Ruby trunk - 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 trunk - Feature #12463: ruby lacks plus-plus	**Rejected**
Has duplicate Ruby trunk - Feature #14706: Atomic Integer incr/decr	**Open**

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)

- Assignee set to ko1 (Koichi Sasada)

- Status changed from Open to Assigned

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)

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

- Tracker changed from Bug to Feature

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.