Ruby - Feature #12020

Documenting Ruby memory model

01/25/2016 07:20 PM - pitr.ch (Petr Chalupa)

Status:	Assigned	
Priority:	Normal	
Assignee:	ko1 (Koichi Sasada)	
Target version:		

Description

Defining a memory model for a language is necessary to be able to reason about a program behavior in a concurrent or parallel environment.

There was a document created describing a Ruby memory model for concurrent-ruby gem, which fits several Ruby language implementations. It was necessary to be able to build lower-level unifying layer that enables creation of concurrency abstractions. They can be implemented only once against the layer, which ensures that it runs on all Ruby implementations.

The Ruby MRI implementation has stronger undocumented guaranties because of GIL semantics than the memory model, but the few relaxations from MRIs behavior allow other implementations to fit the model as well and to improve performance.

This issue proposes to document the Ruby memory model. The above mentioned memory model document which was created for concurrent-ruby can be used as a starting point: https://docs.google.com/document/d/1pVzU8w_QF44YzUCCab990Q_WZOdhpKolCIHaiXG-sPw/edit#. Please comment in the document or here.

The aggregating issue of this effort can be found here.

Related issues:

Related to Ruby - Feature #12019: Better low-level support for writing concur...	**Assigned**

History

#1 - 01/25/2016 09:02 PM - normalperson (Eric Wong)

email@pitr.ch wrote:

> This issue proposes to document the Ruby memory model. The above mentioned memory model document which was created for concurrent-ruby can be used as a starting point: https://docs.google.com/document/d/1pVzU8w_QF44YzUCCab990Q_WZOdhpKolCIHaiXG-sPw/edit#

Hello, I am interested in the topic but do not use JavaScript.
Can you please provide a plain-text or basic HTML version?
Thank you.

It tried changing the "/edit#" in the URL to "/pub" but could not see
anything useful.

For reference, C Ruby programmers may find Linux memory-barriers.txt
useful:

https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/plain/Documentation/memory-barriers.txt

AFAIK, we remain C89-compatible for old compilers, so we have many undefined
behaviors to define and deal with.

#2 - 01/25/2016 10:19 PM - Eregon (Benoit Daloze)

- Related to Feature #12019: Better low-level support for writing concurrent libraries added

#3 - 01/25/2016 10:26 PM - Eregon (Benoit Daloze)

- File RubyMemoryModel.rtf added

Eric Wong wrote:

> email@pitr.ch wrote:

> This issue proposes to document the Ruby memory model. The above mentioned memory model document which was created for concurrent-ruby can be used as a starting point:
> https://docs.google.com/document/d/1pVzU8w_QF44YzUCCab990Q_WZOdhpKolClHaiXG-sPw/edit#

> Hello, I am interested in the topic but do not use JavaScript.
> Can you please provide a plain-text or basic HTML version?
> Thank you.

I attached a RTF version to this issue.

#4 - 01/26/2016 02:41 AM - normalperson (Eric Wong)

eregontp@gmail.com wrote:

> I attached a RTF version to this issue.

Thanks.

I'm not sure if shared memory is even a good model for Ruby (and not my decision). Anyways, my comments below if matz/ko1 decide to go down this route.

Background: I am only a simple C programmer with some familiarity with Userspace RCU and Linux kernel memory model. I have zero experience in Java, and I do not know any C++ beyond what is in C.

For those unfamiliar with RCU, it is basically a poor man's GC; and all Rubies have a GC implementation anyways. In fact, working with the quirks with our conservative GC is not much different from working with RCU and the relaxed memory ordering model it favors.

> Core behavior
> Following sections covers the various storages in the Ruby language (e.g. local variable, instance variable, etc.). We consider the following operations:
> - read - reading a value from an already defined storage
> - write - writing a value from an already defined storage
> - define - creates a new storage and stores the default value or a supplied value
> - undefine - removes an existing storage
>
> Key properties are:
> - volatility (V) - A written value is immediately visible to any subsequent volatile read of the same variable on any Thread. It has same meaning as in Java, it provides sequential consistency. A volatile write happens-before any subsequent volatile read of the same variable.

Perhaps we call this "synchronous" or "coherent" instead.
The word "volatile" is highly misleading and confusing to me as a C programmer. (Perhaps I am easily confused :x)

Anyways, I am not convinced (volatile|synchronous|coherent) access should happen anywhere by default for anything because of costs.

Those requiring synchronized data should use special method calls to ensure memory ordering.

> Constant variables
> - volatility - yes
> - atomicity - yes
> - serializability - yes
> - scope - a module
>
> A Module or a Class definition is actually a constant definition. The definition is atomic, it assigns the Module or the Class to the constant, then its methods are defined atomically one by one.
> It’s desirable that once a constant is defined it and its value is immediately visible to all threads, therefore it’s volatile.

<snip (thread|fiber)-local, no objections there>

> Method table

- volatility - yes
- atomicity - yes
- serializability - yes
- scope - a class

Methods are also stored where operations defacto are: read -> method lookup, write -> method redefinition, define -> method definition, undefine -> method removal. Operations over method tables have to be visible as soon as possible otherwise Threads could execute different versions of methods leading to unpredictable behaviour, therefore they are marked volatile. When a method is updated and the method is being executed by a thread, the thread will finish the method body and it'll use the updated method obtained on next method lookup.

I strongly disagree with volatility in method and constant tables. Any programs defining methods/constants in parallel threads and expecting them to be up-to-date deserve all the problems they get.

Maybe volatility for require/autoload is a special case only iff a method/constant is missing entirely; but hitting old methods/constants should be allowed by the implementation.

Methods (and all other objects) are already protected from memory corruption and use-after-free by GC. There is no danger in segfaulting when old/stale methods get run.

The inline, global (, and perhaps in the future: thread-specific) caches will all become expensive if we need to ensure read-after-write consistency by checking for changes on methods and constants made by other threads.

> Threads
> Threads have the same guarantees as in in Java. Thread.new happens-before the execution of the new thread's block. All operations done by the thread happens-before the thread is joined. In other words, when a thread is started it sees all changes made by its creator and when a thread is joined, the joining thread will see all changes made by the joined thread.

Good. For practical reasons, this should obviate the need for constant/method volatility specified above.

> Beware of requiring and autoloading in concurrent programs, it's possible to see partially defined classes. Eager loading or blocking until classes are fully loaded should be used to mitigate.

No disagreement, here :)

#5 - 01/26/2016 08:59 AM - naruse (Yui NARUSE)

@Petr

You can "publish" the document and provide a simpler view with following process:

```
To publish a file:

Open a document, spreadsheet, presentation, or drawing.
Click the File menu.
Select Publish to the Web.
While the entire file will be published, some file types have more publishing options:
Spreadsheet: Choose to publish the entire spreadsheet or individual sheets. You can also choose a publishing f
ormat (web page, .csv, .tsv, .pdf, .xlsx, .ods).
Presentation: Choose how quickly to advance the slides.
Drawing: Choose the image size for your drawing.
Click Publish.
Copy the URL and send it to anyone you'd like to see the file. Or, embed it into your website.
```

#6 - 01/26/2016 03:19 PM - pitr.ch (Petr Chalupa)

Thanks, I've published the document on following address, it'll be updated automatically, works without JS. Sorry i did not think about non-JS viewers.
https://docs.google.com/document/d/1pVzU8w_QF44YzUCCab990Q_WZOdhpKolCIHaiXG-sPw/pub

#7 - 01/26/2016 03:20 PM - pitr.ch (Petr Chalupa)

- File deleted (RubyMemoryModel.rtf)

#8 - 02/03/2016 09:04 PM - pitr.ch (Petr Chalupa)

Thank you, for taking time to read it and for your input. I apologise for delayed answer, I was rather busy lately.

