Pat Shaughnessy blogger, rubyist, aspiring author

Why You Should Be Excited About Garbage Collection in Ruby 2.0

March 23rd 2012 — 49 Comments

While not very glamorous, Bitmap Marking Garbage Collection is a dramatic, creative innovation!

You may have heard last week how Innokenty Mihailov's great Enumerable::Lazy feature was accepted into the Ruby 2.0 code base. But you may not have heard about an even more significant change that was merged into Ruby 2.0 in January: a new algorithm for garbage collection called "Bitmap Marking." The developer behind this sophisticated and innovative change, Narihiro Nakamura, has been working on this since 2008 at least and also implemented the "Lazy Sweep" garbage collection algorithm already included in Ruby 1.9.3. The new Bitmap Marking GC algorithm promises to dramatically reduce overall memory consumption by all Ruby processes running on a web server!

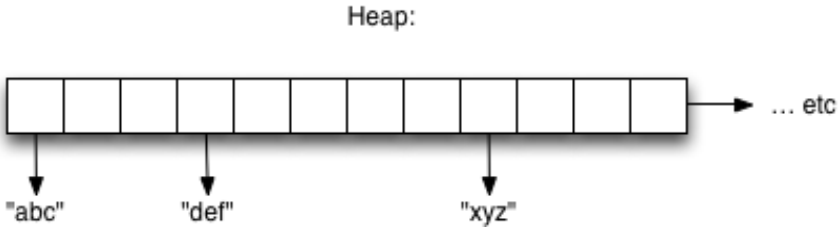
But what does "bitmap marking" really mean? And exactly why will it reduce memory consumption? If you know Japanese you can read a detailed academic paper published in 2008 by Narihiro Nakamura along with Yukihiro ("Matz") Matsumoto. I was so interested I spent some time this week studying the garbage collection code in MRI Ruby myself, and this article will summarize what I learned. You won't get any Ruby programming tips here today, but hopefully you'll come away with a better understanding of how garbage collection actually works internally, of why Ruby 2.0 is something to look forward to, and of how innovative and creative the Ruby core developers really are.

Mark and Sweep

As I explained in my article from January, Never create Ruby strings longer than 23 characters, every Ruby string value is saved internally by MRI in a C structure called RString, short for "Ruby String." Each RString structure is split into two halves like this:

At the bottom we have the actual string data itself, while at the top I've shown the word "flags" to represent various internal metadata values about the string that Ruby keeps track of. It turns out that all values used by your Ruby program are saved in similar structures called RArray, RHash, RFile, etc. They all share the same basic layout: some data and the same set of flags. The common name for this type of structure, which is shared across all the internal object types, is RValue – meaning "Ruby Value."

Ruby allocates and organizes these RValue structures in arrays called "heaps." Here's a conceptual diagram of a Ruby heap array, containing the three string values along with many other RValue's:

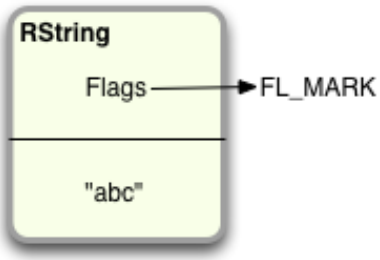


As your Ruby program runs, whenever you create a new variable or value of some type the Ruby interpreter finds an available RValue structure in the heap and uses it to save the new value. Of course, you don't need to worry about this at all; it's all handled automatically and smoothly for you.

Well – it's not that smooth at times, actually. What happens when the RValue structures in the heap run out? …when there are none left to save a new value your program requires? This actually happens more frequently than you might expect because there are many RValue structures that you might not be aware of created internally by Ruby. In fact, your Ruby code itself is converted into a large number of RValue structures as it is parsed and converted into byte code.

