Ruby - Feature #12607

Ruby needs an atomic integer

07/21/2016 05:04 AM - shyouhei (Shyouhei Urabe)

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

Description

(This one was derived from bug #12463)

Although I don't think += would become atomic, at the same time I understand Rodrigo's needs of *easier* counter variable that resists inter-thread tampering. I don't think ruby's Integer class can be used for that purpose for reasons (mainly because it is not designed with threads in mind). Rather we should introduce a integer class which is carefully designed.

Why not import Concurrent::AtomicFixnum into core?

Related issues:

Related to Ruby - Feature #12463: ruby lacks plus-plus	**Rejected**
Has duplicate Ruby - Feature #14706: Atomic Integer incr/decr	**Closed**

History

#1 - 07/21/2016 05:04 AM - shyouhei (Shyouhei Urabe)

- Related to Feature #12463: ruby lacks plus-plus added

#2 - 07/21/2016 05:29 AM - sawa (Tsuyoshi Sawada)

Do we want to have another integer variant just after having Fixnum and Bignum been excluded in favor of the Integer class?

#3 - 07/21/2016 12:26 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I guess the proper way of helping users to write concurrent code in Ruby is to provide concurrent classes in stdlib rather than relying on third-party gems.

I don't think it makes sense to write something like 1.+=(value) since 1 is immutable. It makes more sense to provide some Counter class to stdlib which would be thread-safe. And a ConcurrentHash and ConcurrentArray. Those alone would already help a lot.

#4 - 08/03/2016 01:58 PM - sawa (Tsuyoshi Sawada)

Sorry, I do not clearly understand why the original Integer class cannot be made atomic. If it is not possible or if there would be any problem, can someone explain why that is?

Ideally, I think it would be better if the original Integer class can be modified in some way rather than additional classes being introduced.

#5 - 08/03/2016 06:57 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

This is not really about Integer handling +=. there's no "+=" method. It's a short-hand syntax: a += 1 is currently expanded to something like "a = a + 1". One proposal is to change it to be expanded to something like "synchronized(a = a + 1)".

Other proposals could offer other concurrent classes so that one would write something like "counter = Counter.new(initial_value)" (0 by default) and "counter.inc(step)" (1 by default). But it doesn't make sense to me to call "count = Integer.new(initial_value)" since Integer instances are constants.

#6 - 08/10/2016 04:14 AM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Assigned

- Assignee set to ko1 (Koichi Sasada)

I heard from Koichi that he has an opinion on this. Please respond.

#7 - 09/04/2016 04:35 PM - jwmittag (Jörg W Mittag)

Tsuyoshi Sawada wrote:

> Do we want to have another integer variant just after having Fixnum and Bignum been excluded in favor of the Integer class?

Well, it's not really another integer variant. It's really not a number at all. It's a concurrency primitive (that happens to be a number).

#8 - 11/05/2016 10:00 AM - ko1 (Koichi Sasada)

At first, we need a box of integer, not an atomic integer.

Like that:

```
box = IntegerBox.new(0)
box.increment
box.increment
box.to_i #=> 2
```

This IntegerBox can be implemented by the following code:

```
class IntegerBox
  def initialize n
    @n = n
    @m = Mutex.new
  end
  def increment i = 1
    @m.synchonization{ @n += i }
  end
  def to_i
    @n
  end
end
```

I'm not sure such small part should be include in Ruby's core library.

BTW, concurrent-ruby supports AtomicFixnum http://ruby-concurrency.github.io/concurrent-ruby/Concurrent/AtomicFixnum.html.

```
af = Concurrent::AtomicFixnum.new(0)
af.increment
af.increment
af.value #=> 2
```

It has update method to set new value. It seems useful.

```
v = af.value
new_v = v + 5
af.update{ new_v }
```

But this small code doesn't work as intended.
This is why I think thread-programming is not easy, even if there is cool threading tools.

#9 - 11/05/2016 10:01 AM - ko1 (Koichi Sasada)

- Status changed from Assigned to Feedback

#10 - 11/05/2016 12:11 PM - shyouhei (Shyouhei Urabe)

So you mean we don't need atomic integer because concurrent-ruby sucks? I don't think that's a valid reason to reject this.

#11 - 11/05/2016 01:27 PM - ko1 (Koichi Sasada)

> So you mean we don't need atomic integer because concurrent-ruby sucks? I don't think that's a valid reason to reject this.

