Ruby master - Feature #17261
Software transactional memory (STM) for Threads and Ractors
10/12/2020 08:24 AM - ko1 (Koichi Sasada)

Status: Rejected
Priority: Normal
Assignee:
Target version:

Description

Abstract
I propose Software transactional memory (STM) for threads and ractors.

Implementation is here: https://github.com/ruby/ruby/pull/3652

The interface is similar to concurrent-ruby, but not the same.
http://ruby-concurrency.github.io/concurrent-ruby/1.1.4/Concurrent/TVar.html

Basic concept
https://en.wikipedia.org/wiki/Software_transactional_memory
Transaction is popular idea on data base systems to keep state consistency.

STM is similar idea to implement optimistic synchronization strategy.

There are several advantages compare with traditional synchronization techniques like Mutex and so on:

- Performance: in some cases, it is faster because of optimistic nature.
- Composability: multiple locks can introduce dead-lock. STM allows nested transaction. In other words, (some kind of) STM can guarantee the progressiveness.

The disadvantages is, it can lead slow down on high-contention cases.

API

- `Thread::atomically do expr end`: make a new transaction and run expr in it. expr can be retried if the conflict is detected.
- `Thread::TVar.new(default_value)`
- `Thread::TVar#value`: get current value of TVar
- `Thread::TVar#value = val`: set TVar value val.
- `Thread::TVar#increment(n=1)`: Just same as `Thread.atomically{ tv.value += 1 }`.

Note that expr for `Thread.atomically` can retries and all `TVar#value=` (set TVar values) are reverted before retries. Another operations such as other memory modification, IO operations includes network operations etc are not reverted.

The very difference between `Concurrent::TVar` is:

- `TVar only refer to shareable objects to support Ractor.``
- `TVar#value=` should be used with `atomically`. We can define as `Thread.atomically{ tv.value = val }`, but it can lead misusing without `atomically`.
- `TVar#increment` is special case to allow setting without `atomically` to support typical single counter cases.

Implementation
https://github.com/ruby/ruby/pull/3652

The implementation is almost same as TL2, lock-based STM with global version clock with pthread/win32 threads.
We can use atomic operations but not supported yet (but only a few performance benefit on my measurements).

Example
N = 1,000,000

tv1 = Thread::TVar.new(0)
tv2 = Thread::TVar.new(0)

r1 = Ractor.new tv1, tv2 do |tv1, tv2|
  loop do
    Thread.atомically do
      v1, v2 = tv1.value, tv2.value
      raise if v1 != v2
    end
  end
end

rs = 3.times.map do
  Ractor.new tv1, tv2 do |tv1, tv2|
    N.times do
      Thread.atомически do
        tv1.value += 1
        tv2.value += 1
      end
    end
  end
end

rs.each{|r| r.take}
p [tv1.value, tv2.value] #=> [3000000, 3000000]

In this case,

- all atomically blocks keep consistency that tv1.value == tv2.value.
- the results [3000000, 3000000] shows consistency on +=1.

Here is famous bank-account example:

class Account
  COUNT = Thread::TVar.new 0

  def initialize deposit = 0
    @i = COUNT.increment
    @balance = Thread::TVar.new(deposit)
  end

  def transfer_from acc, n
    Thread::atomically do
      acc.withdraw n
      self.deposit n
    end
  end

  def transfer_to acc, n
    Thread::atomically do
      self.withdraw n
      acc.deposit n
    end
  end

  def withdraw n
    @balance.value -= n
  end

  def deposit n
    @balance.value += n
  end

  def balance
    @balance.value
  end
end
AN = 1_0000
N = 10_000_000
RN = 10
iter = 0
accs = AN.times.map(Account.new.freeze).freeze

require 'benchmark'

# :forward
# two ractors operate N times: a[i].transfer(a[i+1])
# R1: a1->a2, a2->a3, ...
# R2: a1->a2, a2->a3, ...

# :reverse
# two ractors operate N times: a[i].transfer(a[i+1]),
# but the order of accounts are reversed.
# R1: a1->a2, a2->a3, ...
# R2: a1->aN-1, a2->aN-2, ...