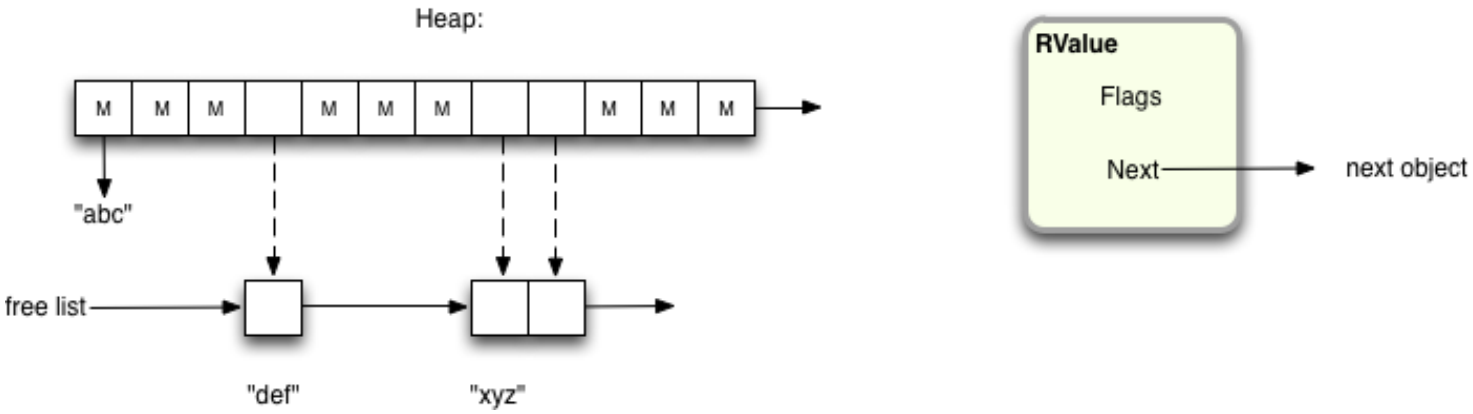
When there are no more RValue structures available and your program needs to save a new value, Ruby runs its "garbage collection" (GC) code. The garbage collector's job is to find which of these RValue's are no longer being referenced by your program and can be recycled and reused for some other value. Here's how it works, at a high level….

First, the GC code "marks" all of the active RValue structures, That is, it loops through all of the variables and other active references that your program has to RValue structures, and marks each one using one of those internal flags called FL_MARK.



This is the first half of Ruby's "Mark and Sweep" GC algorithm. The marked structures are actively being used by your Ruby program and cannot be freed or reused.

Once all the structures in the system are marked, the remaining structures are "swept" into a single linked list using the "next" pointer in each RValue structure: In this diagram, I've shown the FL_MARK flags in the heap array with the letter "M," and below that you can see the list of unmarked RValue's, called the "free list:"



As you might guess, the free list can now be used to provide new RValue structures to your Ruby program as it continues to run. Now every time your Ruby program allocates a new object or value, it uses an RValue from the free list, and removes it from the list. Eventually the free list will become empty again and Ruby will have to start another garbage collection.

After a while it might be that there are no unmarked structures left in the heap at all, that all of the available RValue's are being used, in which case Ruby will allocate an entire new heap with more RValue structures. (Actually it allocates new heaps 10 at a time.) A typical Ruby program might end up having many different heap arrays.

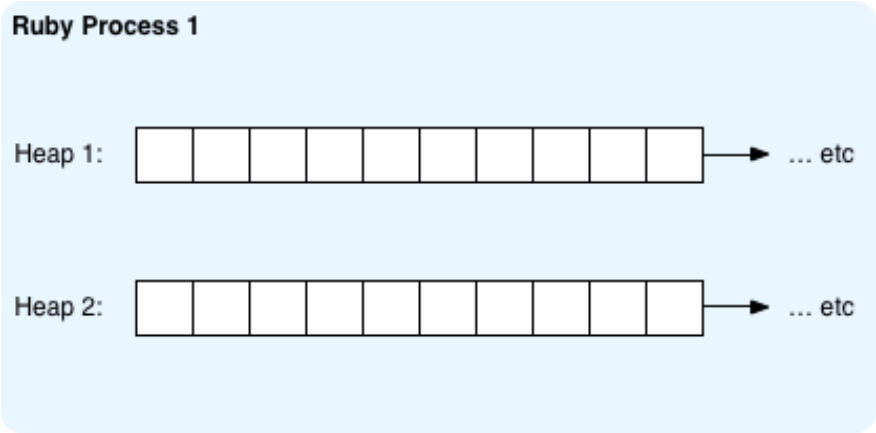
Copy-On-Write: how Unix shares memory across different child processes

Before we can get to "Bitmap Marking" and why it's important, we first need to learn about a feature of Linux and other Unix and Unix-like operating systems that is related to memory management and memory allocation: Copy-On-Write optimization. On these OS's when a process calls fork to create a child process which is a copy of itself, the new child process will share all of the memory – all of the data, variables, etc.

