github.com/mainameiz/segfault_app

Thanks for checking in to it.

> Currently, starting a thread in a finalizer is dangerous. The termination of the interpreter is: (1) kill all threads except the main thread, (2) run all finalizers, and (3) destruct all (including all mutexes, the main thread, timer thread, VM itself, etc.). If a finalizer creates a thread, it starts running during or after (3), which leads to a catastrophic situation.

Yes, I think there are multiple parts to this bug. It makes sense to me that starting threads in a finalizer like the concurrenty-ruby gem is doing is not correct.

> An easy solution is to prohibit thread creation after the process (3) is started. The following patch fixes the segfault of your example.

Yes, that prevents the segfault in my test application as well.

> However, as the last two lines show, Timeout cannot be used safely in a finalizer. I'm unsure if it is acceptable, but to support thread creation in a finalizer, we need to revamp the termination process. ko1 (Koichi Sasada) and nobu (Nobuyoshi Nakada), what do you think?
>
> davidw (David Welton) I could be wrong as I don't understand your statement about thread_join, but I couldn't see the behavior by running your example under gdb. Anyways, thanks for the great investigation. It is really helpful.

Apologies, I was just kind of guessing at what was going on - I am not at all familiar with Ruby's internals! I think the interaction is kind of complex, as you can see from the simple code I posted in the first bug report, which only tries to launch a thread in a finalizer - and fails. It's strange that the timeout.rb code is able to successfully create a thread.

Thank you for your prompt response and looking into the problem.

#4 - 11/04/2019 09:49 PM - davidw (David Welton)

```
modified   lib/timeout.rb
@@ -94,7 +94,7 @@ def timeout(sec, klass = nil, message = nil)   #:yield: +sec+
       ensure
         if y
           y.kill
-          y.join # make sure y is dead.
+          # y.join # make sure y is dead.
         end
       end
     end
```

This also stops the segfault from happening.

In my test app, the call to timeout seems to be coming from the Mongo gem:

```
      def connect!
```

```
        Timeout.timeout(options[:connect_timeout], Error::SocketTimeoutError) do
          socket.setsockopt(IPPROTO_TCP, TCP_NODELAY, 1)
          handle_errors { socket.connect(::Socket.pack_sockaddr_in(port, host)) }
          self
        end
      end
```

When I comment out that join, I get the same error message as with my simple example I pasted in the original report:

```
Exception `ThreadError' at /home/davidw/.rvm/gems/ruby-2.6.5/gems/concurrent-ruby-1.1.5/lib/concurrent/atomic/
ruby_thread_local_var.rb:87 - can't alloc thread
```

I think that join is somehow marking the main thread as being in some state other than KILLED...?

#5 - 11/14/2019 03:45 AM - davidw (David Welton)

This code pretty reliably produces a segmentation fault on my machine:

```
require 'timeout'

Thread.DEBUG = 1

class Foo
  def initialize
    ObjectSpace.define_finalizer(self, proc do
                                   Foo.foo_finalizer
                                 end)
  end

  def bar
    puts 'foo'
  end

  def Foo.foo_finalizer
    STDERR.puts "finalizing a Foo"
    Thread.new do
      sleep 5
      STDERR.puts "finalizing foo thread done"
    end
  end
end

class Bar
  def initialize
    ObjectSpace.define_finalizer(self, proc do
                                   Bar.bar_finalizer
                                 end)
  end

  def foo
    puts 'foo'
  end

  def Bar.bar_finalizer
    Timeout::timeout(2) do
      100.times do
        f = Foo.new
        f.bar
      end
    end
  end
end

b = Bar.new
b.foo
```

Written like that, it looks weird, but the fact that it's appearing for a lot of people is because there is some combination of finalizers and threads being used by, I think, the mongo, mongoid and concurrent-ruby gems, which is why a number of people have reported problems in the github issue I linked in the initial report. And I would guess that a larger number are simply seeing the segmentation fault and not knowing where to file a report.

Something that I *think* is important here is the Timeout. That runs a join against the main thread, and I *think* (I am not familiar with this code!) that is tickling thread_join_sleep, which does th->status = THREAD_STOPPED_FOREVER;, so that then in thread_s_new, it doesn't raise the "can't alloc thread" exception, because it's in a SLEEP_FOREVER state.

The previously mentioned patch that checks the main thread status in thread_create_core would likely break any code that uses Timeout in a

finalizer.

Does that make sense?

#6 - 12/10/2019 01:16 AM - davidw (David Welton)

I was fiddling around a bit, and trying to understand how things work, for my own edification.

```
diff --git a/eval.c b/eval.c
index 1eeaec56cb..097a105b33 100644
--- a/eval.c
+++ b/eval.c
@@ -238,6 +238,7 @@ ruby_cleanup(volatile int ex)

     ruby_finalize_1();

+    rb_thread_terminate_all();
     /* unlock again if finalizer took mutexes. */
     rb_threadptr_unlock_all_locking_mutexes(GET_THREAD());
     EC_POP_TAG();
diff --git a/thread.c b/thread.c
index eca14b4b4c..4e6abb94ba 100644
--- a/thread.c
+++ b/thread.c
@@ -682,7 +682,7 @@ thread_do_start(rb_thread_t *th)
         else {
             args_ptr = RARRAY_CONST_PTR(args);
         }
-
+        rb_funcallv(NULL, idInspect, 0, 0);
         th->value = rb_vm_invoke_proc(th->ec, proc,
                                       (int)args_len, args_ptr,
                                       VM_BLOCK_HANDLER_NONE);
```

Makes it not crash.

I think the 'correct' thing would be to actually wait on the threads to finish.

Also, I have no idea why that 'inspect' works; it must cause some side effect that I'm unaware of that makes thread_do_start not crash on the call to rb_vm_invoke_proc.

#7 - 12/19/2019 07:46 PM - davidw (David Welton)

```
--- a/thread.c
+++ b/thread.c
@@ -682,7 +682,7 @@ thread_do_start(rb_thread_t *th)
         else {
             args_ptr = RARRAY_CONST_PTR(args);
         }
-
+        rb_funcallv(NULL, idInspect, 0, 0);
         th->value = rb_vm_invoke_proc(th->ec, proc,
                                       (int)args_len, args_ptr,
                                       VM_BLOCK_HANDLER_NONE);
```

I replaced the funcall with

```
RUBY_VM_CHECK_INTS(GET_EC());
```

And everything seems to work ok.

I realized that even in 'normal' circumstances, Ruby does not wait for threads to finish; the programmer must request that, so I'm not sure that's actually correct.

With the above patch, I don't get the segfault any more.

I don't have nearly enough familiarity with the internals to know if this is actually the best or most correct way of fixing the problem though.

Thank you