> volatility (V) - A written value is immediately visible to any
> subsequent volatile read of the same variable on any Thread. It has
> same meaning as in Java, it provides sequential consistency. A volatile
> write happens-before any subsequent volatile read of the same variable.

> Perhaps we call this "synchronous" or "coherent" instead.
> The word "volatile" is highly misleading and confusing to me
> as a C programmer. (Perhaps I am easily confused :x)

We can definitely consider a different name. I would defer it for later though, to avoid confusion now.

> Anyways, I am not convinced (volatile|synchronous|coherent) access
> should happen anywhere by default for anything because of costs.
>
> Those requiring synchronized data should use special method calls
> to ensure memory ordering.

I've added following paragraph to the document explaining a little bit why volatility is preferred.

"The volatile property has noticeable impact on performance, on the other hand it’s often quite convenient property, since it simplifies reasoning about the program. Therefore unless it presents a performance issue volatility is preferred."

It tries to be in alignment with rest of the Ruby language to be user-friendly. Therefore the volatility behaviour is on Constants and similar. I've also elaborate in the document in the Constants part why is there no performance loose by making them volatile: "Ruby implementations may take advantage of constancy of the variables to avoid doing volatile reads on each constant variable read. MRI can check a version number. JRuby can use SwitchPoint, and JRuby+Truffle can use Assumptions, where both allow to treat the values as real constants during compilation."

> Constant variables
> volatility - yes
> atomicity - yes
> serializability - yes
> scope - a module
> A Module or a Class definition is actually a constant definition. The
> definition is atomic, it assigns the Module or the Class to the
> constant, then its methods are defined atomically one by one.
> It’s desirable that once a constant is defined it and its value is
> immediately visible to all threads, therefore it’s volatile.

> <snip (thread|fiber)-local, no objections there>

> Method table
> volatility - yes
> atomicity - yes
> serializability - yes
> scope - a class
> Methods are also stored where operations defacto are: read -> method
> lookup, write -> method redefinition, define -> method definition,
> undefine -> method removal. Operations over method tables have to be
> visible as soon as possible otherwise Threads could execute different
> versions of methods leading to unpredictable behaviour, therefore they
> are marked volatile. When a method is updated and the method is being
> executed by a thread, the thread will finish the method body and it’ll
> use the updated method obtained on next method lookup.

> I strongly disagree with volatility in method and constant tables. Any
> programs defining methods/constants in parallel threads and expecting
> them to be up-to-date deserve all the problems they get.

I see that this approach would be easier for Ruby implementers, on the other hand it would create very hard to debug bugs for users. Even though I agree that they should not do parallel loading, I would still like to protect them. Making both volatile should have only minor impact on code loading, if that's not the case, it should definitely be reconsidered.

> Maybe volatility for require/autoload is a special case only iff a

> method/constant is missing entirely; but hitting old methods/constants
> should be allowed by the implementation.

Volatility of require/autoload and the fact that it blocks when other thread is loading given file/constant are very useful in parallel environment to make sure that some feature/class is fully loaded before using it. Both are usually used only on program paths which run only once during loading or reloading, therefore there are not performance critical.

> Methods (and all other objects) are already protected from memory
> corruption and use-after-free by GC. There is no danger in segfaulting
> when old/stale methods get run.
>
> The inline, global (, and perhaps in the future: thread-specific)
> caches will all become expensive if we need to ensure read-after-write
> consistency by checking for changes on methods and constants made
> by other threads.

I've tried to explain little bit in the document, this should not have any overhead. MRI with GIL does not have to ensure read-after-write consistency, other compiling implementations are actively invalidating the compiled code if it depends on a constant which was just redefined (or a method).

I did not entirely understood why you are against volatility on constants and methods, I tried to explain better why they are suggested to be volatile though. Could you elaborate?

> > Threads
> > Threads have the same guarantees as in in Java. Thread.new
> > happens-before the execution of the new thread's block. All operations
> > done by the thread happens-before the thread is joined. In other words,
> > when a thread is started it sees all changes made by its creator and
> > when a thread is joined, the joining thread will see all changes made
> > by the joined thread.
>
> Good. For practical reasons, this should obviate the need for
> constant/method volatility specified above.

It would certainly help if they wouldn't be volatile, require and autoload guaranties as well.

> > Beware of requiring and autoloading in concurrent programs, it's
> > possible to see partially defined classes. Eager loading or blocking
> > until classes are fully loaded should be used to mitigate.
>
> No disagreement, here :)

#9 - 02/04/2016 05:28 PM - Eregon (Benoit Daloze)

Eric Wong wrote:

> I'm not sure if shared memory is even a good model for Ruby (and not my
> decision). Anyways, my comments below if matz/ko1 decide to go down
> this route.

Shared memory at the user level is only one possibility indeed.
But it is one the current model supports, even if MRI prevents actual parallelism.
Also, for a memory model we must take the bottom layer, which seems shared memory here.

> Anyways, I am not convinced (volatile|synchronous|coherent) access
> should happen anywhere by default for anything because of costs.

> I strongly disagree with volatility in method and constant tables. Any
> programs defining methods/constants in parallel threads and expecting
> them to be up-to-date deserve all the problems they get.

The idea is it's only volatile/coherent for storages which are naturally "global"
and where the performance overhead is very limited.

As Petr said, the impact on constants is only for the uncached case,
which is already much slower than a cached constant lookup.

For methods it is very similar as Ruby implementations invalidate
the method caches when the method table is changed, which mean
there is only a minor overhead on method lookup for populating the cache.

On MRI, there is of course no overhead since the GIL guarantees these properties and much more.

> The inline, global (, and perhaps in the future: thread-specific)
> caches will all become expensive if we need to ensure read-after-write
> consistency by checking for changes on methods and constants made
> by other threads.

You are right, inline caches would have overhead on some platforms,
unless some form of safepoints/yieldpoints are available to the VM to clear the caches or ensure visibility
(with a serial number check, it could just ensure visibility of the new serial to every thread).
If the VM actually runs Ruby code in parallel, then it also most likely uses safepoints
for the GC so I would guess Ruby VMs either have them or do not run Ruby code in parallel.

The global cache could use a similar approach to avoid overhead.

With this, the overhead would be limited to the slow path method/constant lookup and the additional cost to invalidate.

#10 - 02/05/2016 09:51 PM - normalperson (Eric Wong)

email@pitr.ch wrote:

> Thank you, for taking time to read it and for your input. I apologise
> for delayed answer, I was rather busy lately.

No worries, I've been busy, too.

> I've added following paragraph to the document explaining a little bit
> why volatility is preferred.
>
> "The volatile property has noticeable impact on performance, on the
> other hand it’s often quite convenient property, since it simplifies
> reasoning about the program. Therefore unless it presents a
> performance issue volatility is preferred."
>
> It tries to be in alignment with rest of the Ruby language to be
> user-friendly. Therefore the volatility behaviour is on Constants and
> similar. I've also elaborate in the document in the Constants part why
> is there no performance loose by making them volatile: "Ruby
> implementations may take advantage of constancy of the variables to
> avoid doing volatile reads on each constant variable read. MRI can
> check a version number.