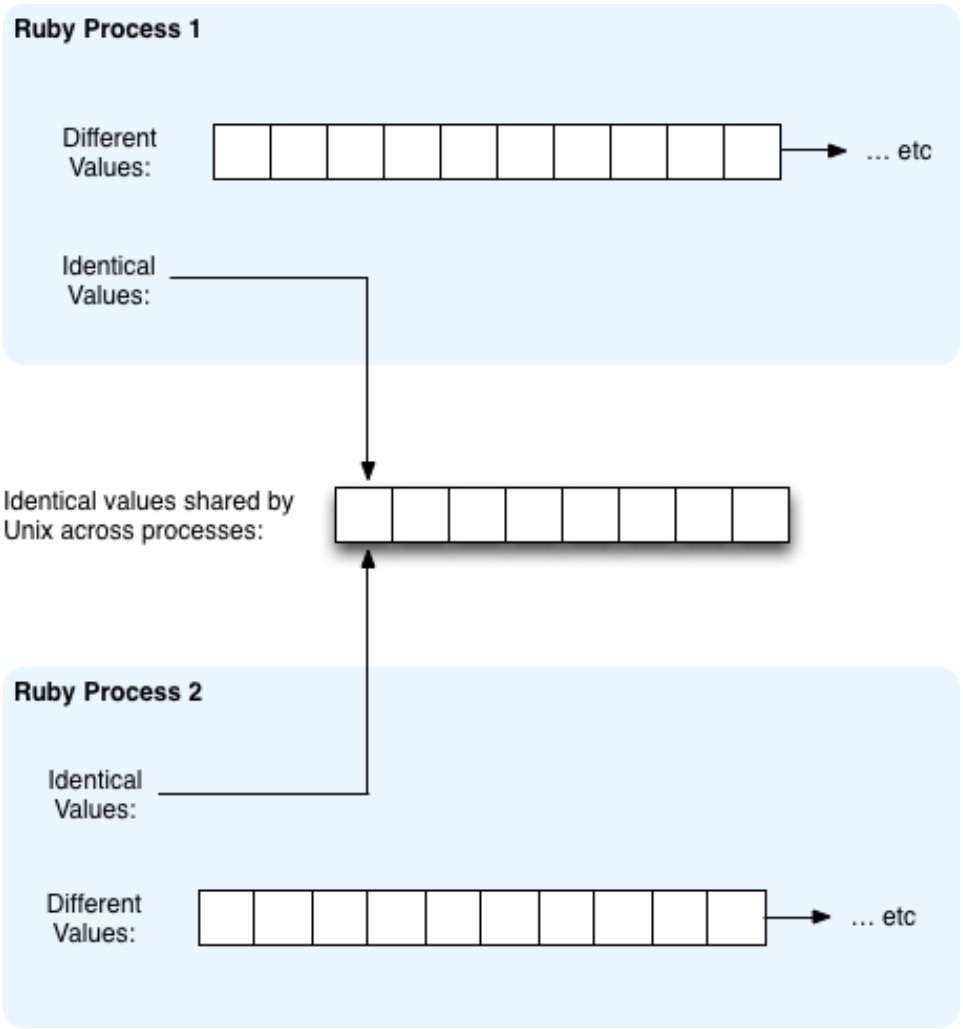
– that the parent had previously allocated. This makes the fork call much faster by avoiding copying memory around unnecessarily, and also reduces the total amount of memory required.

This is called “Copy-On-Write” because separate copies of a shared memory segment are made when and if one of the child processes tries to modify the shared memory. This is similar to the trick that the Ruby interpreter itself uses to manage RString values; for details check out a post I wrote in January about this: Seeing double: how Ruby shares string values.

To understand this better, take a look at this conceptual diagram of a Ruby process:



Here I’ve shown a Ruby program that has two heaps as an example. Now suppose this Ruby program is running on a web server – maybe it’s a Rails web application – and now a second HTTP request arrives from another user:

