Ruby master - Bug #11164

Garbage collector in Ruby 2.2 provokes unexpected CoW

05/20/2015 03:06 PM - tkalmus (Thomas Kalmus)

Status:	Assigned		
Priority:	Normal		
Assignee:	authorNari (Narihiro Nakamura)		
Target version:			
ruby -v:	2.2.2	**Backport:**	

Description

How do I prevent the GC from provoking copy-on-write, when I fork my process ? I have recently been analyzing the garbage collector's behavior in Ruby, due to some memory issues that I encountered in my program (I run out of memory on my 60core 0.5Tb machine even for fairly small tasks). For me this really limits the usefulness of ruby for running programs on multicore servers. I would like to present my experiments and results here.

The issue arises when the garbage collector runs during forking. I have investigated three cases that illustrate the issue.

Case 1: We allocate a lot of objects (strings no longer than 20 bytes) in the memory using an array. The strings are created using a random number and string formatting. When the process forks and we force the GC to run in the child, all the shared memory goes private, causing a duplication of the initial memory.

Case 2: We allocate a lot of objects (strings) in the memory using an array, but the string is created using the rand.to_s function, hence we remove the formatting of the data compared to the previous case. We end up with a smaller amount of memory being used, presumably due to less garbage. When the process forks and we force the GC to run in the child, only part of the memory goes private. We have a duplication of the initial memory, but to a smaller extent.

Case 3: We allocate fewer objects compared to before, but the objects are bigger, such that the amount of memory allocated stays the same as in the previous cases. When the process forks and we force the GC to run in the child all the memory stays shared, i.e. no memory duplication.

Here I paste the Ruby code that has been used for these experiments. To switch between cases you only need to change the “option” value in the memory_object function. The code was tested using Ruby 2.2.2, 2.2.1, 2.1.3, 2.1.5 and 1.9.3 on an Ubuntu 14.04 machine.

Sample output for case 1:

```
ruby version 2.2.2
 proces   pid log                    priv_dirty   shared_dirty
 Parent  3897 post alloc                    38            0
 Parent  3897 4 fork                         0           37
 Child   3937 4 initial                      0           37
 Child   3937 8 empty GC                    35            5
```

The exact same code has been written in Python and in all cases the CoW works perfectly fine.

Sample output for case 1:

```
python version 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2]
 proces   pid log                    priv_dirty shared_dirty
 Parent  4308 post alloc                    35            0
 Parent  4308 4 fork                         0           35
 Child   4309 4 initial                      0           35
 Child   4309 10 empty GC                    1           34
```

Ruby code

```
$start_time=Time.new

# Monitor use of Resident and Virtual memory.
class Memory
```

```
    shared_dirty = '.+?Shared_Dirty:\s+(\d+)'
    priv_dirty = '.+?Private_Dirty:\s+(\d+)'
    MEM_REGEXP = /#{shared_dirty}#{priv_dirty}/m

    # get memory usage
    def self.get_memory_map( pids)
        memory_map = {}
        memory_map[ :pids_found] = {}
        memory_map[ :shared_dirty] = 0
        memory_map[ :priv_dirty] = 0

        pids.each do |pid|
            begin
                lines = nil
                lines = File.read( "/proc/#{pid}/smaps")
            rescue
                lines = nil
            end
            if lines
                lines.scan(MEM_REGEXP) do |shared_dirty, priv_dirty|
                    memory_map[ :pids_found][pid] = true
                    memory_map[ :shared_dirty] += shared_dirty.to_i
                    memory_map[ :priv_dirty] += priv_dirty.to_i
                end
            end
        end
        memory_map[ :pids_found] = memory_map[ :pids_found].keys
        return memory_map
    end

    # get the processes and get the value of the memory usage
    def self.memory_usage( )
        pids   = [ $$]
        result = self.get_memory_map( pids)

        result[ :pids]   = pids
        return result
    end

    # print the values of the private and shared memories
    def self.log( process_name='', log_tag="")
        if process_name == "header"
            puts " %-6s %5s %-12s %10s %10s\n" % ["proces", "pid", "log", "priv_dirty", "shared_di
rty"]
        else
            time = Time.new - $start_time
            mem = Memory.memory_usage( )
            puts " %-6s %5d %-12s %10d %10d\n" % [process_name, $$, log_tag, mem[:priv_dirty]/1000
, mem[:shared_dirty]/1000]
        end
    end
end

# function to delay the processes a bit
def time_step( n)
    while Time.new - $start_time < n
        sleep( 0.01)
    end
end

# create an object of specified size. The option argument can be changed from 0 to 2 to visualize
the behavior of the GC in various cases
#
# case 0 (default) : we make a huge array of small objects by formatting a string
# case 1 : we make a huge array of small objects without formatting a string (we use the to_s func
tion)
# case 2 : we make a smaller array of big objects
```

```
def memory_object( size, option=1)
    result = []
    count = size/20

    if option > 3 or option < 1
        count.times do
            result << "%20.18f" % rand
        end
    elsif option == 1
        count.times do
            result << rand.to_s
        end
    elsif option == 2
        count = count/10
        count.times do
            result << ("%20.18f" % rand)*30
        end
    end

    return result
end

##### main #####

puts "ruby version #{RUBY_VERSION}"

GC.disable

# print the column headers and first line
Memory.log( "header")

# Allocation of memory
big_memory = memory_object( 1000 * 1000 * 10)

Memory.log( "Parent", "post alloc")

lab_time = Time.new - $start_time
if lab_time < 3.9
    lab_time = 0
end

# start the forking
pid = fork do
    time = 4
    time_step( time + lab_time)
    Memory.log( "Child", "#{time} initial")

    # force GC when nothing happened
    GC.enable; GC.start; GC.disable

    time = 8
    time_step( time + lab_time)
    Memory.log( "Child", "#{time} empty GC")

    sleep( 1)
    STDOUT.flush
    exit!
end

time = 4
time_step( time + lab_time)
Memory.log( "Parent", "#{time} fork")

# wait for the child to finish
Process.wait( pid)
```