For MRI, checking a version number still requires a memory model of that
version number to be defined. I'd rather not have consitency guarantees
of the version number.

This goes for constants and methods at least, which are already
versioned in MRI.

> > Maybe volatility for require/autoload is a special case only iff a
> > method/constant is missing entirely; but hitting old methods/constants
> > should be allowed by the implementation.
>
> Volatility of require/autoload and the fact that it blocks when other
> thread is loading given file/constant are very useful in parallel
> environment to make sure that some feature/class is fully loaded
> before using it. Both are usually used only on program paths which run
> only once during loading or reloading, therefore there are not
> performance critical.

Agreed. So perhaps missing constant/method falls back to
(volatile|synchronous|coherent) checking.

However, redefined/included/extended existing constant/methods should
only be eventually consistent; they may be cached locally per-thread.

> The inline, global (, and perhaps in the future: thread-specific) caches will all become expensive if we need to ensure read-after-write consistency by checking for changes on methods and constants made by other threads.

> I've tried to explain little bit in the document, this should not have any overhead. MRI with GIL does not have to ensure read-after-write consistency, other compiling implementations are actively invalidating the compiled code if it depends on a constant which was just redefined (or a method).

MRI has GIL today, I do not want MRI to have a GIL in the future.

To give us the most freedom in the future, I prefer we have as few guarantees as practical about consistency.

> I did not entirely understood why you are against volatility on constants and methods, I tried to explain better why they are suggested to be volatile though. Could you elaborate?

See above :> Any version checks for thread-specific caches would need to define the consistency of the version number itself.

#11 - 02/29/2016 11:06 PM - pitr.ch (Petr Chalupa)

I understand your point, I would like explore how it could be solved in MRI before relaxing the constant and method redefinition though. The relaxation could lead to undesirable unpredictable behaviour for users.

As you've mentioned the version would have to be a volatile (Java) or an atomic (C++11) variable to guarantee that the value is up to date. That would mean volatile read before each method call or constant read, volatile reads are not terribly expensive though. E.g. on x86 it's just a mov instruction (same as regular load) (I am not sure what other platforms MRI targets). Volatile writes are more expensive but that is happening only on rare path, the method or constant redefinition. Without JIT and more optimisations it might have only small or no overhead in MRI, which could be measured in current MRI with GIL just by making the version number atomic (in C terminology). (I am not capable of altering the MRI's source code to measure it though.)

But as Benoit has suggested:

> You are right, inline caches would have overhead on some platforms,
> unless some form of safepoints/yieldpoints are available to the VM to clear the caches or ensure visibility
> (with a serial number check, it could just ensure visibility of the new serial to every thread).
> If the VM actually runs Ruby code in parallel, then it also most likely uses safepoints
> for the GC so I would guess Ruby VMs either have them or do not run Ruby code in parallel.

when MRI has no GIL it will need some king of safepoint to park threads allowing GC to run. It would allow to remove any overhead on the fast path, the version checking. Roughly it would work as follows, a constant redefinition would change the constant, update version number, wait for all threads to reach the safepoint to make sure that all threads will see new version number on next read, finish constant redefinition.

I feel silly for such a late answer, I did not get any email about new comment even though I watch the issue.

#12 - 03/17/2016 06:39 AM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Assigned

- Assignee set to ko1 (Koichi Sasada)

Koichi has some opinions in this area and wants to dump them to this thread. Please go ahead.

#13 - 03/17/2016 08:59 AM - pitr.ch (Petr Chalupa)

Great thanks. I am looking forward to continue the discussion.

#14 - 04/11/2016 07:18 AM - ko1 (Koichi Sasada)

Sorry for late to comment on this topic.
(and sorry i don't read all of comments on this topic)

At first, I need to appreciate you for such a great document.

However, my opinion is negative.
Basically, (at least on MRI) *I* against this proposal because it is too difficult to understand and to implement.

I believe we should introduce memory consistency model on more higher-level abstraction, like Go language does.

Difficulty of understanding

For example, JSR-133 is well documented by great people.
But I'm not sure how many people understand it correctly very details (and evils are in details).

To make it easy, we need to introduce more clear, more little rules with higher level abstraction.

Difficulty of implementation

As you know, there are various computer architectures enabling shared memory parallel computing with different memory consistency model.
I'm not sure we can enable to implement on all of them.

For example (trivial example), your all "atomicity" fields are true,
but I'm not sure how to implement them correctly on Float value (as you write in the middle of this document) on any computer architecture.

As you know, some computers reorder memory access.
To serialize them, we need to issue extra instructions.
Maybe we need to issue them many times if we need to satisfies all of them.

Also strict rules will become hurdles for future optimizations.

On MRI, we don't need to care with such memory access reordering
because MRI uses pthread_mutex (or similar API) on switching.

Note

This is my opinion, and also Matz had agreed with this opinion.

However, it is only personal opinion.
We need to discuss about it.
So that your contribution is great.

#15 - 04/20/2016 05:40 PM - pitr.ch (Petr Chalupa)

Thank you for responding and for taking time to read the proposal.

Let me start by elaborating more on the motivation behind all of the related proposals, since I did not really explained it in detail when I was opening them. I apologise for not doing that sooner.

Motivation

I would like to clear up a possible misunderstanding about the target users of this document and this memory model. It's not intended to be directly used by majority of the Ruby programmers. (Even though the document aims to be understandable it will still be difficult topic.) It's intended to be used by concurrency enthusiasts, giving them tools to build many different concurrency abstractions as gems.

At this point Ruby is a general purpose language, with direct support for Threads and shared memory. As it was announced in few presentations, there are plans to add new easy to use abstraction to Ruby in some future release and maybe deprecate Threads. Lets call this scenario A. (Block-quotes are used for better logical structure.)

> (A) I understand the need to add such abstraction (actors, channels, other?) to Ruby to enable Ruby users to build concurrent applications with ease. For future reference let's call the one future abstraction Red. The Red would then have well documented and defined behaviour in concurrent and parallel execution (This is what I think you are referring to). However providing just one abstraction in standard library (and deprecating Threads) will hurt usability of Ruby language.
>
> The problems lies in that there is no single concurrency abstraction which would fit all problems. Therefore providing just Red will left Ruby language suitable to just some subset of problems.
>
> Assuming: Only the Red would be documented and providing high-level guarantees; threads would be deprecated; low-level concurrency would not be documented and guaranteed. Developers who would like to contribute new

> abstraction to solve another group of problems would be left with following (I think not very good) choices:
>
> > (1) Implement the abstraction in underlying language used for the particular Ruby implementation (in C for MRI, in Java for JRuby(+Truffle)) using guarantees provided by the underlying language. Meaning the author of the new abstraction has to understand 3 programming languages (C, Ruby, Java) and 3 implementations to develop the implementation 3 times. That would discourage people and also make the whole process error prone and difficult.
> >
> > (2) Implement the abstraction using the Red. This approach gives users the desired abstraction (avoiding using different languages and understanding implementation details) but it will probably have bad performance since the Red is not suited to solve this problem. For example implementing ConcurrentHashMap (allowing parallel reads and writes) with actors would perform badly. (Admittedly this is a little extreme example, but it demonstrates the problem and I could not think of a better one.)