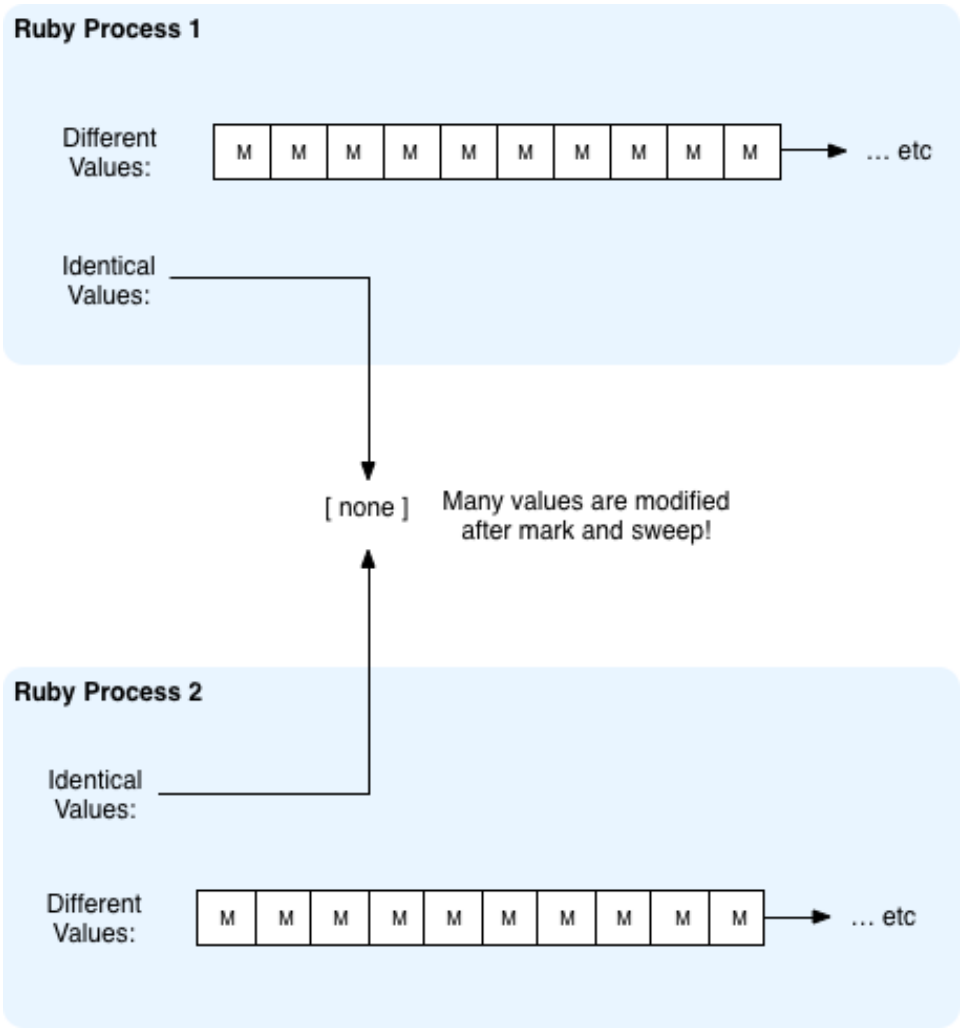


Now we have two Ruby processes running. Possibly this server is running Apache with something like Passenger that forks a separate Ruby process to handle each HTTP request.

The nice thing about Copy-On-Write optimization in Linux is that many of the RValue structures in the heap arrays can be shared between these two Ruby programs, since they often contain the same values. It might not seem that this would be the case at first glance; why would many – or any – of the variables in two Ruby programs be the same? But remember on a web server you are actually running two or more copies of the same code, creating the same variables over and over again. Also, many of the RValue structures in the heap actually correspond to the parsed version of your Ruby program itself – the nodes in the "Abstract Syntax Tree" (AST). Since each process is running the same code, all of these nodes will have the same values and won't ever change. Of course, some of the data values will be different and will be saved separately inside each process – user data typed into web forms and submitted, results of SQL queries on different records, etc.

But, as great as this sounds, it doesn't actually work for Ruby!

Why not? Well, because as soon as Ruby has to run the Mark & Sweep garbage collection algorithm I explained above, all of those AST nodes and many other RValue structures in the heap are all marked, since they are still being used by the Ruby program. This means they are modified to set the FL_MARK flag, and the Copy–On-Write code in the operating system has to start creating new copies of the memory. So in fact on a typical Ruby web server this is what happens:

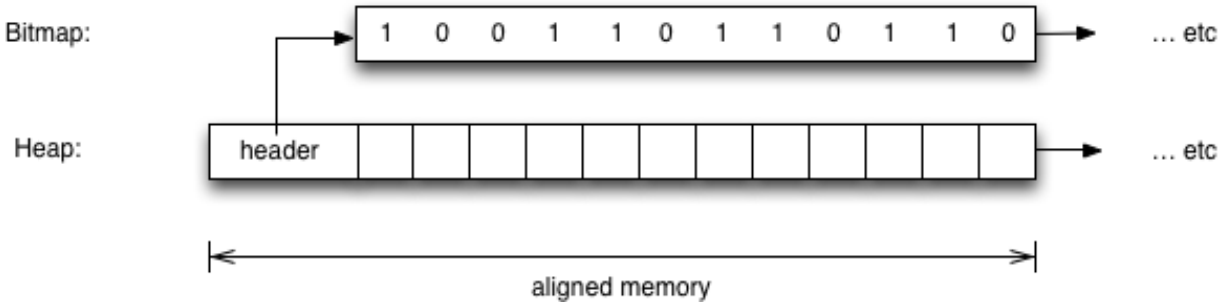


That one little FL_MARK bit is wreaking havoc! It prevents what would normally be a tremendous reduction in server memory usage from actually happening.

One important note here: Hongli Lai from Phusion, the creators of the popular Passenger middleware component that connects Apache with Rack based Ruby apps, patched Ruby 1.8 and created a new version of Ruby known as Ruby Enterprise Edition that solves this problem and contains a number of other performance improvements. So in fact many Ruby 1.8 apps that use REE have been able to take advantage of Unix Copy-On-Write for years now. But Copy-On-Write still doesn't work with standard MRI Ruby 1.8 or 1.9.

