Ruby - Feature #6783

Infinite loop in inspect, not overriding inspect, to_s, and no known circular references. Stepping into inspect in debugger locks it up with 100% CPU.

07/23/2012 11:22 PM - garysweaver (Gary Weaver)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

In Ruby 1.9.3p194 in Rails 3.2.6 in rails console, in my script I'm calling inspect on a Ruby object and even though I'm not overriding inspect, to_s, and there are no known circular references, inspect is going into an infinite loop locking Ruby up with 100% CPU usage.

At first, I would think this problem is probably outside of Ruby and either in my code or in a gem that I'm using, however the problem is that using the Debugger gem, if I set a breakpoint above the issue and use "s" (by itself) to step into the line where it calls inspect, it locks up there, so I can't debug the issue. When I do that I hit ctrl-c, I'm in .../.rvm/rubies/ruby-1.9.3-p194/lib/ruby/1.9.1/irb.rb:

```
   64      trap("SIGINT") do
=> 65        irb.signal_handle
   66      end
```

and breaking out of that, or if I don't step into it and I break out of it, I see:

```
path_to_script/script_name.rb:739:in `call'
path_to_script/script_name.rb:739:in `inspect'
path_to_script/script_name.rb:739:in `inspect'
...
    (~100 times)
    path_to_script/script_name.rb:739:in `block (2 levels) in my_method_name'
```

In a situation like this, how can I debug the issue? Is there anything in the inspect method that could causing this behavior?

I think the most likely culprit is some bad code on my part in the script, but unfortunately I can't debug it when the debugger can't step into inspect.

Thanks for any help you can provide.

Related issues:

Related to Ruby - Feature #6733: New inspect framework	**Open**	
Related to Ruby - Feature #18285: NoMethodError#message uses a lot of CPU/is ...	**Closed**	
Is duplicate of Ruby - Bug #6291: Backtrace printout halts for an extremely l...	**Closed**	**04/14/2012**

History

#1 - 07/23/2012 11:29 PM - garysweaver (Gary Weaver)

Note: unfortunately this is an internal script that depends on a local DB so can't share, but can duplicate the problem each time, and was able to duplicate in ruby-1.9.3-p125 also. I don't have ruby-head setup yet, but working on it.

#2 - 07/24/2012 04:27 AM - garysweaver (Gary Weaver)

Figured it out. The problem was that the inspect was just generating a lot of data.

It really wasn't taking 100% CPU. The process in Activity Monitor in OS X appeared as though it was taking 100%, but I could also see in Activity Monitor that only 12-15% CPU was actually being used.

Setting the breakpoint/stepping into inspect is of little use, but that is because inspect is C rather than Ruby code, so that can't be stepped into with the Ruby debugger gem.

Here is an example of something with somewhat similar behavior to what I was describing. As you can see, the issue was that the structure was just getting larger and larger, but not because of circular references:

```
class Thing
  attr_accessor :a_hash
```

```
  attr_accessor :an_array
  attr_accessor :another_array
end

class Reporter
  def report(num)
    t1 = Time.now
    first_ref = Thing.new
    this_ref = first_ref
    num.times do
      next_ref = Thing.new
      this_ref.a_hash = {"a#{rand(9999999)}".to_sym => next_ref}
      this_ref.an_array = [next_ref]
      this_ref.another_array = [next_ref]
      this_ref = next_ref
    end
    this_ref.a_hash = {"a#{rand(9999999)}".to_sym => first_ref}
    this_ref.an_array = [first_ref]
    this_ref.another_array = [first_ref]

    #breakpoint
    puts first_ref.inspect
    puts "#{num} times took #{Time.now - t1}ms"
  end
end

r = Reporter.new
20.times {|i| r.report(i)}
```

But if you add more and more to a linked object chain where the last item in the chain self-references the first, you can see that self-reference and the depth of self-reference are not problems here:

```
class Thing
  attr_accessor :ref
end

class Reporter
  def report(num)
    t1 = Time.now
    first_ref = Thing.new
    this_ref = first_ref
    num.times do
      next_ref = Thing.new
      this_ref.ref = next_ref
      this_ref = next_ref
    end
    this_ref.ref = first_ref

    #breakpoint
    puts first_ref.inspect
    puts "#{num} times took #{Time.now - t1}ms"
  end
end

r = Reporter.new
1000.times {|i| r.report(i)}
```

Please close this ticket, and sorry about that. Wanted to explain what happened in case it helps anyone else.