The above is to best of my knowledge where Ruby is heading in future, please correct me if I misunderstood and/or misrepresented it in any way.

To avoid the above outlined difficulties Ruby could take a different path, which is related to these proposals (or theirs evolved successors).

> (B) Ruby would stay general purpose language with direct threads support and shared memory with documented memory model. The low-level documentation would allow people (who are interested) to write different concurrent abstractions efficiently. One of them would become the standard and preferred way how to deal with concurrency in Ruby. Let's call it Blue. The Blue abstraction would (as Red would) be part of the standard library. Same as Red it would have well documented and defined behaviour in concurrent and parallel execution, but in this case based on the lower-level model. The documentation would be directed at all Ruby users and made as easy to understand as possible.
>
> Majority of the Ruby users would use Blue the go-to abstraction as they would use the Red in scenario A. The key difference is that there is the low-level model to support creation of new abstractions. Therefore when the Blue cannot solve a particular issue a Ruby user can pick a different concurrency abstraction created by somebody else and provided as a gem or create a new one.

I believe this would make the Ruby language more flexible.

Difficulty of understanding

This is something I believe can be improved over time. Also as mentioned above it's not intended to be used be everyone. Could you point me to parts which are not understandable, or lack explanation? I would like to improve them, to make the document more comprehensible.

The document is intentionally not as detailed and formal as JSR-133, to keep understandability. The price is as you say and I agree in details and omissions which may be left unspecified. I believe the high-level documentation for the Red will unfortunately suffer the same problem of evil details.

If the memory model is reviewed by many people and given some time to mature, I believe it will cover majority of the situations, omitted corner-cases can be fixed later. I think the current situation is much worse when each implementation has different rules and any document will improve the situation greatly.

Difficulty of implementation

(Various architectures) I am not a C programmer so I am not that well informed but I believe that in C this is solved by C11 standard and before that by various libraries. Can MRI use C11 in future when it'll be dropping GIL?

(Atomic Float) I agree that it is more difficult when Floats are required to by atomic, but if they were not it would be quite a surprising to Ruby users that

a simple reference assignment of a Float object (as it's represented in Ruby) is not atomic. Therefore this is chosen to be atomic purely not to surprise users and to avoid educating users about torn reads/writes. Same applies to Fixnum which is bigger than int and fits into long (using Java primitive names here). Even though this is more difficult I think it makes sense to protect users from concerning about torn reads/writes. The implementation itself should be trivial on all 64-bit platforms, only 32-bit platforms will require some tricks. This [1] post suggests that it can be done.

(Strict rules) The document tries to be balanced between restricting optimisation and creating ugly surprises for users. I am expecting there will be more discussion about the rules:

- How to implement it on all Ruby implementations?
- Will it prevent any optimisations?
- Will it expose unexpected behaviour to users?

The document is really just a first draft and everything is open for discussion and improvement which I both hope for. It was prepared not to limit any of the Ruby implementations, but problems can be missed, if it turns out a rule is too strict it can be relaxed.

(MRI with GIL) yes MRI already provides all of the guaranties specified thanks to GIL. It's even stronger. On the other hand if I understand correctly MRI is looking for ways how to remove GIL and the fact that GIL provides stronger undocumented guarantees makes this difficult. Users rely on it (intentionally or unintentionally) even though they shouldn't. Having a document describing what is guaranteed and what not, may make easier transition to MRI without GIL in future.

In Conclusion

I hope that maybe I've changed your mind a little bit about the B scenario and this proposal, that we could discuss more the issues this model could bring for MRI. I would like to help to solve them or avoid them by relaxing rules.

I believe that if this model (or its evolved successor) is accepted in all Ruby implementations over time, it will help the Ruby language a lot to be prepared for concurrency and parallelism, which is nowadays non-optional.

[1] http://shipilev.net/blog/2014/all-accesses-are-atomic/

#16 - 04/30/2016 07:30 PM - ko1 (Koichi Sasada)

Sorry for late response.

Petr Chalupa wrote:

> Let me start by elaborating more on the motivation behind all of the related proposals, since I did not really explained it in detail when I was opening them. I apologise for not doing that sooner.

No problem. Thank you for your explanation.

> ## Motivation
>
> I would like to clear up a possible misunderstanding about the target users of this document and this memory model. It's not intended to be directly used by majority of the Ruby programmers. (Even though the document aims to be understandable it will still be difficult topic.) It's intended to be used by concurrency enthusiasts, giving them tools to build many different concurrency abstractions as gems.

I (may) understand what you want to say.
As you wrote:

> Even though the document aims to be understandable it will still be difficult topic

I agree with that, and I believe most of us can't understand and guarantee all of specifications.
At least I don't believe I can implement it.
(Of course, it is because of my low skill. Somebody should be able to implement it)

> At this point Ruby is a general purpose language, with direct support for Threads and shared memory. As it was announced in few presentations, there are plans to add new easy to use abstraction to Ruby in some future release and maybe deprecate Threads. Lets call this scenario A. (Block-quotes are used for better logical structure.)
>
> > (A) I understand the need to add such abstraction (actors, channels, other?) to Ruby to enable Ruby users to build concurrent applications with ease. For future reference let's call the one future abstraction Red. The Red would then have well documented and defined behaviour in concurrent and parallel execution (This is what I think you are referring to). However providing just one abstraction in standard library (and deprecating Threads) will hurt usability of Ruby language.
> >
> > The problems lies in that there is no single concurrency abstraction which would fit all problems. Therefore providing just Red will left Ruby language suitable to just some subset of problems.
> >
> > Assuming: Only the Red would be documented and providing high-level guarantees; threads would be deprecated; low-level concurrency would not be documented and guaranteed. Developers who would like to contribute new abstraction to solve another group of problems would be left with following (I think not very good) choices:

I agree the flexibility should be decreased.

> > (1) Implement the abstraction in underlying language used for the particular Ruby implementation (in C for MRI, in Java for JRuby(+Truffle)) using guarantees provided by the underlying language. Meaning the author of the new abstraction has to understand 3 programming languages (C, Ruby, Java) and 3 implementations to develop the implementation 3 times. That would discourage people and also make the whole process error prone and difficult.
> >
> > (2) Implement the abstraction using the Red. This approach gives users the desired abstraction (avoiding using different languages and understanding implementation details) but it will probably have bad performance since the Red is not suited to solve this problem. For example implementing ConcurrentHashMap (allowing parallel reads and writes) with actors would perform badly. (Admittedly this is a little extreme example, but it demonstrates the problem and I could not think of a better one.)
>
> The above is to best of my knowledge where Ruby is heading in future, please correct me if I misunderstood and/or misrepresented it in any way.
>
> To avoid the above outlined difficulties Ruby could take a different path, which is related to these proposals (or theirs evolved successors).

I understand your concerns. I agree there are such disadvantages.

However, I believe productivity by avoiding shared-everything will help programmers.

For (1), I agree there is such difficulties.
I don't have any comment on it.
Yes, there is.

For (2), you mentioned about performance.
However, I believe Ruby should contribute programmer's happiness.
I believe performance is not a matter.

It seems strange because parallelism is for performance.
I assume such drawback can be overcame with (a) design patterns (b) parallelism (# of cores).

I also propose problem issue (3).
We need more time to discuss to introduce new abstraction.
(The biggest problem is I couldn't propose Red specifications)
We need more learning cost and need to invent efficient patterns using Red.