Garbage Collection in Ruby 2.0: Bitmap Marking

Here's where Narihiro Nakamura's changes for Ruby 2.0 come in! Instead of using the FL_MARK bit in each of the RValue structures to indicate that Ruby is still using an value and that it cannot be freed, Ruby 2.0 saves this information in something called a "bitmap" instead. No… here "bitmap" does not refer to an image file; "bitmap" in this context refers to a literal collection of bits mapped back to the RValue structures:

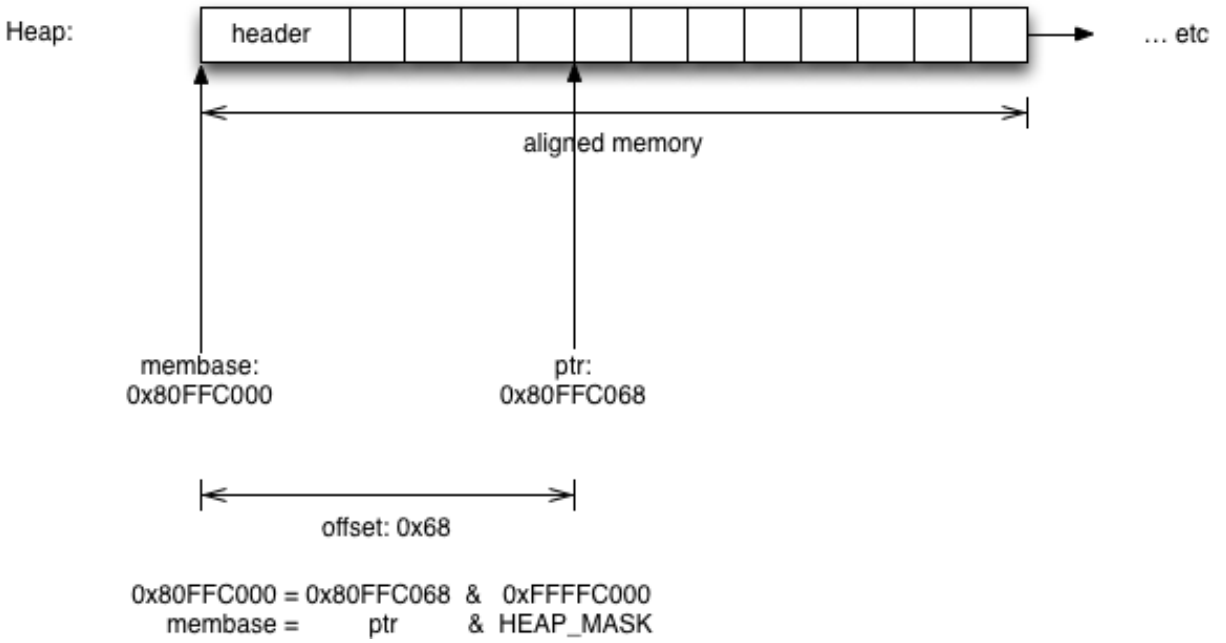


For each heap in Ruby 2.0 there is now a corresponding memory structure that contains a series of 1 or 0 bit values. As you might guess, the 1 values are equivalent to the FL_MARK flag being set in a Ruby 1.8 or Ruby 1.9 process, while a 0 is equivalent to the FL_MARK flag not being set. In other words, the FL_MARK bits have been moved out of the RString and other object value structures, and into this separate memory area called the bitmap.

Narihiro implemented this by adding a header structure to the beginning of each heap which contains a pointer to the bitmap corresponding to that heap's RValue structures, along with some other values. What this means is that Ruby 2.0 can now mark all of the in-use structures during the "mark" portion of the GC processing without actually modifying the structures themselves, allowing Unix to continue to share memory across different Ruby processes! The bitmaps themselves, of course, are modified frequently by Ruby 2.0, but since they use a contiguous stream of bits they are actually quite small and can be saved separately in each process without using too much memory.

One interesting and important detail here is that the memory allocated for heaps now must be "aligned." What this means is that when allocating memory for the heap, instead of calling malloc as usual, the Ruby C code calls posix_memalign which on a Linux or Unix operating system returns the new memory aligned to a power of two address boundary.

What the heck does that mean? Well if you're familiar with C programming or bitwise arithmetic, it allows the Ruby C code to quickly calculate the location of the "header" structure, which contains the pointer to the bitmap, from a given RValue object's memory address. Let's take another look at a Ruby 2.0 heap:



Suppose that the Ruby 2.0 garbage collector code needs to mark the fifth RValue object in this heap,

referred to by the ptr value. The memory alignment trick allows Ruby 2.0 to take the ptr value and quickly calculate the address of it's heap header structure. All Ruby 2.0 has to do is mask out the last few bits of the RValue address, the "68" hexadecimal offset in this example, to obtain the address of the header structure, at "membase" or 0x80FFC000 in this 32-bit example.