Python code

```
import re
import time
import os
import random
import sys
import gc

start_time=time.time()

# Monitor use of Resident and Virtual memory.
class Memory:

    def __init__(self):
        self.shared_dirty = '.+?Shared_Dirty:\s+(\d+)'
        self.priv_dirty = '.+?Private_Dirty:\s+(\d+)'
        self.MEM_REGEXP = re.compile("{shared_dirty}{priv_dirty}".format(shared_dirty=self.shared_
dirty, priv_dirty=self.priv_dirty), re.DOTALL)

    # get memory usage
    def get_memory_map(self, pids):
        memory_map = {}
        memory_map[ "pids_found" ] = {}
        memory_map[ "shared_dirty" ] = 0
        memory_map[ "priv_dirty" ] = 0

        for pid in pids:
            try:
                lines = None

                with open( "/proc/{pid}/smaps".format(pid=pid), "r" ) as infile:
                    lines = infile.read()
            except:
                lines = None

            if lines:
                for shared_dirty, priv_dirty in re.findall( self.MEM_REGEXP, lines ):
                    memory_map[ "pids_found" ][pid] = True
                    memory_map[ "shared_dirty" ] += int( shared_dirty )
                    memory_map[ "priv_dirty" ] += int( priv_dirty )

        memory_map[ "pids_found" ] = memory_map[ "pids_found" ].keys()
        return memory_map

    # get the processes and get the value of the memory usage
    def memory_usage( self):
        pids   = [ os.getpid() ]
        result = self.get_memory_map( pids)

        result[ "pids" ]   = pids

        return result

    # print the values of the private and shared memories
    def log( self, process_name='', log_tag=""):
        if process_name == "header":
            print " %-6s %5s %-12s %10s %10s" % ("proces", "pid", "log", "priv_dirty", "shared_dir
ty")
        else:
            global start_time
            Time = time.time() - start_time
            mem = self.memory_usage( )
            print " %-6s %5d %-12s %10d %10d" % (process_name, os.getpid(), log_tag, mem["priv_dir
ty"]/1000, mem["shared_dirty"]/1000)

# function to delay the processes a bit
def time_step( n):
    global start_time
```

```
    while (time.time() - start_time) < n:
        time.sleep( 0.01)

# create an object of specified size. The option argument can be changed from 0 to 2 to visualize
the behavior of the GC in various cases
#
# case 0 (default) : we make a huge array of small objects by formatting a string
# case 1 : we make a huge array of small objects without formatting a string (we use the to_s func
tion)
# case 2 : we make a smaller array of big objects
def memory_object( size, option=2):
    count = size/20

    if option > 3 or option < 1:
        result = [ "%20.18f"% random.random() for i in xrange(count) ]

    elif option == 1:
        result = [ str( random.random() ) for i in xrange(count) ]

    elif option == 2:
        count = count/10
        result = [ ("%20.18f"% random.random())*30 for i in xrange(count) ]

    return result

##### main #####

print "python version {version}".format(version=sys.version)

memory = Memory()

gc.disable()

# print the column headers and first line
memory.log( "header")   # Print the headers of the columns

# Allocation of memory
big_memory = memory_object( 1000 * 1000 * 10)   # Allocate memory

memory.log( "Parent", "post alloc")

lab_time = time.time() - start_time
if lab_time < 3.9:
    lab_time = 0

# start the forking
pid = os.fork()     # fork the process
if pid == 0:
    Time = 4
    time_step( Time + lab_time)
    memory.log( "Child", "{time} initial".format(time=Time))

    # force GC when nothing happened
    gc.enable(); gc.collect(); gc.disable();

    Time = 10
    time_step( Time + lab_time)
    memory.log( "Child", "{time} empty GC".format(time=Time))

    time.sleep( 1)

    sys.exit(0)

Time = 4
time_step( Time + lab_time)
memory.log( "Parent", "{time} fork".format(time=Time))
```

```
# Wait for child process to finish
os.waitpid( pid, 0)
```