I didn't mean that. After "BTW" is only my impression about threading.

#12 - 11/05/2016 01:55 PM - shyouhei (Shyouhei Urabe)

OK, then let's discuss the needs of this class.

An obvious benefit of this class to be in core is that we don't even need Mutex in theory. Because we have GVL, if we do this in core we can just increment the backend integer and that should suffice. No overheads must be there. Even when we give up GVL someday in a future, we could still write this using LOCK CMPXCHG or something equivalent. Or JRuby people might want to implement this class using java.util.concurrent.atomic.AtomicLong. Either way, the resulting implementation must be much smarter than the mutex-synchronized pure-ruby implementation.

#13 - 11/05/2016 02:14 PM - ko1 (Koichi Sasada)

I agree with performance advantages.

I'm negative to introduce this feature because it is too hard.

People who know well about thread programming can use this feature using concurrent-ruby and I don't want to recommend it for people who don't know about threading well. Also I think such convenient tools can lead misuse. I think using Mutex explicitly is more easy to understand the behavior of application. So I don't want to encourage these by introducing ruby std libs.

#14 - 04/24/2018 03:06 AM - shyouhei (Shyouhei Urabe)

- Has duplicate Feature #14706: Atomic Integer incr/decr added

#15 - 04/24/2018 12:22 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Should this issue be assigned to Matz?

#16 - 04/26/2018 09:04 PM - mperham (Mike Perham)

I don't like pulling in the entire concurrent-ruby gem only for a few lines of code. I've implemented my own slow but thread-safe Counter here:

https://github.com/mperham/sidekiq/blob/b58c505df3501d8c1de5207552d68dd2d9abea31/lib/sidekiq/processor.rb#L189

I would love to see an optimized atomic implementation in stdlib!

#17 - 04/27/2018 02:11 AM - shyouhei (Shyouhei Urabe)

We have already shown the benefits of atomic integers very well. What Ko1 says is the downside of it ("it is too hard for mere mortals"). Now is the time for us to tell him it's not that bad. Showing another benefits of concurrency does not help.

#18 - 04/27/2018 07:14 AM - Student (Nathan Zook)

I think that you might not have understood his concern. Getting multi-threaded code right is hard, no matter the primitives available. "People who know well about thread programming can use this feature using concurrent-ruby and I don't want to recommend it for people who don't know about threading well." The issue is about setting traps for the unwary.

I think that it is interesting that the original discussing was around doing an atomic "++" -- because "++" and even "+" are not atomic in C.

I spent a few years doing assembler, mostly in PowerPC. PowerPC (as of 10 years ago) did not have any instruction comparable to the X86 compare and exchange. It does make sense, strictly from a performance standpoint, to have a mutex capability that can be dropped down to a few integer instructions. But doing that requires that you understand exactly how those instructions work in the major architectures. Given that the Ruby Integer class can spill any cache line, it is nonsensical to talk about fast locks using it. You would want a register-sized value in some sort of wrapping class.

In my mind, it is a matter of boosting performance at the risk of creating traps for unwary programmers. So far as I'm concerned, ruby's metaprogramming capacity has already filled the wind in those sails. Therefore, I would be in favor of a robust, fast atomic capacity. But not with the Integer class. You really need to go with a 32-bit integer to maintain compatibility.

If you need more than 32 bits in your concurrency primitives, you had BETTER be brewing your own extensions. ;)

#19 - 04/27/2018 06:07 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Just to make it clearer, since I was mentioned by Shyouhey in the issue's description, performance has never been the reason why I asked for support for atomic operations such as incrementing an integer. It might seem like it is the case because the mentioned ticket (#12463) is about performance but my comment there was just incidental as it mentioned a new syntax change to Ruby and I felt it would be interested if the new syntax wasn't used for performance issues only, but that could be a chance to improve a common operation which is an atomic incrementing operation. Not a performance improvement, but an improvement in the sense that writing such operations would be greatly simplified by not requiring us to write an explicit mutex block for those operations.

I think we shouldn't put too much focus on the performance subject while discussing such an operator introduction. Mike Perham, on the other hand, didn't suggest a new syntax as far I as could understand his request. While "i++" is certainly shorter than "count = Thread::Count.new; count.inc" the latter doesn't involve any changes to the language syntax and it's certainly much shorter than using a mutex or having to create a counter class ourselves for every project where we need an atomic increment operation.