Conclusion

Garbage collection isn't the most glamorous or interesting part of the Ruby language at first glance, but as we've seen if you take a close look at how it works there's a lot of interesting innovation going on. Practically speaking, the Bitmap Marking change will help MRI Ruby 2.0 work better in production web server environments, reducing memory consumption dramatically. But I view Bitmap Marking less as a practical improvement that will help my Rails apps run better, and more just as an exciting, creative solution to a complex problem. It was great fun learning how GC works in Ruby 2.0 and I hope you now have a better appreciation of all the hard work the talented Ruby core team is doing!

http://morgo-crosswords-solution.com

• Reply • Share ›

patshaughnessy Mod → Leandro López (inkel) • 4 years ago

¡Gracias Leandro! I LOVED that Narihiro started the presentation in Spanish :) What a class act!

I'll definitely set aside some time to watch this over the weekend - thanks so much for the link!

• Reply • Share ›

Leandro López (inkel) → patshaughnessy • 4 years ago

My pleasure! But that wasn't the only trick he had on his sleeve ;) hint: pay careful attention to the slides! One of my favorite presentations of that evening.

1 • Reply • Share ›

Jerry Cheung • 4 years ago

Interesting read! I love posts about ruby internals.

Could you talk more about the memory alignment for the allocated heaps? From your explanation, it sounds like all the RValues would need to less than or equal to a fixed size in order for the bitmap to index a position. Does this mean that if you have a large value, then you would need to have several chained together RValues in order to reconstruct the original value?

1 • Reply • Share ›

patshaughnessy Mod → Jerry Cheung • 4 years ago

Good question - yes, the RValue structures are all the same length, which makes this algorithm possible and also simplifies a lot of other C code in MRI. What happens if the data value doesn't fit into the RValue is that there's an additional memory allocation and the data is saved elsewhere, and a pointer to it saved in the RValue. So no chaining of RValues, but instead the "data" becomes a pointer to the actual data.

In my post from January, Never create Ruby strings longer than 23 characters, I wrote a lot more about how RString works. As you might guess, 23 characters of a string fit into the RValue, but not more. The other data structures, like RArray, etc., have similar behavior.

See: http://patshaughnessy.net/2012...

2 • Reply • Share ›

vikramjit sidhu • a year ago

Hi, very interesting and well written post! I couldnt understand this part however "First, the GC code “marks” all of the active RValue structures, That is, it loops through all of the variables and other active references that your program has to RValue structures". Does the Ruby

program keep a separate structure of all the variables currently being referenced by it? Or does it iterate through the whole array heap and find active references?

• Reply • Share ›

patshaughnessy Mod ➚ vikramjit sidhu • a year ago

Thanks! Yea sorry that sentence was a bit vague.

What happens is that all of the values (variables, objects, classes - everything) in your program are stored in the heap. Each object, depending on what type it is, keeps track of which other objects it contains or references. For example, an array keeps track of all of the values in the array. An object keeps track of its instance variables (and class and other things). So all of these references form a large tree - the leaf nodes are simple values that don't reference anything else, and the root of the tree will be some internal pointers the Ruby VM keeps track of. Then, during the marking process Ruby walks this tree, starting at the root (or roots - there are more than one) and following the references to all the objects your code is actually still referencing. Object you no longer reference will not be marked.

• Reply • Share ›

Gaurav Reddy • 2 years ago

Thanks for sharing :D

• Reply • Share ›

Martin Grenfell • 2 years ago

Thanks, this is a very interesting article, and well written for us GC noobs :)

• Reply • Share ›

Lauree Roberts ➚ Martin Grenfell • a year ago

Thanks for the great write up.

I actually agree with Martin.

Ruby on Rails Development

• Reply • Share ›

Agis Anastasopoulos • 3 years ago

Actually, in CoW, the copy is done whether the parent *or* the child process tries to modify the memory.

• Reply • Share ›

Someoneee • 3 years ago

Awesome! I just don't understand *when* is GC invoked on the child process (ex. in a Unicorn worker) and breaks the CoW functionality. Is it guaranteed that the GC will run on *every* child process when it's forked and modify it's memory? Or the GC invoked on the master process will break CoW?

• Reply • Share ›

patshaughnessy Mod → Someoneee • 3 years ago

Thanks for reading!