Thread model is well known (I don't say thread model is easy to use :p).
This is clearly advantage of thread model.

I agree there are many issues (1 to 3, and moer).
But I believe the productivity by simplicity is most important (for me, a ruby programmer).

> (B) Ruby would stay general purpose language with direct threads support and shared memory with documented memory model. The low-level documentation would allow people (who are interested) to write different concurrent abstractions efficiently. One of them would become the standard and preferred way how to deal with concurrency in Ruby. Let's call it Blue. The Blue abstraction would (as Red would) be part of the standard library. Same as Red it would have well documented and defined behaviour in concurrent and parallel execution, but in this case based on the lower-level model. The documentation would be directed at all Ruby users and made as easy to understand as possible.
>
> Majority of the Ruby users would use Blue the go-to abstraction as they would use the Red in scenario A. The key difference is that there is the low-level model to support creation of new abstractions. Therefore when the Blue cannot solve a particular issue a Ruby user can pick a different concurrency abstraction created by somebody else and provided as a gem or create a new one.

I believe this would make the Ruby language more flexible.

I agree it is flexible.
However it will be error prone if shared-everything model is allowed.

Difficulty of understanding

> This is something I believe can be improved over time. Also as mentioned above it's not intended to be used be everyone. Could you point me to parts which are not understandable, or lack explanation? I would like to improve them, to make the document more comprehensible.
>
> The document is intentionally not as detailed and formal as JSR-133, to keep understandability. The price is as you say and I agree in details and omissions which may be left unspecified. I believe the high-level documentation for the Red will unfortunately suffer the same problem of evil details.
>
> If the memory model is reviewed by many people and given some time to mature, I believe it will cover majority of the situations, omitted corner-cases can be fixed later. I think the current situation is much worse when each implementation has different rules and any document will improve the situation greatly.

To point out, I need to read more carefully and try to implement with parallel threads.
(evils will be in implementation details)

Difficulty of implementation

> (Various architectures) I am not a C programmer so I am not that well informed but I believe that in C this is solved by C11 standard and before that by various libraries. Can MRI use C11 in future when it'll be dropping GIL?

Not sure, sorry.
From CPU architecture, there are several overhead for strong memory consistency.

> (Atomic Float) I agree that it is more difficult when Floats are required to by atomic, but if they were not it would be quite a surprising to Ruby users that a simple reference assignment of a Float object (as it's represented in Ruby) is not atomic. Therefore this is chosen to be atomic purely not to surprise users and to avoid educating users about torn reads/writes. Same applies to Fixnum which is bigger than int and fits into long (using Java primitive names here). Even though this is more difficult I think it makes sense to protect users from concerning about torn reads/writes. The implementation itself should be trivial on all 64-bit platforms, only 32-bit platforms will require some tricks. This [1] post suggests that it can be done.

I assume that there are pros and cons about performance.

Shared everything model (thread-model)

- Pros. we can share everything easily.
- Cons. requires fine-grain consistency control for some data structures to guarantee memory model.

Shared nothing model (Red):

- Pros. Do not need to care fine grain memory consistency
- Cons. we can't implement shared data structures in Ruby (sometimes, it can be performance overhead).

> (Strict rules) The document tries to be balanced between restricting optimisation and creating ugly surprises for users. I am expecting there will be more discussion about the rules:
>
> - How to implement it on all Ruby implementations?
> - Will it prevent any optimisations?
> - Will it expose unexpected behaviour to users?
>
> The document is really just a first draft and everything is open for discussion and improvement which I both hope for. It was prepared not to limit any of the Ruby implementations, but problems can be missed, if it turns out a rule is too strict it can be relaxed.
>
> (MRI with GIL) yes MRI already provides all of the guaranties specified thanks to GIL. It's even stronger. On the other hand if I understand correctly MRI is looking for ways how to remove GIL and the fact that GIL provides stronger undocumented guarantees makes this difficult. Users rely on it (intentionally or unintentionally) even though they shouldn't. Having a document describing what is guaranteed and what not, may make easier transition to MRI without GIL in future.

I agree Ruby programmers can rely on GIL guarantees (and it is not good for other implementation).

BTW, such strong GIL guarantee helps people from some kind of thread-safety bugs.
("help" means decreasing bug appearance rate. As you wrote, it is also "bad" thing)

> ## In Conclusion
>
> I hope that maybe I've changed your mind a little bit about the B scenario and this proposal, that we could discuss more the issues this model could bring for MRI. I would like to help to solve them or avoid them by relaxing rules.
>
> I believe that if this model (or its evolved successor) is accepted in all Ruby implementations over time, it will help the Ruby language a lot to be prepared for concurrency and parallelism, which is nowadays non-optional.
>
> [1] http://shipilev.net/blog/2014/all-accesses-are-atomic/

I don't change my mind.
I believe simplicity is more important than flexibility.

However, your comments clear many kind of things.
I agree that many people agree with you.

Again, my comment is only my thoughts.
I don't against B scenario for other implementations, and for MRI if someone contribute.

Actually, sometime Matz said he want to go B scenario.
He proposed Actors on threads (people should care to modify objects inter actors (threads)).
Same approach of Cellroid.
But I'm against on it :p (and Matz said he agree with me, when I asked. I'm not sure current his idea)

#17 - 05/19/2016 09:02 PM - pitr.ch (Petr Chalupa)

Koichi Sasada wrote:

> Sorry for late response.
>
> Petr Chalupa wrote:
>
> > Let me start by elaborating more on the motivation behind all of the related proposals, since I did not really explained it in detail when I was opening them. I apologise for not doing that sooner.

No problem. Thank you for your explanation.

> ## Motivation
>
> I would like to clear up a possible misunderstanding about the target users of this document and this memory model. It's not intended to be directly used by majority of the Ruby programmers. (Even though the document aims to be understandable it will still be difficult topic.) It's intended to be used by concurrency enthusiasts, giving them tools to build many different concurrency abstractions as gems.

I (may) understand what you want to say.
As you wrote:

> Even though the document aims to be understandable it will still be difficult topic

I agree with that, and I believe most of us can't understand and guarantee all of specifications.
At least I don't believe I can implement it.
(Of course, it is because of my low skill. Somebody should be able to implement it)

Luckily there are other languages with their memory models, where the guaranties are already provided. Their working solutions can be reused and applied in MRI.

> At this point Ruby is a general purpose language, with direct support for Threads and shared memory. As it was announced in few presentations, there are plans to add new easy to use abstraction to Ruby in some future release and maybe deprecate Threads. Lets call this scenario A. (Block-quotes are used for better logical structure.)
>
> > (A) I understand the need to add such abstraction (actors, channels, other?) to Ruby to enable Ruby users to build concurrent applications with ease. For future reference let's call the one future abstraction Red. The Red would then have well documented and defined behaviour in concurrent and parallel execution (This is what I think you are referring to). However providing just one abstraction in standard library (and deprecating Threads) will hurt usability of Ruby language.
> >
> > The problems lies in that there is no single concurrency abstraction which would fit all problems. Therefore providing just Red will left Ruby language suitable to just some subset of problems.
> >
> > Assuming: Only the Red would be documented and providing high-level guarantees; threads would be deprecated; low-level concurrency would not be documented and guaranteed. Developers who would like to contribute new abstraction to solve another group of problems would be left with following (I think not very good) choices:

I agree the flexibility should be decreased.

