Ruby - Feature #17176

GC.auto_compact / GC.auto_compact=(flag)

09/16/2020 06:39 PM - tenderlovemaking (Aaron Patterson)

Status:	Closed
Priority:	Normal
Assignee:	
Target version:	

Description

Hi,

I'd like to make compaction automatic eventually. As a first step, I would like to introduce two functions:

- GC.enable_autocompact
- GC.disable_autocompact

One function enables auto compaction, the other one disables it. Automatic compaction is *disabled* by default. When it is enabled it will happen only on every major GC.

I've made a pull request here: https://github.com/ruby/ruby/pull/3547

This patch makes *object movement* happen at the same time as page sweep. When one page finishes sweeping, that page is filled.

Sweep + Move Phase

During sweep, we keep a pointer to the current sweeping page. This pointer is kept in heap->sweeping_page. At the beginning of sweep, this is the *first* element of the heap's linked list.

At the same time, the compaction process points at the *last* page in the heap, and that is stored in heap->compact_cursor here.

Incremental sweeping sweeps one page at a time in the gc_page_sweep function. At the end of that function, we call gc_fill_swept_page. gc_fill_swept_page fills the page that was just swept and moves the movement cursor towards the sweeping cursor.

When the sweeping cursor and the movement cursor meet, sweeping is paused, and references are updated. This can happen in 2 ways, the sweeping cursor "runs in to the moving cursor" which is here. Or the moving cursor runs in to the sweep cursor which happens here.

Either way, the sweep step is paused and references are updated.

Reference Updating

Reference updating hasn't changed, but since reference updating happens before the GC finishes a cycle, it must take in to account garbage objects here.

Read Barrier

During the sweep phase, some objects may touch other objects. For example, T_CLASS must remove itself from a parent class.

```
class A; end
class B < A; end

const_set(:B, nil)
```

When B is freed, it must remove itself from A's subclasses. But what if A moved? To fix this, I've introduced a read barrier. The read barrier protects heap_page_body using mprotect. If something tries to read from the page, an exception will occur and we can move all objects back to the page (invalidate the movement).

The lock function is here.
The unlock function is here.

It uses sigaction to catch the exception here.

Cross Platform

mprotect and sigaction are not cross platform, they doesn't work on Windows. On Windows the read barrier uses exception handlers that are built in to Windows. I implemented them here.

The read barrier seems to work on all platforms we're testing.

Statistics

GC.stat(:compact_count) contains the number of times compaction has happened, so we can write things like this:

```
GC.enable_autocompact

cc = GC.stat(:compact_count)
list = []
loop do
  500.times { list << Object.new }
  break if cc < GC.stat(:compact_count)
end

p GC.stat(:compact_count)
```

We can check when the read barrier is triggered with GC.stat(:read_barrier_faults)

I've also added GC.latest_compact_info so you can see what types of objects moved and how many. For example:

```
[aaron@tc-lan-adapter ~/g/ruby (autocompact)]$ cat test.rb
list = []
500.times {
  list << Object.new
  Object.new
  Object.new
}

GC.enable_autocompact
count = GC.stat :compact_count
loop do
  list << Object.new
  break if GC.stat(:compact_count) > count
end

p GC.latest_compact_info
[aaron@tc-lan-adapter ~/g/ruby (autocompact)]$ make runruby
./miniruby -I./lib -I. -I.ext/common  ./tool/runruby.rb --extout=.ext  -- --disable-gems ./test.rb
{:considered=>{:T_OBJECT=>408}, :moved=>{:T_OBJECT=>408}}
[aaron@tc-lan-adapter ~/g/ruby (autocompact)]$
```

Recap

New methods:

- GC.enable_autocompact
- GC.disable_autocompact
- GC.last_compact_info

New statistics in GC.stat:

- GC.stat(:read_barrier_faults)

Diff is here: https://github.com/ruby/ruby/pull/3547

Associated revisions

Revision 67b2c21c327c96d80b8a0fe02a96d417e85293e8 - 11/02/2020 10:42 PM - tenderlovemaking (Aaron Patterson)

Add GC.auto_compact= true/false and GC.auto_compact

- GC.auto_compact=, GC.auto_compact can be used to control when compaction runs. Setting auto_compact= to true will cause compaction to occurr duing major collections. At the moment, compaction adds significant overhead to major collections, so please test first!

[Feature #17176]

Revision 67b2c21c327c96d80b8a0fe02a96d417e85293e8 - 11/02/2020 10:42 PM - tenderlovemaking (Aaron Patterson)

Add GC.auto_compact= true/false and GC.auto_compact

- GC.auto_compact=, GC.auto_compact can be used to control when compaction runs. Setting auto_compact= to true will cause compaction to occurr duing major collections. At the moment, compaction adds significant overhead to major collections, so please test first!

