Ruby trunk - Bug #5437

Using fibers leads to huge memory leak

10/12/2011 10:06 PM - rupert (Robert Pankowecki)

Status:	Rejected		
Priority:	Normal		
Assignee:	ko1 (Koichi Sasada)		
Target version:			
ruby -v:	ruby 1.9.3dev (2011-10-11 revision 33457) [x86_64-linux]	**Backport:**	

Description

It appears to me that there is something wrong with reallocating (reusing?) memory used by fibers. Here is a little script:

```
require 'fiber'

fibers = 10_000.times.map do
Fiber.new do
f = Fiber.current
Fiber.yield
end
end; nil

fibers.each(&:resume); nil
fibers.each(&:resume); nil
fibers = []

GC.start
```

Running this code in IRB multiple times leads to consuming more and more memory every time. However every time I execute this code the amount of newly consumed memory is lower. I was able to repeat this bug with ruby 1.9.2.p290 and 1.9.3-head using 32 and 64 bit architecture.

Here are some memory stats:
a) 32bit
Iter | VSZ | RSS
1 186268 63508
2 360196 119580
3 360340 147600
4 360484 178552
5 360616 199968
6 360612 210716
7 360612 221920
8 360612 224544
9 360612 229976

b) 64bit
1 395152 107600
2 739924 176532
3 739924 187972
4 739924 203300
5 739924 217716
6 739924 218344
7 739924 218540
8 739924 234040
9 739924 234256

We can see on our production instance that the more Fibers are in use the more memory is never reclaimed back to OS and the bigger the leak is.

History

#1 - 10/13/2011 05:23 AM - normalperson (Eric Wong)

Robert Pankowecki robert.pankowecki@gmail.com wrote:

> ruby -v: ruby 1.9.3dev (2011-10-11 revision 33457) [x86_64-linux]
>
> fibers = []
>
> GC.start

GC.start is never guaranteed to release memory (especially not with lazy sweep). Even if GC.start always releases memory (via free(3), malloc(3)/free(3) implementations are not guaranteed to release memory back to the kernel.

> We can see on our production instance that the more Fibers are in use the more memory is never reclaimed back to OS and the bigger the leak is.

Since you're on Linux, I assume you're using glibc/eglibc and hitting this problem with its malloc. The comment starting with "Update in 2006" in malloc/malloc.c (git://sourceware.org/git/glibc.git) can give you an explanation of why memory isn't always given back to the kernel.

To workaround this glibc malloc issue, you can try setting MALLOC_MMAP_THRESHOLD_=131072 (or another suitable value) in the env and possibly trade performance for less memory use.

#2 - 10/17/2011 05:50 PM - rupert (Robert Pankowecki)

- File cieknace_fibery added

- File 65536.txt added

- File 131072.txt added

- File 262144.txt added

I tried to use your setting but as you can see in the results files it does not change very much. Also I added finalizers to those fibers to make sure that they are garbage collected and most of them are. So why is the memory increasing when new fibers can use the memory allocated by the old, unused, garbagecollected fibers ? Am I misunderstanding something?

#3 - 03/11/2012 04:17 PM - ko1 (Koichi Sasada)

- Category set to core

- Assignee set to ko1 (Koichi Sasada)

I'll review it later. however, i think it is libc (malloc) issue...

#4 - 03/18/2012 06:46 PM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Assigned

#5 - 08/08/2012 07:54 PM - rupert (Robert Pankowecki)

Any progress ? Looks like this is still a problem in Ruby2.0: https://gist.github.com/bf4b57d5bf419dbf56ae

#6 - 08/09/2012 06:23 AM - normalperson (Eric Wong)

"rupert (Robert Pankowecki)" robert.pankowecki@gmail.com wrote:

> ## Any progress ? Looks like this is still a problem in Ruby2.0: https://gist.github.com/bf4b57d5bf419dbf56ae
>
> Bug #5437: Using fibers leads to huge memory leak
> https://bugs.ruby-lang.org/issues/5437#change-28726

Try increasing your outer loop from 10.times to 100.times, or 1000.times, etc... You should see memory usage stabilize with a bigger loop.

Returning memory back to the OS is expensive (if the app is likely to

reuse it), so Ruby probably wants to hold onto some internal structures
(as does glibc malloc).

#7 - 08/09/2012 08:38 PM - rupert (Robert Pankowecki)

Yes the memory usage is stable but that does not change the fact that it is simply huge and never returned back. In real applications this leads to memory usage always as big as in the moment of biggest peak of used fibers. Even after few hours of using 10% of Fibers that were necessary in the peak. So for me, never reclaiming memory back to OS is a leak.

#8 - 08/10/2012 09:23 AM - drbrain (Eric Hodel)

- Status changed from Assigned to Rejected

=begin
Your test does not illustrate any leak and is likely not capable of demonstrating a leak in a diagnosable way.

You seem to have some misconceptions about how malloc() and free() work and how ruby's GC works.

Finalizers are not invoked immediately after object collection. They are invoked at some point in the future so more Fiber objects may be collected than indicated by your counter.

Calling GC.start is not guaranteed to collect all objects without visible references. Ruby's GC is conservative and walks the stack looking for possible references to objects, even if the stack value only coincidentally points to an object.

Ruby calls malloc() to allocate memory. If malloc() does not have any free memory available to assign to ruby it will ask for more from the kernel via mmap() (and sometimes sbrk(), but less frequently on modern OSs). When ruby is done with memory it wall call free(). This will tell malloc() the memory is no longer being used and is available for a future call to malloc(). Sometimes free() will result in returning memory to the OS, but not always since that reduces performance. (malloc() may immediately need to ask the OS for more memory.)

You should not always expect to see a reduction of the resident size of a process when Ruby's GC runs, especially on a very small script that runs for a very short period.

To demonstrate a leak you need to demonstrate process growth in a long-running program. Running for a few seconds and checking the size of ((%ps%)) is insufficient.

I made a better test for Fiber leaks:

require 'fiber'

loop do
fibers = 10.times.map do
n = Fiber.new do
f = Fiber.current
Fiber.yield
end
n
end; nil

```
fibers.each(&:resume); nil
fibers = []
```

end

This allocates a small number of fibers and runs continuously. If Ruby is leaking fibers we should see a small but steady growth in process size over time. We should also be able to run tools on the process like OS X's leaks(1) to detect the leak.

I ran my test for over 25 minutes. The memory size reached 39.6MB and remained constant for the duration of the test.

leaks(1) reported:

Process: ruby20 [96158]
Path: /Users/drbrain/Work/svn/ruby/trunk/ruby20
Load Address: 0x104616000
Identifier: ruby20
Version: ??? (???)
Code Type: X86-64 (Native)
Parent Process: make [96143]

Date/Time: 2012-08-09 17:18:55.562 -0700
OS Version: Mac OS X 10.8 (12A269)
Report Version: 7

leaks Report Version: 2.0
Process 96158: 16276 nodes malloced for 8215 KB
Process 96158: 0 leaks for 0 total leaked bytes.

As you can see there is no leak detected. Also, note that only about 20% of the resident memory capacity is part of an active allocation.

Due to these results I am reasonably certain that no leak has occurred.

We will reopen if you can demonstrate a leak in a long-running process.

=end

Files

cieknace_fibery	510 Bytes	10/17/2011	rupert (Robert Pankowecki)
65536.txt	1.61 KB	10/17/2011	rupert (Robert Pankowecki)
131072.txt	1.61 KB	10/17/2011	rupert (Robert Pankowecki)
262144.txt	1.61 KB	10/17/2011	rupert (Robert Pankowecki)