History

#1 - 05/22/2015 07:09 AM - tkalmus (Thomas Kalmus)

- File memory_test_min.py added

- File memory_test_min.rb added

- Assignee set to usa (Usaku NAKAMURA)

#2 - 05/22/2015 07:10 AM - tkalmus (Thomas Kalmus)

- File deleted (memory_test_min.py)

#3 - 05/22/2015 07:10 AM - tkalmus (Thomas Kalmus)

- File deleted (memory_test_min.rb)

#4 - 05/22/2015 07:55 AM - usa (Usaku NAKAMURA)

- Status changed from Open to Assigned

- Assignee changed from usa (Usaku NAKAMURA) to authorNari (Narihiro Nakamura)

- Priority changed from 5 to Normal

#5 - 05/22/2015 11:17 AM - tkalmus (Thomas Kalmus)

- Assignee changed from authorNari (Narihiro Nakamura) to usa (Usaku NAKAMURA)

- Priority changed from Normal to 5

Calling the GC several times before forking the process solves the issue and I am quite surprised. I have also run the code using Ruby 2.0.0 and the issue doesn't even appear, so it must be related to this generational GC.
However, if I call the memory_object function without assigning the output to any variables (I am only creating garbage), then the memory is duplicated. The amount of memory that is copied depends on the amount of garbage that I create - the more garbage, the more memory becomes private.

Any ideas how I can prevent this ?

Here are some results

Running the GC in 2.0.0

```
ruby version 2.0.0
 proces   pid log          priv_dirty shared_dirty
 Parent  3664 post alloc           67          0
 Parent  3664 4 fork                1         69
 Child   3700 4 initial             1         69
 Child   3700 8 empty GC            6         65
```

Calling memory_object(1000*1000) in the child

```
ruby version 2.0.0
 proces   pid log          priv_dirty shared_dirty
 Parent  3703 post alloc           67          0
 Parent  3703 4 fork                1         70
 Child   3739 4 initial             1         70
 Child   3739 8 empty GC           15         56
```

Calling memory_object(1000*1000*10)

```
ruby version 2.0.0
 proces   pid log          priv_dirty shared_dirty
 Parent  3743 post alloc           67          0
 Parent  3743 4 fork                1         69
 Child   3779 4 initial             1         69
 Child   3779 8 empty GC           89          5
```

#6 - 05/22/2015 11:45 AM - usa (Usaku NAKAMURA)

- Assignee changed from usa (Usaku NAKAMURA) to authorNari (Narihiro Nakamura)

- Priority changed from 5 to Normal

Priority field and Assignee field is for use of ruby-core team, not for reporters.
DON'T TOUCH THEM!

#7 - 05/26/2015 06:29 AM - tkalmus (Thomas Kalmus)

Here is a discussion about the source of the problem :
http://stackoverflow.com/questions/30353272/garbage-collector-in-ruby-2-2-provokes-unexpected-cow

Files

memory_test_min.rb	3.29 KB	05/20/2015	tkalmus (Thomas Kalmus)
memory_test_min.py	3.5 KB	05/20/2015	tkalmus (Thomas Kalmus)