So, I guess we all agree that it's possible to improve atomic increments, but Koichi is concerned about people doing wrong multi-thread programming just because Ruby would introduce some Thread-safe classes or something like that.

I don't really understand that being a reason to not implement something a lot of people would find useful. For example, one should always use the block version when opening a file so that it automatically closes it after running the block whenever it's possible. However Ruby allows people to open and close the file in separate actions and that too could lead to bugs, but that didn't prevent Ruby from providing an open variation without a block (requiring an explicit close).

Finally, I think this issue is mostly about changing Ruby's syntax while Mike's is about adding a new Thread::Counter core class. Those are two different and interesting proposals but they should be probably discussed independently as they don't really conflict with each other, even if Koichi's concerns apply to both of them.

I just don't think Ruby can protect developers from themselves so it shouldn't even try that. It doesn't work IMHO but it prevents those who are familiar

with thread programming to do it with less cerimony in Ruby.

Anyway, I just suggested that maybe Matz could give us his opinion on the subject. That's because Shyouhei assigned this ticket to Koichi and set the state to "Feedback" and Koichi has already provided his feedback. So, if it's decided already, this issue's state should be changed to Rejected, but if it's still open to discussion and since Koichi has already specified his opinion, maybe we should assign it to Matz now that Koichi has already provided feedback on this.

But I'd probably leave the performance reason off the table when discussing both tickets. They don't seem as much relevant as the easiness of writing multi-threaded application, but maybe that's just my opinion.

#20 - 04/27/2018 06:08 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

- Tracker changed from Bug to Feature

- Backport deleted (2.1: UNKNOWN, 2.2: UNKNOWN, 2.3: UNKNOWN)

I'm not sure if type was set to Bug on purpose, so I'm changing it to Feature instead, but feel free to change it back to Bug if you really think it's a bug.

#21 - 04/27/2018 06:19 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

Oh, sorry, I just noticed this issue isn't about changing Ruby's syntax, it was just my comment on the other related issue that mentioned that :P So this is basically the same request as Mike's.

#22 - 01/06/2021 12:06 PM - Eregon (Benoit Daloze)

FWIW, I think an atomic integer makes sense in core, because it's easier to optimize it that way.
But, since it seems a lot nicer to support fixnums and bignums (not just fixnums), and notably the Integer class is now used for both, I think the better thing to expose is an AtomicReference.
As an added benefit, that has other usages and notably adds a general compare-and-swap operation.

TruffleRuby for instance has such a class, which is currently used by concurrent-ruby:
https://github.com/oracle/truffleruby/blob/07d62320c0d83efd6f7f99a805d6f61d1e03d60d/doc/user/truffleruby-additions.md#atomic-references
https://github.com/oracle/truffleruby/blob/07d62320c0d83efd6f7f99a805d6f61d1e03d60d/src/main/ruby/truffleruby/core/truffle/truffleruby.rb#L23-L51

#23 - 01/06/2021 03:17 PM - dsisnero (Dominic Sisneros)

I agree this should be added to core because of ractor and #17433.

#24 - 01/12/2021 07:54 AM - ko1 (Koichi Sasada)

I believe introducing STM https://bugs.ruby-lang.org/issues/17261 is more general for this purpose.

#25 - 01/12/2021 11:13 AM - Eregon (Benoit Daloze)

ko1 (Koichi Sasada) wrote in #note-24:

> I believe introducing STM https://bugs.ruby-lang.org/issues/17261 is more general for this purpose.

True, but an atomic integer remains useful even if there is an STM.
An STM is overkill and much slower if all someone wants is atomic counters.

#26 - 01/13/2021 07:32 AM - ko1 (Koichi Sasada)

Eregon (Benoit Daloze) wrote in #note-25:

> True, but an atomic integer remains useful even if there is an STM.
> An STM is overkill and much slower if all someone wants is atomic counters.

TVar proposed in https://bugs.ruby-lang.org/issues/17261 has #increment method and it is enough fast.

#27 - 01/13/2021 06:05 PM - Dan0042 (Daniel DeLorme)

What is the use case for this atomic integer? I sometimes have a use for an atomic monotonically increasing counter, but that could be satisfied with a very simple API like n = MyCounter.next
On the other hand I've never felt the need for decrement or reset or in fact any integer atomic operations other than += 1
2¢

#28 - 01/13/2021 09:45 PM - chrisseaton (Chris Seaton)

> What is the use case for this atomic integer?