Just to make sure we understand each other. I agree that the flexibility should be decreased for users so they can write their concurrent code with ease. I believe we disagree on which level it should be achieved on though. I am advocating for library level, you I think for language level.

> > (1) Implement the abstraction in underlying language used for the particular Ruby implementation (in C for MRI, in Java for JRuby(+Truffle)) using guarantees provided by the underlying language. Meaning the author of the new abstraction has to understand 3 programming languages (C, Ruby, Java) and 3 implementations to develop the implementation 3 times. That would discourage people and also make the whole process error prone and difficult.
> >
> > (2) Implement the abstraction using the Red. This approach gives users the desired abstraction (avoiding using different languages and understanding

> implementation details) but it will probably have bad performance since the Red
> is not suited to solve this problem. For example implementing ConcurrentHashMap
> (allowing parallel reads and writes) with actors would perform badly.
> (Admittedly this is a little extreme example, but it demonstrates the problem
> and I could not think of a better one.)

> The above is to best of my knowledge where Ruby is heading in future, please
> correct me if I misunderstood and/or misrepresented it in any way.
>
> To avoid the above outlined difficulties Ruby could take a different path,
> which is related to these proposals (or theirs evolved successors).

I understand your concerns. I agree there are such disadvantages.

However, I believe productivity by avoiding shared-everything will help programmers.

For (1), I agree there is such difficulties.
I don't have any comment on it.
Yes, there is.

For (2), you mentioned about performance.
However, I believe Ruby should contribute programmer's happiness.
I believe performance is not a matter.

It seems strange because parallelism is for performance.
I assume such drawback can be overcame with (a) design patterns (b) parallelism (# of cores).

I am using Ruby for 10 years (thank You!) and I see and understand the big benefit of Ruby caring about programmer's happiness. I care about it very much too and I try to avoid any suggestions which would lead to sacrificing it. I think that so far all of the proposals were shaped by user happiness and performance. (For example: the discussion about volatile constants in this issue, current rules are harder to implement but better for users.) If it's not true I would like to fix it.

Regarding (2), Users may sacrifice some performance but in this case it might perform quite badly. Few examples for consideration follow:

(clojure agents) Implementation of agents using actors: since agent has to be able to report it's value at any time it would need to be modeled using at least 2 actors: one to hold and report the value, second to process the updates.

(go channels) Implementing go sized channel using actors: The channel is blocking. The channel is represented with one or two actors. One is simpler but has higher contention, using two actors it avoids some contenting between head and tail of the channel. To simulate blocking: actors, which are sending messages to the channel, will not continue with other message processing until they receive confirmation from the channel that they can continue, that they are not blocked. Actors waiting on messages from channel would have to send challenge to the channel that they want to receive a message from channel and do not process other messages until they receive the message from channel.

My intuition is that the slowdown will be 2x and *more* (I'll do some tests).
The outlined implementations are much more complex compared to the conventional implementation using shared memory.

It also touches another issue, for some problems just one abstraction will inevitably lead to some awkward usage patterns and unnecessary complexity for users, where the Red abstraction does not provide any natural way of support for solving the problem.

(actor future state) Staying with the hypothetic actors as Red example for one more paragraph to support previous claim. Supposing there is an application with some state and background processing. Actors support state and events generated based on the state changes naturally. However they are not the best choice to model background processing. An actor doing a background job isn't responsive to any messages during the execution, therefore the first step is to always break up state actors and it's background processing to two actors. Then the actor responsible for background processing is just a wrapper around an asynchronously executed function without any state, which might be better modeled by just a block executed on a thread-pool for IO tasks or by a Future

object. Another issue could arise if one general actor is used to process all the background jobs (which is a good thought at first glance), the actor will become bottleneck allowing to execute just one background job at a time (tasks with blocking IO can also easily deadlock it). Easy fix is to introduce a pool of actors to process background jobs, but then again they will be slower than shared-memory thread-pool implementation.

Of course all of these examples were for actors not for Red. It's not directly applicable, but it shows what kind of problems can be (I think unavoidably) anticipated for Red.

It's not always possible to just throw more cores at the problem, the algorithm has to support such scaling.

Going back to user happiness, the scenario A sacrifices happiness of some users:

(group1) Concurrent library implementers, because of (1) and (2). This is probably not a biggest group of users but I think it's an important group since their work will be used by *many* users.

(group2) Second group is larger, it's users which would like to use Ruby to solve a problem where Red would be of limited help. These users will be looking for alternative solution and will be disappointed that the choice will be severely limited, because group1 will not write new abstractions (admittedly this is just my projection).

Therefore scenario A does not have just positive impact on user happiness (those users, whose problems fit well to be solved by Red (probably bigger group than group1 and group2 combined)).

Since Ruby is nowadays mostly used in long running processes not in scripts the performance becomes more important. In my observation performance is the most common reason why people leave Ruby, not because they are unhappy with the language but because they pay too much for their servers to run their applications.

> I also propose problem issue (3).
> We need more time to discuss to introduce new abstraction.
> (The biggest problem is I couldn't propose Red specifications)
> We need more learning cost and need to invent efficient patterns using Red.
>
> Thread model is well known (I don't say thread model is easy to use :p).
> This is clearly advantage of thread model.
>
> I agree there are many issues (1 to 3, and moer).
> But I believe the productivity by simplicity is most important (for me, a ruby programmer).

It looks like for the purpose of this discussion I should know more about what is considered to become Red. Later you mention sharing nothing, how would that work for classes, constants, method definitions etc.? How would the isolated parts communicate with each other, deep freezing or copying the messages? Are there any sources like talks or issues I could read?

When I first heard about Red being planned I was thinking about deep-freezing or deep-cloning to ensure messages cannot lead to shared memory issues, Red being actors, channels etc. and isolation achieved only by convention and user education.

Yeah (3) will take time, that's a common problem for both A and B scenarios. B might be in a better situation though because more people can get involved writing more abstractions until a winning one is picked and becomes part of Ruby standard library.

> > (B) Ruby would stay general purpose language with direct threads support and shared memory with documented memory model. The low-level documentation would allow people (who are interested) to write different concurrent abstractions efficiently. One of them would become the standard and preferred way how to deal with concurrency in Ruby. Let's call it Blue. The Blue abstraction would (as Red would) be part of the standard library. Same as Red it would have well documented and defined behavior in concurrent and parallel execution, but in this case based on the lower-level model. The documentation would be directed at all Ruby users and made as easy to understand as possible.

> > Majority of the Ruby users would use Blue the go-to abstraction as they would use the Red in scenario A. The key difference is that there is the low-level model to support creation of new abstractions. Therefore when the Blue cannot solve a particular issue a Ruby user can pick a different concurrency abstraction created by somebody else and provided as a gem or create a new one.
>
> I believe this would make the Ruby language more flexible.

I agree it is flexible.
However it will be error prone if shared-everything model is allowed.

Currently Ruby has shared memory, how would that be taken away? It would be *huge* incompatible change, I believe.

Yeah it is difficult to use, but I would like to stress that it's only for group1 (mentioned above). Most of the users would not have to deal with it since they'll use just one of the available abstractions (blue being the most common one and advised by Ruby to be used).

