Ruby - Bug #18730

Double `return` event handling with different tracepoints

04/14/2022 02:50 PM - hurricup (Alexandr Evstigneev)

Status:	Closed		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:		**Backport:**	2.6: UNKNOWN, 2.7: UNKNOWN, 3.0: UNKNOWN, 3.1: UNKNOWN

Description

I'm not sure if this is a bug or intentional behavior, but feels a bit unexpected. Ruby 3.0.x, 3.1.x affected.

Here is the script demonstrating the issue:

```
def bar
  42 #bp here
end

tp_line = TracePoint.new(:line) do |tp0|
  puts "Got line event from #{tp0.path}:#{tp0.lineno}"

  tp_multi1 = TracePoint.new(:return, :b_return, :line) do |tp|
    if tp.lineno == 3
      puts "Got first return `#{tp.event}` from #{tp.path}:#{tp.lineno}"
      tp.disable
      # tp0.disable # uncommenting this line changes things to the more expected


      tp_multi2 = TracePoint.new(:return, :b_return, :line) do |tps|
        puts "Got second return `#{tps.event}` from #{tps.path}:#{tps.lineno}"
      end
      tp_multi2.enable(target: RubyVM::InstructionSequence.of(method :bar))
    end
  end
  tp_multi1.enable
end

tp_line.enable(target: RubyVM::InstructionSequence.of(method :bar))

bar
```

1. We set a line TP to the bar method iseq (consider it a line breakpoint)
2. When line event is triggered we setting another untargeted tracepoint for the same method, to catch line, return and b_return events (consider it attempt to step into something)
3. When return event of the bar method is triggered, we disabling second tracepoint and setting another one, targeted to the same method and multiple events.

Output i get:

```
Got line event from /home/hurricup/test.rb:2
Got first return `return` from /home/hurricup/test.rb:3
Got second return `return` from /home/hurricup/test.rb:3
```

The questions are:

1. why return triggered on the second tracepoint, when we already handeled it?
2. why disabling line tracepoint changes behavior?

Associated revisions

Revision a687756284187887835aa345adc89b2718054e4a - 05/30/2022 05:54 PM - alanwu (Alan Wu)

Fix use-after-free with interacting TracePoints

vm_trace_hook() runs global hooks before running local hooks.
Previously, we read the local hook list before running the global hooks
which led to use-after-free when a global hook frees the local hook
list. A global hook can do this by disabling a local TracePoint, for
example.

Delay local hook list loading until after running the global hooks.

Issue discovered by Jeremy Evans in GH-5862.

[Bug #18730]

Revision a687756284187887835aa345adc89b2718054e4a - 05/30/2022 05:54 PM - alanwu (Alan Wu)

Fix use-after-free with interacting TracePoints

vm_trace_hook() runs global hooks before running local hooks.
Previously, we read the local hook list before running the global hooks
which led to use-after-free when a global hook frees the local hook
list. A global hook can do this by disabling a local TracePoint, for
example.

Delay local hook list loading until after running the global hooks.

Issue discovered by Jeremy Evans in GH-5862.

[Bug #18730]

Revision a6877562 - 05/30/2022 05:54 PM - alanwu (Alan Wu)

Fix use-after-free with interacting TracePoints

vm_trace_hook() runs global hooks before running local hooks.
Previously, we read the local hook list before running the global hooks
which led to use-after-free when a global hook frees the local hook
list. A global hook can do this by disabling a local TracePoint, for
example.

Delay local hook list loading until after running the global hooks.

Issue discovered by Jeremy Evans in GH-5862.

[Bug #18730]

History

#1 - 04/15/2022 11:12 AM - hurricup (Alexandr Evstigneev)

- Description updated

#2 - 04/26/2022 08:40 PM - jeremyevans0 (Jeremy Evans)

hurricup (Alexandr Evstigneev) wrote:

> The questions are:
>
> 1. why return triggered on the second tracepoint, when we already handeled it?

You are adding and enabling a separate return event tracepoint (tp_multi2) on the same method before the method returns, while it is still in the process of handling the first return event tracepoint (tp_multi1). I don't think it is unexpected that this tracepoint (tp_multi2) would also be called. Note that if you add a tp_multi2.disable call directly after the tp_multi2.enable call, you don't get the second return event printed.