# :shuffle
# RN ractors operate N times: a[rand].transfer(a[rand])
# It simulates normal bank-operation

mode = :shuffle

loop do
  iter += 1

  btime = Time.now

  case mode
  when :forward
    rs = []
    rs << Ractor.new(accs).do |accs|
      N.times{|i|
        a1, a2 = accs[i%accs.size], accs[(i+1)%accs.size]
        a1.transfer_to(a2, 1)
      }
    end

    rs << Ractor.new(accs).do |accs|
      N.times{|i|
        a1, a2 = accs[i%accs.size], accs[(i+1)%accs.size]
        a1.transfer_from(a2, 1)
      }
    end

    rs.each{|r| r.take}
  when :reverse
    rs = reverse
    rs << Ractor.new(accs).do |accs|
      N.times{|i|
        a1, a2 = accs[i%accs.size], accs[(i+1)%accs.size]
        a1.transfer_to(a2, 1)
      }
    end

    rs << Ractor.new(accs.reverse.freeze).do |accs|
      N.times{|i|
        a1, a2 = accs[i%accs.size], accs[(i+1)%accs.size]
        a1.transfer_from(a2, 1)
      }
    end

  end

end

03/14/2022
This program creates AN bank accounts and repeats N transfer operations. You can observe that huge AN reduces conflicts and the execution time is low. Small AN reduces conflicts -> many retries and the execution time is high.

<table>
<thead>
<tr>
<th>AN</th>
<th>Execution time (s)</th>
<th>Retry counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>6.914</td>
<td>958,969</td>
</tr>
<tr>
<td>1,000</td>
<td>3.107</td>
<td>186,267</td>
</tr>
<tr>
<td>10,000</td>
<td>2.549</td>
<td>26,183</td>
</tr>
<tr>
<td>100,000</td>
<td>2.627</td>
<td>2,458</td>
</tr>
</tbody>
</table>

Now x10 retries doesn't affect execution time x10, this is because the current Ractor implementation (acquiring a global lock to raise an exception, and it reduces the retry counts). If we improve the Ractor's implementation, the result would be more worse.

### Consideration

**Thread::atomically in ractors**

At first, I implemented this feature with Ractor::atomically and Ractor::TVar. However, this STM feature will help the thread programming. This is why I moved from Ractor::atomically to Thread::atomically.

Introduce Concurrent namespace what concurrent-ruby are using. However, there are small differences so that I'm not sure is is feasible.

Another idea is to support alias: Thread::atomically and Ractor::atomically.

**Thread::TVar can refer only shareable objects**

Threads can access all objects so we don't need to restrict by such rule. However, to support ractors, this restriction is needed.

One idea is separate Thread::TVar and Ractor::TVar, but it can introduce confusion.
Only with shareable objects, thread programs become more thread-safe, so I think it is good choice to have current restriction.

**Bug detection**

Similar to locking, we can forget to use a atomically like that:

```ruby
class C
  def initialize
    @tv1 = Thread::TVar.new(0)
    @tv2 = Thread::TVar.new(0)
  end
  def tv1() = @tv1.value
  def tv2() = @tv2.value
  def tv1 = (v)
    Thread.atomically{ @tv1.value = v }
  end
  def tv2 = (v)
    Thread.atomically{ @tv2.value = v }
  end
end

obj = C.new
obj.tv1 += 1
obj.tv2 += 2
```

It works but it can introduce inconsistency if tv1 and tv2 are tightly coupled with because tv1 and tv2 are not accessed in the same transaction.

If tv1 and tv2 need to be modified consistently, we need to write like the following:

```ruby
Thread.atomically do
  obj.tv1 += 1
  obj.tv2 += 1
end
```

and tv1/tv2/tv1=tv2= methods should not be defined.

I mean we can write bad programs easily.

It is same situation with traditional locking (we need to use Mutex appropriately). The duty to use it correctly is for programmer.

There are some advantages compared with traditional locking:

- We can concentrate on TVars. On traditional thread programming we need to check all memory state.
- We can introduce logging mechanism and we can find wrong usage (for example: tv1 and tv2 are set within independent transactions). I think we can make some checker based on the log. On traditional thread programming, there are several similar works, but it is difficult to check it because the target of state is most of memory operations.

**Related works**

- There are many STM implementation techniques.
  https://www.morganclaypool.com/doi/abs/10.2200/S00070ED1V01Y200611CAC002
- Concurrent Haskell and Clojure are famous to support STM in language (I think).
  - The model of STM is similar to Clojure.
  - Clojure allows to access TVar (ref in Clojure) value without atomically (dosync in Clojure).
  - Clojure doesn't allow to set TVar value without atomically.
  - The API is similar to Concurrent Haskell (TVar and atomically).
- Concurrent-ruby has Concurrent::TVar.
  - But it allows to have an unshareable object.
  - But is allows to set the value with atomically.