> ### Difficulty of understanding
>
> This is something I believe can be improved over time. Also as mentioned above it's not intended to be used be everyone. Could you point me to parts which are not understandable, or lack explanation? I would like to improve them, to make the document more comprehensible.
>
> The document is intentionally not as detailed and formal as JSR-133, to keep understandability. The price is as you say and I agree in details and omissions which may be left unspecified. I believe the high-level documentation for the Red will unfortunately suffer the same problem of evil details.
>
> If the memory model is reviewed by many people and given some time to mature, I believe it will cover majority of the situations, omitted corner-cases can be fixed later. I think the current situation is much worse when each implementation has different rules and any document will improve the situation greatly.

To point out, I need to read more carefully and try to implement with parallel threads.
(evils will be in implementation details)

Thanks a lot, I really appreciate that you are looking at it in more detail and that you are willing to discuss this in length.

> ### Difficulty of implementation
>
> (Various architectures) I am not a C programmer so I am not that well informed but I believe that in C this is solved by C11 standard and before that by various libraries. Can MRI use C11 in future when it'll be dropping GIL?

Not sure, sorry.
From CPU architecture, there are several overhead for strong memory consistency.

Yeah there are, but comparing GIL vs noGIL, running on all cores with some slight overhead is advantageous.

> (Atomic Float) I agree that it is more difficult when Floats are required to by atomic, but if they were not it would be quite a surprising to Ruby users that a simple reference assignment of a Float object (as it's represented in Ruby) is not atomic. Therefore this is chosen to be atomic purely not to surprise users and to avoid educating users about torn reads/writes. Same applies to Fixnum which is bigger than int and fits into long (using Java primitive names here). Even though this is more difficult I think it makes sense to protect users from concerning about torn reads/writes. The implementation itself should be trivial on all 64-bit platforms, only 32-bit platforms will require some tricks. This [1] post suggests that it can be done.

Atomic float and C11: as far as I know if it's declared as atomic float but operations load and write are done with memory_order_relaxed then it keeps atomicity property without any ordering constrains, therefore without performance overhead (there might be exceptions but even 32bit platforms can use some tricks like SSE instruction to make the float atomic without overhead).

> I assume that there are pros and cons about performance.
>
> Shared everything model (thread-model)
>
> - Pros. we can share everything easily.
> - Cons. requires fine-grain consistency control for some data structures to guarantee memory model.
>
> Shared nothing model (Red):
>
> - Pros. Do not need to care fine grain memory consistency
> - Cons. we can't implement shared data structures in Ruby (sometimes, it can be performance overhead).

I am sorry, but I'm not sure how to interpret the comparison. It's important to distinguish where does the pros. and cons. apply. In thread-model case, Pros. applies to the users and Cons. applies to the Ruby implementers. In Red, Pros. applies to the implementers and Cons. to the Ruby users. Shared everything comes out better in this comparison emphasizing users.

Regarding the implementor's point of view, I appreciate the amount of work and complexity this will be creating. I am part of the JRuby+Truffle team and we would have to comply and deal with RMM too. Still I believe it's worth the effort.

> (Strict rules) The document tries to be balanced between restricting optimisation and creating ugly surprises for users. I am expecting there will be more discussion about the rules:
>
> - How to implement it on all Ruby implementations?
> - Will it prevent any optimisations?
> - Will it expose unexpected behaviour to users?
>
> The document is really just a first draft and everything is open for discussion and improvement which I both hope for. It was prepared not to limit any of the Ruby implementations, but problems can be missed, if it turns out a rule is too strict it can be relaxed.
>
> (MRI with GIL) yes MRI already provides all of the guaranties specified thanks to GIL. It's even stronger. On the other hand if I understand correctly MRI is looking for ways how to remove GIL and the fact that GIL provides stronger undocumented guarantees makes this difficult. Users rely on it (intentionally or unintentionally) even though they shouldn't. Having a document describing what is guaranteed and what not, may make easier transition to MRI without GIL in future.

> I agree Ruby programmers can rely on GIL guarantees (and it is not good for other implementation).

Yeah, alternative implementations may suffer the issue of users relying on GIL already. In practice it may not be that bad though, at least in code which is meant to be run concurrently or on parallel. These libraries tend to use slower Mutexes to stay safe, because instance variables do not have precisely defined behavior. (This is just my personal view, we should ask Charles and Tom how often this came up in their issues.)

> BTW, such strong GIL guarantee helps people from some kind of thread-safety bugs. ("help" means decreasing bug appearance rate. As you wrote, it is also "bad" thing)

In Conclusion

> I hope that maybe I've changed your mind a little bit about the B scenario and this proposal, that we could discuss more the issues this model could bring for MRI. I would like to help to solve them or avoid them by relaxing rules.
>
> I believe that if this model (or its evolved successor) is accepted in all Ruby implementations over time, it will help the Ruby language a lot to be prepared for concurrency and parallelism, which is nowadays non-optional.

[1] http://shipilev.net/blog/2014/all-accesses-are-atomic/

> I don't change my mind.
> I believe simplicity is more important than flexibility.

I am of the same opinion simplicity is important for users, however I think we (whole Ruby community no matter the implementation) could have both simplicity and flexibility.

> However, your comments clear many kind of things.
> I agree that many people agree with you.
>
> Again, my comment is only my thoughts.
> I don't against B scenario for other implementations, and for MRI if someone contribute.

To sum up regarding contribution, headius was so kind and offered to work on the accompanying proposals since he has an experience with C which I have only limited. I think the current form of the Ruby Memory Model fits MRI with GIL so no contribution should be needed.

I suppose you meant contributing a work on removing GIL and ensuring RMM compliance?

Benoit Daloze (eregon) and I will gladly help to find solutions if needed.

> Actually, sometime Matz said he want to go B scenario.
> He proposed Actors on threads (people should care to modify objects inter actors (threads)).
> Same approach of Cellroid.
> But I'm against on it :p (and Matz said he agree with me, when I asked. I'm not sure current his idea)

We could also scope down the discussion to just the most important parts of RMM, which (I think) are local and instance variables, rest of the related proposals are mostly related to them.

I've also posted a comment to https://bugs.ruby-lang.org/issues/12021, it provides an example of how the low-level model could be used to support a simple and nice high-level behavior of Proc.

#18 - 06/14/2016 02:19 AM - headius (Charles Nutter)

I have had a quick read through comments on this issue, and I have a few responses. Sorry for my late arrival...I had not realized there was this much discussion happening :-)

I think my position on this boils down to three things:

1. Ruby currently has a shared-memory, thread-based concurrency system.
2. There's no memory model documented for that system.
3. There needs to be such a model.

Anything outside these discussion points seems irrelevant to me. Yes, Ruby 3 may have some new concurrency model, some day. Not even matz knows what it will be. That *possible* future can't be used as a point against fixing specification gaps in the *current* model. And waiting until 2020 (most recent estimate for Ruby 3 that I've heard) to formally publish a memory model for Ruby seems unreasonable.

Ruby has needed this memory model formality since we started working on JRuby ten years ago. We had to unilaterally declare a memory model for JRuby in order to reconcile Ruby's lack of a memory model with Java's explicit and strict model.

We don't want to be the only Ruby 2.x implementation that has a memory model, especially if it may conflict with the realities of CRuby. Therefore we need to do something today.

As I understand it, the model described by this document does not force many (or any?) changes on CRuby. CRuby would be able to meet all or most of the requirements of the memory model by having the global lock, but other implementations without a lock would be able to run code in parallel without breaking user expectations.