> 1. why disabling line tracepoint changes behavior?

On the master branch, if I uncomment the tp0.disable call, I get a segfault with the following backtrace:

```
Thread 1 received signal SIGSEGV, Segmentation fault.
0x00000885cb83e9eb in exec_hooks_body (ec=0x884de217650, list=0x8859bf5fd20,
    trace_arg=0x7f7ffffdfd68) at vm_trace.c:325
325             if (!(hook->hook_flags & RUBY_EVENT_HOOK_FLAG_DELETED) &&
(gdb) bt
#0  0x00000885cb83e9eb in exec_hooks_body (ec=0x884de217650, list=0x8859bf5fd20,
```

```
    trace_arg=0x7f7ffffdfd68) at vm_trace.c:325
#1  0x00000885cb839f60 in exec_hooks_protected (ec=0x884de217650, list=0x8859bf5fd20,
    trace_arg=0x7f7ffffdfd68) at vm_trace.c:380
#2  0x00000885cb839cb7 in rb_exec_event_hooks (trace_arg=0x7f7ffffdfd68, hooks=0x8859bf5fd20,
    pop_p=0) at vm_trace.c:424
#3  0x00000885cb7fff11 in rb_exec_event_hook_orig (ec=0x884de217650, hooks=0x8859bf5fd20,
    flag=16, self=9368423812640, id=0, called_id=0, klass=0, data=85, pop_p=0)
    at ./vm_core.h:2030
#4  0x00000885cb826d8a in vm_trace_hook (ec=0x884de217650, reg_cfp=0x8854a7fdf40,
    pc=0x8859bf78330, pc_events=16, target_event=532, global_hooks=0x884de24f410,
    local_hooks=0x8859bf5fd20, val=85) at ./vm_insnhelper.c:5646
#5  0x00000885cb822dd4 in vm_trace (ec=0x884de217650, reg_cfp=0x8854a7fdf40)
    at ./vm_insnhelper.c:5749
#6  0x00000885cb7f0cec in vm_exec_core (ec=0x884de217650, initial=0) at vm.inc:5100
#7  0x00000885cb803ac3 in rb_vm_exec (ec=0x884de217650, mjit_enable_p=true) at vm.c:2287
#8  0x00000885cb804a42 in rb_iseq_eval_main (iseq=0x884e4ec4b60) at vm.c:2546
#9  0x00000885cb57f9e6 in rb_ec_exec_node (ec=0x884de217650, n=0x884e4ec4b60) at eval.c:280
#10 0x00000885cb57f8a1 in ruby_run_node (n=0x884e4ec4b60) at eval.c:321
#11 0x00000882cf893db0 in rb_main (argc=3, argv=0x7f7ffffe18d8) at ./main.c:47
#12 0x00000882cf893dff in main (argc=3, argv=0x7f7ffffe18d8) at ./main.c:56
(gdb) p hook
$1 = (rb_event_hook_t *) 0xdfdfdfdfdfdfdfdf
```

0xdfdfdfdfdfdfdfdf is the pattern OpenBSD's memory allocator uses for freed data, so this is almost definitely a use-after-free bug.

I was able to simplify the reproducer:

```
def bar
  42 #bp here
end

tp_line = TracePoint.new(:line) do |tp0|
  tp_multi1 = TracePoint.new(:return, :b_return, :line) do |tp|
    tp0.disable
  end
  tp_multi1.enable
end

# Removing the target for this enable call fixes the segfault
tp_line.enable(target: RubyVM::InstructionSequence.of(method :bar))

bar
```

#3 - 04/27/2022 06:36 AM - hurricup (Alexandr Evstigneev)

jeremyevans0 (Jeremy Evans) wrote in #note-2:

> You are adding and enabling a separate return event tracepoint (tp_multi2) on the same method before the method returns, while it is still in the process of handling the first return event tracepoint (tp_multi1). I don't think it is unexpected that this tracepoint (tp_multi2) would also be called. Note that if you add a tp_multi2.disable call directly after the tp_multi2.enable call, you don't get the second return event printed.