For example for issuing unique IDs across multiple Ractors.

> or in fact any integer atomic operations other than += 1

For example a CAS to update a bank balance is a common requirement.

#29 - 01/13/2021 10:40 PM - marcandre (Marc-Andre Lafortune)

chrisseaton (Chris Seaton) wrote in #note-28:

> > What is the use case for this atomic integer?
>
> For example for issuing unique IDs across multiple Ractors.

You can achieve something similar with Object.new.object_id

> > or in fact any integer atomic operations other than += 1
>
> For example a CAS to update a bank balance is a common requirement.

Why atomic though... This can all be achieved with a Ractor.

#30 - 01/13/2021 11:45 PM - chrisseaton (Chris Seaton)

> You can achieve something similar with Object.new.object_id

You may want them to be consecutive.

> Why atomic though... This can all be achieved with a Ractor.

A Ractor send has relatively high synchronisation overhead - an atomic integer is conventionally implemented at the cache-coherence level and is lock-free.

I can only offer that I use atomic integers for many things fairly frequently in my working life.

#31 - 01/14/2021 02:25 AM - Dan0042 (Daniel DeLorme)

chrisseaton (Chris Seaton) wrote in #note-28:

> For example for issuing unique IDs across multiple Ractors.

That's the "monotonically increasing counter" case I was talking about, and for that I really think a generator (like the above MyCounter.next) is a more appropriate tool than an atomic integer.

> For example a CAS to update a bank balance is a common requirement.

Can you elaborate? In my experience financial transactions will be handled at the DB level. Not to mention you usually need more intricate synchronization such as atomic update of the balances of *two* accounts. I don't quite see how an atomic integer is a realistic tool for money stuff.

> I can only offer that I use atomic integers for many things fairly frequently in my working life.

That's great, so you should easily be able to give a concrete example *other* than generating unique sequential IDs. I'm quite curious, so looking forward to a nice realistic use case.

#32 - 01/14/2021 02:34 AM - chrisseaton (Chris Seaton)

> In my experience financial transactions will be handled at the DB level.

So imagine you're implementing the DB then.

> Can you elaborate?

If you want to perform an arbitrary operation on a counter you can do that atomically, without a lock, by doing a CAS between the value as it currently is and the value you want to set it to, even if it takes you a long time to compute that. Atomics let you do this in a lock-free way.

> so you should easily be able to give a concrete example other than generating unique sequential IDs

An use-case I was working on earlier today was profiling operations like coverage and sampling tools. These need to be reset as well if they're in danger of over-flowing a range that the machine can natively implement with low-overhead.

#33 - 01/14/2021 03:41 AM - Dan0042 (Daniel DeLorme)

chrisseaton (Chris Seaton) wrote in #note-32:

> An use-case I was working on earlier today was profiling operations like coverage and sampling tools.

Ah yes, beautiful example! Accumulating metrics require both atomicity and performance.

#34 - 01/14/2021 01:06 PM - Eregon (Benoit Daloze)

There is also the well-known example of metrics in Sidekiq, and all these:
https://github.com/search?l=Ruby&q=AtomicFixnum&type=Code

> I really think a generator (like the above MyCounter.next) is a more appropriate tool than an atomic integer.

And how do you implement that efficiently, and in a way that's thread-safe?
To be clear, Enumerator.new {} is not efficient, so a generator is not good enough, and Fibers can't be resumed across threads.

ko1 (Koichi Sasada) wrote in #note-26:

> TVar proposed in https://bugs.ruby-lang.org/issues/17261 has #increment method and it is enough fast.

I think that cannot be as efficient as an atomic integer.
Reasoning: TVar#increment needs to also be atomic with other changes, including Thread.atomically { tv.value = tv.value * 2 }, for decent STM semantics.
That implies extra tracking for TVar#increment besides just a single fetch-and-add, isn't it?
For instance, it would be incorrect to execute that atomically block in parallel with the fetch-and-add (might result in the increment being lost). So TVar#increment needs to sync somehow with Thread.atomically and that's the overhead.

#35 - 01/14/2021 02:43 PM - Dan0042 (Daniel DeLorme)

Eregon (Benoit Daloze) wrote in #note-34:

> And how do you implement that efficiently, and in a way that's thread-safe?
> To be clear, Enumerator.new {} is not efficient, so a generator is not good enough, and Fibers can't be resumed across threads.