Sorry, I don't think there's a simple answer to this. Ruby creates and releases objects quite frequently; not only the objects you create and use in your code but also internal objects, that correspond to the AST nodes or the internal parsed version of your Ruby code for example. So most likely GC will run in every Ruby process. Exactly how often it runs might be configureable; I'm not sure off the top of my head. You can also disable GC entirely. I'm also guessing that it's equally likely the master or the child process would break the CoW.

I think it would be really interesting to model or measure Ruby's GC behavior at a detailed level - how often does GC actually run? which objects are created and released most often? which copy of the process uses GC more? or do they use it equally often? etc. That might lead us to an accurate estimate for how much Narihiro's work in Ruby 2.0 is saving us. There are GC debugging options available at the C source code level that might make this possible. cheers.

• Reply • Share ›

Marc d'Entremont • 3 years ago

Great post. Thanks so much for the clear descriptions. I'm wondering if having specific heap collections for the instructions separate from data could further improve the memory sharing and reduce the effort in the marking

• Reply • Share ›

patshaughnessy Mod → Marc d'Entremont • 3 years ago

Thanks! That's an interesting idea about separate collections for instructions/AST nodes - it probably would help.

• Reply • Share ›

Jarl • 3 years ago

Thanks for a great article. Very interesting, sounds very promising.

• Reply • Share ›

Coyo Stormbringer • 3 years ago

although I did not understand everything in that article, a more efficient garbage collection would be great for ruby. I can't wait to see what ruby 2.0 is like!

• Reply • Share ›

cantbecool • 3 years ago

Thank you for writing such an informative post on Rubies GC and its changes in 2.0. It's great to finally be able to know what is actually going on with Ruby under the hood, albeit at a higher

level.

• Reply • Share ›

we4tech • 3 years ago

Thanks for details explanation didn't know the surprise on forked process copy on write :)

• Reply • Share ›

Chris Holaday • 3 years ago

Love the article, I always wondered how REE got its memory size smaller. Keep up the great work, your material is very interesting and its presentation is top quality.

• Reply • Share ›

patshaughnessy Mod → Chris Holaday • 3 years ago

Thanks for reading, Chris!

• Reply • Share ›

John Crepezzi • 4 years ago

Thanks for the great write-up - thanks for the Ruby internals content, there should be more like it

• Reply • Share ›

patshaughnessy Mod → John Crepezzi • 4 years ago

You're welcome! Definitely more Ruby Internals posts and other content on the way here... stay tuned.

• Reply • Share ›

shadabahmed • 4 years ago

Really great writeup Pat :)

• Reply • Share ›

Noname • 4 years ago

" reducing memory consumption dramatically" What sort of percentage should we expect? 50% reduction? (Just in my mind anything less would not really be dramatic), and Ruby is actually quite heavy on memory usage so some breakthrough improvement is much needed.

• Reply • Share ›

patshaughnessy Mod → Noname • 4 years ago

I'm not sure off the top of my head - but it should be easy to measure with some actual Rails apps using techniques similar to what Matthew Conway and Narihiro Nakamura posted in other comments here. I'll try that over the next few days and let you know... it may also depend on the type of Ruby app.

• Reply • Share ›

Kyle Drake • 4 years ago

Are there security/buffer overflow implications with using posix_memalign instead of malloc in this manner? Does not using malloc here prevent the random allocation of memory?

Reply • Share ›

Kyle Drake ➚ Kyle Drake • 4 years ago

This wikipedia section talks about malloc randomization a bit, just so there's context: http://en.wikipedia.org/wiki/O...

Reply • Share ›

patshaughnessy Mod ➚ Kyle Drake • 4 years ago

Interesting question - I have no idea. I agree it's something to be concerned about, since you're right it's possible that posix_memalign works in a very different manner and might cause Ruby 2.0 to hit performance or security issues because of it...

Reply • Share ›

Theo Hultberg • 4 years ago

Will the GC still be stop-the-world? The number one problem with the current GC is that it isn't working in the background.

Reply • Share ›

patshaughnessy Mod ➚ Theo Hultberg • 4 years ago

Nope! Actually that's exactly the problem that Narihiro Nakamura addressed in Ruby 1.9.3 with his "Lazy Sweep" algorithm. In a nutshell Lazy Sweeping works by garbage collecting only until a free RValue is found, and then returning it - instead of freeing all the unused RValue's.

I didn't have time to get into that today in this post, but maybe I'll do a follow up article on how Lazy Sweeping works someday.

1 Reply • Share ›

pierogi emoji ➚ patshaughnessy • 4 years ago

There was a pretty good summary on the LL1 mailing list (http://people.csail.mit.edu/gr..., though the author may be wrong in attributing it to Noman Ramsey.

Reply • Share ›

Astro • 4 years ago

Is that improvement specific to forking processes or has the Ruby memory situation improved in general? Also, didn't REE implement this years ago?