My points are:

1. This is inconsistent with pre 2.6 behavior, when tp set inside the tp handler was not triggered by the same event.
2. Other TPs should not affect this behavior, it should be consistent.

I'm not saying that it is definitely wrong to trigger such TP, but it feels wrong. Because VM already started processing :return event and invoking hooks. And not sure that it should invoke handlers set after processing started. But this is my personal feeling, nothing objective. Still, may be such behavior is an opportunity to implement some tricky things, hard to say, but need to be consistent for sure, one way or another.

#4 - 04/28/2022 10:37 PM - jeremyevans0 (Jeremy Evans)

I've submitted a pull request to fix the use-after-free bug: https://github.com/ruby/ruby/pull/5862

#5 - 04/29/2022 11:05 AM - hurricup (Alexandr Evstigneev)

jeremyevans0 (Jeremy Evans) wrote in #note-4:

> I've submitted a pull request to fix the use-after-free bug: https://github.com/ruby/ruby/pull/5862

I can confirm that with this patch, behavior is consistent and do not depend on tp0 state. Return event is always handled by both handlers.
But it still feels strange and inconsistent.
E.g. this code should stuck in infinite loop, but it passes 2 times and that's it:

```
def bar
  42 #bp here
end

def set_return_tp
  TracePoint.new(:return) do |tp|
    puts 'Return hit'
    tp.disable
    set_return_tp
  end.enable
end

set_return_tp
bar
```

Or:

```
def bar
  42 #bp here
end

def set_return_tp1
  TracePoint.new(:return) do |tp|
    puts 'Return hit 1'
    tp.disable
    set_return_tp2
  end.enable
end

def set_return_tp2
  TracePoint.new(:return) do |tp|
    puts 'Return hit 2'
    tp.disable
    set_return_tp1
  end.enable(target: RubyVM::InstructionSequence.of(method :bar))
end

set_return_tp1
#set_return_tp2
bar
```

This hits twice.

And

```
def bar
  42 #bp here
end

def set_return_tp1
  TracePoint.new(:return) do |tp|
    puts 'Return hit 1'
    tp.disable
    set_return_tp2
  end.enable
end

def set_return_tp2
  TracePoint.new(:return) do |tp|
    puts 'Return hit 2'
    tp.disable
    set_return_tp1
  end.enable(target: RubyVM::InstructionSequence.of(method :bar))
end

#set_return_tp1
set_return_tp2
bar
```

This hits once.

So something is definitely wrong there.

#6 - 04/30/2022 12:50 AM - alanwu (Alan Wu)

So having just read the code, I understand why you are seeing this behavior. We run global handlers before local handlers, so if you have a global handler G and a local handler L, and G enables L, they both run. It is weird because this seems to be the only situation where two handlers interact.

Here is a script to show all 4 possible choices for global versus local for the two interacting handlers:

```
(0..0b11).each do |mode|
  Class.new do
    def foo
      1
    end
    me = instance_method(:foo)

    one_enable = mode[0] == 1 ? { target: me } : {}
    two_enable = mode[1] == 1 ? { target: me } : {}
    one_ran = false
    two_ran = false

    two = TracePoint.new(:return) do
      two_ran = true
      two.disable
    end

    one = TracePoint.new(:return) do
      one_ran = true
      two.enable(**two_enable)
      one.disable
    end

    one.enable(**one_enable)

    new.foo

    p [mode, one_ran, two_ran]
  end
end
```

The output has evolved over time:

```
ruby 2.6.6p146 (2020-03-31 revision 67876) [x86_64-darwin19]
[0, true, true]
[1, true, false]
[2, true, true]
[3, true, false]
ruby 2.7.3p183 (2021-04-05 revision 6847ee089d) [x86_64-darwin19]
[0, true, false]
[1, true, false]
[2, true, false]
[3, true, false]
ruby 3.0.0p0 (2020-12-25 revision 95aff21468) [x86_64-darwin19]
[0, true, false]
[1, true, false]
[2, true, true]
[3, true, false]
# 3.1 same as 3.0
```

I suppose 2.7 behavior is the most consistent.