Having an explicit memory model would actually *help* people avoid shared-memory problems. Right now, without a memory model, we can't safely build the sorts of concurrency primitives people need. If we can get some formality in Ruby *today* we can build Actors, Futures, Channels, concurrent lock-free collections, and more...using nothing but Ruby. That's what we want...to empower Ruby the language to solve the concurrency problems of today's Rubyists.

As an example...

I have done development on the JVM for the past 20 years. I have only had to write explicit threaded code since I started working on JRuby. Java

users don't Thread.new...they spin up executors, wait on futures, message actors. All the utilities and patterns discussed here are supported by Java and the JDK classes low level, shared-memory, threading primitives backed by a strong memory model. If we had such a memory model for Ruby, we'd be able to provide the same features *efficiently* and make it *less likely* that people would be using Thread directly.

I think it's valuable to be able to implement Ruby's future concurrency models in Ruby, isn't it? That won't happen without a memory model.

A few specific responses:

ko1:

> However it will be error prone if shared-everything model is allowed.

Shared-everything is how Ruby works today. We want a solution for today's Ruby. (Petr said something similar above.) And to repeat my last point...shared-everything would be a whole lot easier for Ruby folks to deal with if they had the kinds of memory guarantees and data structures and concurrency APIs that Java folks take for granted. We're making things MUCH MUCH WORSE by not having a model in place.

> Shared everything model (thread-model)
>
> - Pros. we can share everything easily.
> - Cons. requires fine-grain consistency control for some data structures to guarantee memory model.

I've already pointed out that this is what we have today, without any formality. But again, having a memory model means we could build those better abstractions *in Ruby* so people don't have to worry about fine-grained consistency themselves.

> I believe performance is not a matter.

It is when it is. People leave CRuby (or Ruby the language) when it becomes important to use up all the cores or run straight-line code really fast. I think we want a language that makes programmers happy *both* when programming *and* when running that code, don't we? Programmer happiness is about more than just having a nice language...it's also about having that language's runtime meet your needs.

> Basically, (at least on MRI) I against this proposal because it is too difficult to understand and to implement.

We want to help you understand this proposal, and I want to help implement any changes that are needed! :-)

> I believe we should introduce memory consistency model on more higher-level abstraction, like Go language does.

That sounds great, but until it happens we need to fix this gaping hole in Ruby's documentation/specification. We have threads today. We can't build better abstractions above threads without a memory model. But we *can* build better abstractions once a model is in place, making it *less* likely people will stumble over threads.

Eric Wong:

> To give us the most freedom in the future, I prefer we have as few guarantees as practical about consistency.

This is probably the most dangerous way to go of all. Threads are here, they're exposed to Ruby, and they need a solution now. People already rely on Ruby's implicit memory model (whether they realize it or not), and they turn to JRuby's explicit memory model when they run on many cores. We're mostly asking that the implicit become explicit.

This doesn't limit any freedom in the future either...especially when people are talking about even more drastic solutions like removing Thread. Fixing Ruby 2.x threading and memory model does not mean Ruby 3 can't change.

> I strongly disagree with volatility in method and constant tables. Any programs defining methods/constants in parallel threads and expecting them to be up-to-date deserve all the problems they get.

I agree to a point...and that point is autoload. With the presence of autoload in Ruby, there will *always* be concurrent modifications of classes happening. I have proposed previously that all requires should be using the SAME lock, so you can't have code running in one load that's affected by code running in another load. I have also proposed that reopening a class should not publish its changes until the class is closed again. Both solutions help the problem, neither was accepted.

Autoload is the problem here, not users loading code in parallel by themselves.

Eric also says:

> The inline, global (, and perhaps in the future: thread-specific) caches will all become expensive if we need to ensure read-after-write consistency by checking for changes on methods and constants made by other threads.

JRuby does this now, and runs most code much faster than in CRuby. Cache coherency for VM-level structures does not mean you have to sacrifice

performance.

...

Sorry my responses are out of order from the comments...and I apologize again for not getting involved earlier. I believe with all my heart we need to make this move for today's Ruby (and today's Rubyists), regardless of what tomorrow's Ruby may or may not do. Please help us!

#19 - 08/17/2016 09:03 AM - pitr.ch (Petr Chalupa)

I am going to RubyKaigi, I would be very interested to have a meeting face to face there to discuss this topic in depth. Koichi, would you be open to spend some time discussing it? Would anybody else be interested? Afaik Charles is not going, Tom Enebo is, I'll ask him if he would be interested, to bring JRuby's perspective to the discussion.

#20 - 08/17/2016 11:55 PM - ko1 (Koichi Sasada)

> Koichi, would you be open to spend some time discussing it?

Sure! Other than my presentation time, I'll be happy to discuss about it.

Thanks,
Koichi

#21 - 09/23/2016 10:23 AM - shyouhei (Shyouhei Urabe)

Can someone summarize the off-line discussion mentioned earlier? I was not there.

#22 - 09/23/2016 10:43 AM - pitr.ch (Petr Chalupa)

Hi Shyouhei, sorry for not doing it so far, I took vacation to travel Japan right after the conference. It is already on my todolist when I get back.

#23 - 09/24/2016 02:25 AM - shyouhei (Shyouhei Urabe)

Great. I'm looking forward.

#24 - 09/29/2016 07:36 AM - pitr.ch (Petr Chalupa)

As the previous comments mention we had a meeting to discus memory model at RubyKaigi. There were about fifteen Ruby implementers siting around the table from MRI, JRuby, JRuby+Truffle, and OMR.

The first meeting started by exchanging little bit of background information. We then freely discussed few topics:

- why are volatile variables needed and some examples
- concurrency models, why is shared memory sometimes convenient and needed
- how would be volatile variables made available in Ruby (in every object vs by a proxy)

The topics were useful to exchange background and to understand each other better, however we did not reach any agreement on any of the topics. We realised that we need some real examples to discuss.

For the next meeting Benoit had prepared code examples that better illustrate the problems faced by language implementers without a memory model in place. In particular, we focused on the problems that affect instance variables in shared Ruby objects. When operations like: instance variable update, new instance variable definition, instance variable type profile change are executed on an object concurrently and/or in parallel the following must not happen:

- An update of an instance variable is lost (see https://docs.google.com/document/d/1pVzU8w_QF44YzUCCab990Q_WZOdhpKolClHaiXG-sPw/edit#heading=h.eukpz1zhpmm2)
- A value out-of-thin-air is read

This helped us to reach agreement that **we need a minimal memory model** which would forbid exactly this type of surprising behaviour. Personally I think this is great.

The same day we had followup meeting with reduced number of attendees Koichi, Benoit and me. We've discussed in detail the current proposed properties for different types of variables. Koichi found them reasonable but of course he will be evaluating it further and discussing the memory model with other MRI developers.

The next steps are:

- Improve the memory model document to make it more understandable (e.g. by adding examples)
- Continue in the discussions around properties of the variables to determine if the current proposal is fine or if it needs changes

#25 - 02/08/2018 09:41 AM - googlefeud (google feud (spammer, locked))

- Subject changed from Documenting Ruby memory model to Book my show

#26 - 02/08/2018 09:59 AM - usa (Usaku NAKAMURA)

- Subject changed from Book my show to Documenting Ruby memory model

#27 - 12/23/2021 11:40 PM - hsbt (Hiroshi SHIBATA)

- Project changed from 14 to Ruby