Reply • Share ›

patshaughnessy Mod ➚ Astro • 4 years ago

Yes it's specific to forking. And yes as I mentioned in the article REE did this years ago.

I didn't have time to research REE internals for this post, so I'm not sure whether Hongli's implementation used bitmap marking or something else.

1 • Reply • Share ›

Hongli Lai patshaughnessy • 4 years ago

We do. The REE implementation works in the exact same way except we identify the containing heap with binary search instead of relying on aligned memory allocation.

1 • Reply • Share ›

patshaughnessy Mod Hongli Lai • 4 years ago

Great work on REE! I could write another entire article about all the stuff you guys did with it.

Any idea if the Ruby team is planning to merge some of the other improvements you made into Ruby 2.0?

• Reply • Share ›

Astro patshaughnessy • 4 years ago

• Reply • Share ›

roger pack Astro • 4 years ago

yes it's still stop the world for the mark phase.

• Reply • Share ›

josh_cheek • 4 years ago

If GC runs and identifies some memory as needing to be swept, then won't it have to to change the heap anyway (e.g. to put a new string in the RValue)? Or is it able to COW for each individual RValue?

• Reply • Share ›

patshaughnessy Mod → josh_cheek • 4 years ago

Great question.

I'm guessing here, but I believe that Linux CoW works on a per memory page basis, and not on an individual RValue basis. So if any RValue in the memory page changes the entire page would be copied. On my 64-bit Mac the page size is 4k and each RValue is 40 bytes, which is probably the same on most 64-bit Linux servers. That's 102 RValue's per memory page, depending on how exactly Linux uses memory pages while allocating the aligned heap arrays.

The other thing to keep in mind is that there are many 1000's of RValue's running in a typical Ruby program, and many of these are the AST nodes that are created as the target script is parsed and then never changed. You can imagine for a Rails app how many syntax nodes are created for all that Ruby code. I imagine that the primary memory savings comes while parsing the Ruby program when it starts.

Anyway, Narihiro's optimization will help tremendously because before Ruby would modify *every* RValue that was *not* going to be freed. Now RValue's are only modified when they are freed and reused.

But this sounds like a fun experiment… I wonder if there's any way to actually detect and measure the CoW page fault events in Linux?

• Reply • Share ›

josh_cheek → patshaughnessy • 4 years ago

This implies grouping memory by longevity would be beneficial as it would be less volatile and thus more likely to be in a page that doesn't change. I don't know much about GC, but I think that grouping by "generations" is a strategy. Perhaps something along those lines would be used to keep the stable memory together and the volatile memory together.

• Reply • Share ›

patshaughnessy Mod → josh_cheek • 4 years ago

Yea I believe that's how the JVM and other garbage collectors work... it would be interesting to explore the details of that someday too. There's some interesting discussion going on about this now over on HN: http://news.ycombinator.com/it...

• Reply • Share ›

Matthew Conway • 4 years ago

I'm looking forward to a COW friendly GC, but I'm not sure the rune in ruby head is working all that well - could be a problem with my test methodology though. Please educate me if that is case:
https://gist.github.com/217312...

• Reply • Share ›

Narihiro Nakamura Matthew Conway • 4 years ago

Hi.
Rss includes shared memory, so we can't see the improvement by `pss -orss pid`.

I fixed your program as the following. Please check it :)
https://gist.github.com/217677...

1 • Reply • Share ›

Matthew Conway Narihiro Nakamura • 4 years ago

Awesome, thanks Narihiro!

• Reply • Share ›

patshaughnessy Mod Matthew Conway • 4 years ago

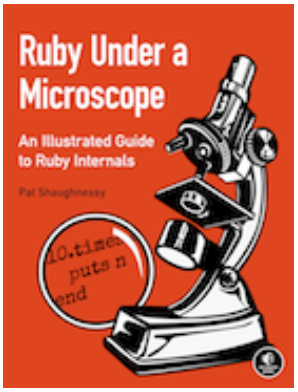
Interesting test idea Matthew - I'll have to dig into this for a while to understand what you've done. If I figure it out, I'll let you know here later. cheers

• Reply • Share ›

Subscribe

- Follow @pat_shaughnessy

Buy my book



- Ruby Under a Microscope

Popular

- Never create Ruby strings longer than 23 characters
- Why You Should Be Excited About Garbage Collection in Ruby 2.0
- Visualizing Garbage Collection in Ruby and Python
- The Joke Is On Us: How Ruby 1.9 Supports the Goto Statement

Recent

- What Do Perl and Go Have in Common?
- Don’t Let Your Data Out of the Database
- Mark Methods Private When You Don’t Test Them
- Using Rake to Generate a Blog

More...