#7 - 04/30/2022 05:32 AM - hurricup (Alexandr Evstigneev)

alanwu (Alan Wu) wrote in #note-6:

> So having just read the code, I understand why you are seeing this behavior. We run global handlers before local handlers, so if you have a global handler G and a local handler L, and G enables L, they both run. It is weird because this seems to be the only situation where two handlers interact.

Aren't both handlers same (global) in my first example?

```
def bar
  42 #bp here
end
```

```
def set_return_tp
  TracePoint.new(:return) do |tp|
    puts 'Return hit'
    tp.disable
    set_return_tp
  end.enable
end

set_return_tp
bar
```

#8 - 05/02/2022 06:33 PM - alanwu (Alan Wu)

> Aren't both handlers same (global) in my first example?

They are, but the two invocations are coming from two separate events while my script is concerned with handling within one event. You can see this from a slightly modified version of your script:

```
def bar
  42 #bp here
end

def set_return_tp
  TracePoint.new(:return) do |tp|
    puts "Return hit #{tp.method_id}"
    tp.disable
    set_return_tp
  end.enable
end

set_return_tp
p :gap
bar

Return hit set_return_tp
:gap
Return hit bar
```

#9 - 05/30/2022 10:39 PM - alanwu (Alan Wu)

- Status changed from Open to Closed

Applied in changeset git|a6877562841878878 35aa345adc89b2718054e4a.

Fix use-after-free with interacting TracePoints

vm_trace_hook() runs global hooks before running local hooks. Previously, we read the local hook list before running the global hooks which led to use-after-free when a global hook frees the local hook list. A global hook can do this by disabling a local TracePoint, for example.

Delay local hook list loading until after running the global hooks.

Issue discovered by Jeremy Evans in GH-5862.

[Bug #18730]

#10 - 06/01/2022 04:36 PM - alanwu (Alan Wu)

Here's some more information to round out this thread. I was a bit sloppy in ruby-core:108449 and the output I posted was misleading. The script was observing return events from tp.enable. Tracing is tricky!
Here is an updated version of the script:

```
# Script intended to observe handling of TracePoint hooks
# within one firing of a particular event.
(0..0b11).each do |mode|
```

```
  Class.new do
    def foo
      1
    end
    me = instance_method(:foo)

    # Unrelated targeting TracePoint. Its presence changes the output of mode 2
    # (global enables local) before a687756284187887835aa345adc89b2718054e4a.
    if false
      tp = TracePoint.new(:return) {}
      tp.enable(target: me)
    end

    one_enable = mode[0] == 1 ? { target: me } : {}
    two_enable = mode[1] == 1 ? { target: me } : {}
    one_ran = 0
    two_ran = 0

    two = TracePoint.new(:return) do |tp|
      next p tp unless tp.method_id == :foo
      two_ran += 1
      tp.disable
    end

    one = TracePoint.new(:return) do |tp|
      next p tp unless tp.method_id == :foo
      one_ran += 1
      two.enable(**two_enable)
      tp.disable
    end

    one.enable(**one_enable) do
      new.foo
    end

    p [mode, one_ran, two_ran]
  ensure
    two&.disable
  end
end

=begin
ruby 2.6.6p146 (2020-03-31 revision 67876) [x86_64-darwin19]
#<TracePoint:return `enable'@<internal:prelude>:138>
[0, 1, 0]
#<TracePoint:return `enable'@<internal:prelude>:138>
[1, 1, 0]
[2, 1, 0]
[3, 1, 0]

ruby 2.7.0p0 (2019-12-25 revision 647ee6f091) [x86_64-darwin19]
[0, 1, 0]
[1, 1, 0]
[2, 1, 0]
[3, 1, 0]

ruby 3.0.0p0 (2020-12-25 revision 95aff21468) [x86_64-darwin19]
[0, 1, 0]
#<TracePoint:return `enable' <internal:trace_point>:197>
[1, 1, 0]
[2, 1, 0]
[3, 1, 0]

ruby 3.1.0p0 (2021-12-25 revision fb4df44d16) [x86_64-darwin20]
[0, 1, 0]
#<TracePoint:return `enable' <internal:trace_point>:213>
[1, 1, 0]
[2, 1, 0]
[3, 1, 0]
=end
```

(the inconsistent prints *might* be coming from the
tracing setup for opt_invokebuiltin_delegate_leave)