[Feature #17176]

Revision 67b2c21c - 11/02/2020 10:42 PM - tenderlovemaking (Aaron Patterson)

Add GC.auto_compact= true/false and GC.auto_compact

- GC.auto_compact=, GC.auto_compact can be used to control when compaction runs. Setting auto_compact= to true will cause compaction to occurr duing major collections. At the moment, compaction adds significant overhead to major collections, so please test first!

[Feature #17176]

History

#1 - 09/16/2020 09:08 PM - Eregon (Benoit Daloze)

API-wise, GC.autocompact = true/false as suggested by @ioquatix (Samuel Williams) on the PR, or GC.auto_compact = true/false, sounds nicer to me than 2 separate methods.

#2 - 09/16/2020 09:15 PM - tenderlovemaking (Aaron Patterson)

Eregon (Benoit Daloze) wrote in #note-1:

> API-wise, GC.autocompact = true/false as suggested by @ioquatix (Samuel Williams) on the PR, or GC.auto_compact = true/false, sounds nicer to me than 2 separate methods.

I mainly went with GC.(enable|disable)_autocompact because we have GC.enable and GC.disable, but I like @ioquatix (Samuel Williams) suggestion too. (Also a GC.auto_compact?)

#3 - 09/18/2020 07:54 AM - ko1 (Koichi Sasada)

Another idea is GC.enable(compact: true/false).

BTW as I mentioned in slack, GC.compact may have an issue (can access to T_MOVED objects from Ruby level) so please fix it before merge this large commit.

(and I need to learn about this code...)

I can repeat tests with this option. Let me know if you want.

#4 - 09/18/2020 07:55 AM - ko1 (Koichi Sasada)

Also do you have any performance results, for example memory consumption or GC time and so on?

#5 - 10/01/2020 08:50 PM - tenderlovemaking (Aaron Patterson)

ko1 (Koichi Sasada) wrote in #note-4:

> BTW as I mentioned in slack, GC.compact may have an issue (can access to T_MOVED objects from Ruby level) so please fix it before merge this large commit.

I think it's an issue with a "use-after-free" bug. I believe it was fixed here: https://github.com/ruby/ruby/pull/3571

I was able to get the crash locally, but after applying ^^^ the crash goes away.

> Another idea is GC.enable(compact: true/false).

I like this better than GC.auto_compact = true/false because we can add more options. I'll change to use that.

> Also do you have any performance results, for example memory consumption or GC time and so on?

I did a survey using RDoc. The summary: Minor GC time is the same, major GC time is much slower, but I still think we should add this as an experimental feature.

Here are more details of the tests I did:

To benchmark compaction, I enabled GC::Profiler and then generated RDoc for Ruby.

I created a file x.rb that looks like this:

```
BEGIN {
  $start = Process.clock_gettime(Process::CLOCK_MONOTONIC)
  # Uncomment for automatic compaction
  #GC.enable_autocompact
  GC::Profiler.enable
}
END {
  profile = GC::Profiler.raw_data.dup
  require "csv"
  CSV($stderr) do |csv|
    syms = %i[HEAP_USE_SIZE HEAP_TOTAL_SIZE HEAP_TOTAL_OBJECTS MOVED_OBJECTS]
    csv << (["major_by", "GC_TIME"] + syms.map(&:to_s))
    profile.each do |record|
      csv << ([record[:GC_FLAGS][:major_by], record[:GC_TIME] * 100] + syms.map { |s| record[s] })
    end
  end
  $stderr.puts
  $stderr.puts
  stats = %i[ heap_allocated_pages heap_eden_pages read_barrier_faults minor_gc_count major_gc_count ]
  stats.each { |s|
    $stderr.puts "#{s}: #{GC.stat(s)}"
  }
  elapsed = Process.clock_gettime(Process::CLOCK_MONOTONIC) - $start
  $stderr.puts "Elapsed: #{elapsed}"
}
```

Then I generated RDoc like this:

```
rm -rf .ext/rdoc
./ruby --disable-gems -I. -r x "./libexec/rdoc" --root "." --encoding=UTF-8 --all --ri --op ".ext/rdoc" --page
-dir "./doc" --no-force-update  "." 2>logs/no_autocompact_$i.log
```

I did this 100 times with automatic compaction enabled, and with automatic compaction disabled.

This patch only adds automatic compaction to major GCs, so I don't expect minor GC times to change.
Here is a plot of the minor GC time:

94854245-775c6180-03e1-11eb-8dcc-9c56988678c7.png
X axis is the GC invoke count, Y axis is time in seconds. As expected, there is really no change.