**History**

#1 - 10/12/2020 08:26 AM - ko1 (Koichi Sasada)
- Description updated

#2 - 10/12/2020 09:29 AM - ioquatix (Samuel Williams)
What does TVar mean?

#3 - 10/12/2020 12:04 PM - hsbt (Hiroshi SHIBATA)
See http://ruby-concurrency.github.io/concurrent-ruby/1.1.4/Concurrent/TVar.html

A TVar is a transactional variable

#4 - 10/13/2020 06:19 PM - Eregon (Benoit Daloze)
Interesting :)

Having it on Thread sounds nice to me.

I wonder how limiting it is to restrict to only shareable values for TVar.
Is there actually any type usable for TVar except Integer?
A frozen String could only be swapped with .value=, which seems rather uninteresting.
For a frozen Array of shareable elements, one would need to create a new frozen Array to modify any element, which seems rather expensive.
One could use Array.new(n) { TVar.new(0) }.freeze I guess, but that would then need to create a new array to change the size of the Array (for <<, pop, shift, unshift,...).
That's probably still enough to build a STM hashtable with an Array of TVar of Entry and the Entry are frozen and have key: shareable, value: TVar.

Not sure about other thread-safe collection using the STM to synchronize, maybe the shareable-only is too limiting for some structures like trees (or causes too much overhead).

#5 - 10/26/2020 04:34 PM - ko1 (Koichi Sasada)
At last dev-meeting, this proposal is not accepted to introduce it in core because the importance of this feature is not sure.

Maybe I'll introduce as ractor gem to study the API and usefulness of this feature.

#6 - 10/28/2020 05:29 PM - chrisseaton (Chris Seaton)
I wrote a long-form blog post to give people interested in this proposed feature some context.
https://chrisseaton.com/truffleruby/ruby-stm/

I've also suggested a benchmark that we can start to use for experimenting.

#7 - 10/28/2020 05:52 PM - Eregon (Benoit Daloze)
I think it would be nice to make it its own gem (tvar maybe?) given it's under Thread and not under Ractor, and it could most likely work on other Ruby implementations (e.g., TruffleRuby).
That should also make it possible to e.g. improve performance independently of Ruby releases.

#8 - 10/29/2020 04:03 PM - ko1 (Koichi Sasada)

I think it would be nice to make it its own gem (tvar maybe?) given it's under Thread and not under Ractor, and it could most likely work on other Ruby implementations (e.g., TruffleRuby).

For this purpose, concurrent-ruby provides TVar for threads.

#9 - 10/29/2020 04:03 PM - ko1 (Koichi Sasada)
- Status changed from Open to Rejected

#10 - 10/29/2020 05:21 PM - Eregon (Benoit Daloze)
ko1 (Koichi Sasada) wrote in #note-8:

For this purpose, concurrent-ruby provides TVar for threads.

That does not work with Ractor though (at least currently).

#11 - 10/29/2020 06:19 PM - ko1 (Koichi Sasada)
Eregon (Benoit Daloze) wrote in #note-10:

That does not work with Ractor though (at least currently).
You wrote "under Thread and not under Ractor". What does it mean?

#12 - 10/29/2020 07:22 PM - Eregon (Benoit Daloze)
ko1 (Koichi Sasada) wrote in #note-11:

You wrote "under Thread and not under Ractor". What does it mean?

I was talking about the namespace, the proposal is Thread::TVar (and not Ractor::TVar), which I agree is a good namespace for it.

In any case, I think it would be valuable to have this work in a gem if it's not in core directly.

#13 - 10/29/2020 07:40 PM - ko1 (Koichi Sasada)
Eregon (Benoit Daloze) wrote in #note-12:

I was talking about the namespace

Ah, I see.

#14 - 10/29/2020 08:54 PM - chrisseaton (Chris Seaton)
I think there's benefits to building STM into the language (if we decided we want STM at all) rather than it being a library.

When you look at more advanced features like conflict resolution and conflict mitigation you may want to do more lower level things like control scheduling.

#15 - 10/30/2020 12:17 AM - ko1 (Koichi Sasada)
chrisseaton (Chris Seaton) wrote in #note-14:

I think there's benefits to building STM into the language (if we decided we want STM at all) rather than it being a library.

I think so, especially with Ractors, but now we (I) don't have enough evidence to persuade, so we can ask later with use cases.

When you look at more advanced features like conflict resolution and conflict mitigation you may want to do more lower level things like control scheduling.

Yeah it will be fun work.