#3 - 11/03/2012 12:06 PM - mame (Yusuke Endoh)

- Status changed from Open to Closed

#4 - 08/17/2016 01:15 PM - stefan.kroes (Stefan Kroes)

I would like to reopen discussion on this subject. I think the default implementation of #inspect tends to hang/explode for complex/large object graphs with lots of cycles. In 10 years of programming Ruby I ran into this twice and had to waste several hours before finding the problem twice (today and several years ago if I remember correctly). Inspect is often used for generating error messages which will hang in turn, misdirecting debugging efforts to the original error.

To clarify: I don't really think this is a bug, just an aspect of Ruby that may cause grief (especially to new users) and can be improved.

A simple script to reproduce:

```
class Base
```

```
  attr_accessor :foos, :bars, :bazs
end

class Foo < Base; end
class Bar < Base; end
class Baz < Base; end

foos = Array.new(100) { Foo.new }
bars = Array.new(100) { Bar.new }
bazs = Array.new(100) { Baz.new }

[*foos, *bars, *bazs].each do |base|
  base.foos = foos
  base.bars = bars
  base.bazs = bazs
end

puts foos.inspect.size

{14:54}[2.3.1]~  time ruby test.rb
127165300
ruby test.rb  7.77s user 0.29s system 97% cpu 8.237 total
```

This example seems somewhat contrived and 7 seconds doesn't seem long but I just had a real-life object graph of a large state machine hang my process for at least 20 minutes (broke it off).

I know the documentation for Object#inspect says User defined classes should override this method to make better representation of obj. but I don't think many people do this, especially as the default implementation is very useful.

Possible solutions include:

- Further limiting recursion of default inspect
- Limiting number of elements shown in inspect for Hash, Array, etc.
- Putting a timeout on the default inspect, informing the user he/she should override inspect with something sensible for a certain class
  - Timeout should be nested so it triggers for the deepest inspect that takes too long

#5 - 09/26/2016 11:07 AM - shyouhei (Shyouhei Urabe)

- Status changed from Closed to Open

reopen as per request (seems like it is no longer a bug ticket but a feature request?)

#6 - 09/26/2016 03:02 PM - headius (Charles Nutter)

See also #9725, my issue/request relating to NameError's behavior of carrying the target object and causing huge inspect-driven memory bloat when attempting to print the message.

#7 - 10/11/2016 07:41 AM - shyouhei (Shyouhei Urabe)

- Related to Feature #6733: New inspect framework added

#8 - 10/11/2016 07:43 AM - shyouhei (Shyouhei Urabe)

- Is duplicate of Bug #6291: Backtrace printout halts for an extremely long time when large amounts of data are allocated added

#9 - 10/11/2016 10:58 AM - shyouhei (Shyouhei Urabe)

We looked at this ticket at developer meeting today and found several former tickets that was linked then. FYI #6733 is the most big-pictured feature request that ultimately solves this problem. Not yet implemented though.

#10 - 08/05/2019 11:13 PM - jeremyevans0 (Jeremy Evans)

- Tracker changed from Bug to Feature

- ruby -v deleted (ruby 1.9.3p194 (2012-04-20 revision 35410) [x86_64-darwin11.4.0])

- Backport deleted (2.1: UNKNOWN, 2.2: UNKNOWN, 2.3: UNKNOWN)

#11 - 11/19/2021 08:00 PM - Eregon (Benoit Daloze)

- Related to Feature #18285: NoMethodError#message uses a lot of CPU/is really expensive to call added