Here is a plot of the major GC time:

94854341-9e1a9800-03e1-11eb-9666-796ab8e0708c.png
X axis is the GC invoke count, Y axis is time in seconds. Adding compaction makes the major invocations significantly slower. The algorithm for doing a major GC doesn't change, so the difference in these lines is completely due to the time it takes to compact the heap.

If we divide heap size by time to get "Objects Per Second", then we can get an idea of GC throughput:

94855783-e20e9c80-03e3-11eb-9d2e-cd79b452b044.png
It's pretty clear that the throughput is lower with compaction enabled.

One interesting thing I've found. If we plot "heap total objects" and compare it to GC time, it makes a graph like this:

94860534-0fab1400-03eb-11eb-8b46-e29cb1cef9f9.png
Whenever the heap size remains stable, GC compaction gets significantly faster. In other words, when the blue line stays level, the red line goes down.

My hypothesis is that this is due to the way the GC adds new pages when the heap expands. When the heap expands, the GC adds pages on the left side of the heap. So the top of "heap_eden" always points at the newest page. The compaction algorithm packs to the left, so as we add new pages, more objects move (even though they didn't need to).

Anyway, I think that introducing this feature has value because it will help people test with more aggressive object movement. I would like auto compaction to be enabled by default some day, but I think for now we can add this as an experimental feature.

Once the speed is acceptable and it seems like bugs are worked out, then we can enable it by default.

#6 - 10/01/2020 09:16 PM - Eregon (Benoit Daloze)

tenderlovemaking (Aaron Patterson) wrote in #note-5:

> ko1 (Koichi Sasada) wrote in #note-4:
>
> > Another idea is GC.enable(compact: true/false).
>
> I like this better than GC.auto_compact = true/false because we can add more options. I'll change to use that.

One potential issue is that this pattern GC.disable; begin; ...; ensure; GC.enable; end will unintentionally lose the compact flag.
I'm not sure it matters too much because (I hope) this pattern is rare.

Another downside is it makes it more complicated to know if auto-compaction can be set (need to check arity, and won't work for the second flag vs GC.respond_to?(:auto_compact=).

#7 - 10/01/2020 09:21 PM - byroot (Jean Boussier)

A third downside is that it makes it complicated to ensure compaction is disabled in a block. e.g. the common pattern of:

```
def with_state
  previous = Some.state
  Some.state = false
  yield
ensure
  Some.state = previous
end
```

For instance I can see myself needing to disable compaction in https://github.com/Shopify/heap-profiler, would be nice if I had a simple way to restore it to it's previous state.

#8 - 10/01/2020 09:37 PM - tenderlovemaking (Aaron Patterson)

byroot (Jean Boussier) wrote in #note-7:

> A third downside is that it makes it complicated to ensure compaction is disabled in a block. e.g. the common pattern of:
>
> ```
> def with_state
>   previous = Some.state
>   Some.state = false
>   yield
> ensure
>   Some.state = previous
> end
> ```
>
> For instance I can see myself needing to disable compaction in https://github.com/Shopify/heap-profiler, would be nice if I had a simple way to restore it to it's previous state.

▯▯

Ok, I'll change it to GC.auto_compact = true/false and GC.auto_compact

#9 - 10/16/2020 11:39 PM - tenderlovemaking (Aaron Patterson)

- Subject changed from GC.enable_autocompact / GC.disable_autocompact to GC.auto_compact / GC.auto_compact=(flag)

I've updated the patches on the PR to reflect the new API. Now you can do GC.auto_compact = true/false and GC.auto_compact.

#10 - 10/20/2020 07:39 PM - tenderlovemaking (Aaron Patterson)

I made another pull request that sets the default to "ON":

https://github.com/ruby/ruby/pull/3316

It looks like there is one optional ruby spec test that is failing, but all of the other tests pass. Since the test suites pass with automatic compaction enabled, it makes me pretty confident about the implementation (though I'm sure there are bugs somewhere).

#11 - 10/26/2020 06:40 AM - matz (Yukihiro Matsumoto)

Accepted as long as it's turned off by default.

Matz.

#12 - 11/02/2020 10:43 PM - tenderlovemaking (Aaron Patterson)

- Status changed from Open to Closed

Applied in changeset git|67b2c21c327c96d80b8a0fe02a96d417e85293e8.

Add GC.auto_compact= true/false and GC.auto_compact

- GC.auto_compact=, GC.auto_compact can be used to control when compaction runs. Setting auto_compact= to true will cause compaction to occurr duing major collections. At the moment, compaction adds significant overhead to major collections, so please test first!

[Feature #17176]