By "a generator" I did not mean the Enumerator::Generator class. I meant a new generator-type class (say, AtomicSequence) that is implemented with atomic CAS operations. No one here is disputing the usefulness of atomic integers *in general*, just their relevance as a concurrency primitive in *ruby core*.

A naive implementation of a monotonic sequence using an atomic integer class might look like this:

```
SEQUENCE.increment
guid = SEQUENCE.to_i
```

oops, wrong! When running concurrently this will produce duplicate guids. The correct usage would be guid = SEQUENCE.increment but you have to depend on the developer to understand the gotcha with SEQUENCE.to_i. In comparison, guid = SEQUENCE.next is always correct; you don't even need to be *aware* of concurrency issues.

What about a naive implementation of metrics/statistics using an atomic integer class...

```
t0 = Time.now
yield
t = ((Time.now - t0) * 1000000).round #microsecond
TIME.increment(t)
NB.increment(1)
avg = TIME.to_i / NB.to_i
```

oops, wrong again! When running concurrently the avg might be incorrect.

Maybe a more useful abstraction might be an AtomicVector? If such a thing is possible...

```
TIME_NB = AtomicVector.new(0, 0)
TIME_NB.increment(t, 1)
avg = TIME_NB.to_a.inject(:/)
```

Basically all I'm trying to say is there's a clear benefit to choosing abstractions that produce the correct result *even when used naively*. Like Ractor. So I understand why ko1 is reluctant about this.

#36 - 01/29/2021 08:59 AM - ko1 (Koichi Sasada)

> ko1 (Koichi Sasada) wrote in #note-26:
>
> > TVar proposed in https://bugs.ruby-lang.org/issues/17261 has #increment method and it is enough fast.
>
> I think that cannot be as efficient as an atomic integer.

Yes. It is slower than single purpose atomic integers with hardware instruction with JIT.
But (I didn't measured yet) increment method call on VM needs method invocation overhead and it is relatively higher than TVar#increment overhead.
Again, I didn't measure and it can be wrong.

#37 - 01/29/2021 12:28 PM - Eregon (Benoit Daloze)

ko1 (Koichi Sasada) wrote in #note-36:

> But (I didn't measured yet) increment method call on VM needs method invocation overhead and it is relatively higher than TVar#increment overhead. Again, I didn't measure and it can be wrong.

Why would the invocation of AtomicInteger#increment be more expensive than the invocation of TVar#increment?
They are both method calls, and TVar doesn't doesn't use Primitive since it is a separate gem (currently at least).

In general, method calls have no cost when inlined with a JIT that can see through the core library, e.g. on TruffleRuby, or I think with MJIT and leaf annotated methods, so I think for best performance AtomicInteger#increment is a clear winner.

#38 - 07/25/2022 01:14 PM - Eregon (Benoit Daloze)

It'd be good to revisit this (e.g. see https://twitter.com/_byroot/status/1550580128723476480).

Ractor has little need for this. But with Thread there is a clear need for this.
Also the STM work was not merged, so that's not a replacement either.
I think @ko1's concern of "too hard to use" should be no blocker: Rubyists frequently use Threads, almost all Ruby webservers and Rails use threads nowadays, there is no point to deny that.
And it is completely unrealistic to think Ractor will ever replace Threads, Ractor can only support a small subset of Ruby gems, and it will never be all or as powerful as threads.
For instance I don't see Rails being able to use Ractor for parallel requests anytime soon.

Regarding that example it's easy to fix, and such conditions are well understood by anyone knowing about compare-and-swap or this kind of concurrency tools.
It can also be documented on the update method.

```
v = af.value
af.update{ |v| v + 5 }
```

So, let's add AtomicInteger or AtomicReference (since it's more general) because it's convenient and needed by existing gems?
Having it in concurrent-ruby works but it's obviously less practical, and less optimized than it could be.

Also there is a trap when using Mutex as a compare-and-swap way with try_lock, because that leaves the Mutex locked and when the thread dies it unlocks all Mutex it had, which can be pretty slow (I found that out when reimplementing the timeout gem).

#39 - 07/25/2022 02:22 PM - chrisseaton (Chris Seaton)

Yes, I think a few low-level concurrency primitives should move from Concurrent Ruby to core. As we add more concurrency to Ruby, and add JIT compilers, ideally the VM should understand these primitives, rather than being opaque extension code.