For some reasons, we need to provide a way to make sharable Proc between ractors.

- (1) A block for the Ractor.new.
- (2) Send a proc between ractors.
- (3) A block for global callback methods: define_method ([Bug #17159]), TracePoint, ...

For (1), we use Proc#isolate (isolate is temporary name here) which prohibit to access outer variables.

```ruby
a = 1
Proc.new{
  p a
}.isolate # => can not isolate a Proc because it accesses outer variables (a).
# error on `isolate` method call
```

There are no states to share, so it is okay.

For (2), Proc#isolate is one option because we can send parameters with an argument call. But it should be a bit long.

```ruby
i, j, k = nil
pr = Proc.new do |i, j, k|
  p i, j, k
end.isolate

r = Ractor.new do |task, param|
  task.call(*param)
end

r.send([pr, [i, j, k]])
```

For (3), maybe we need to make more flexible Proc which can read outer block parameter on that snapshot (discussed in #17159).

Now, I named it with freeze, because it seems frozen Proc.

```ruby
a = 1
# try to read, and returns old value (snapshot at `freeze`)  
pr = Proc.new{
  p a #=> 1
}
pr = pr.freeze
pr.call
a = 2
pr.call #=> 1

# try to write, and it is not allowed
pr2 = Proc.new{
  a = 1
}
pr2 = pr.freeze
```
can not freeze a Proc because it accesses outer variables (a). (ArgumentError)

To share the "frozen" Proc between reactors, outer values should be (deep) frozen. It means readable values (in above case, a) should be shareable.

Now we named it Proc#shareable!

```ruby
a = [1, [2, 3]]
pr = Proc.new{
  p a.frozen? #=> true
}.shareable!

a[0] = 0 #=> frozen error
```

This ticket has three different variant of mutability and shareability for Proc.

<table>
<thead>
<tr>
<th>outer lvar</th>
<th>shareable</th>
<th>freeze/making shareable other objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. isolate</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>b. freeze</td>
<td>allow to read</td>
<td>No</td>
</tr>
<tr>
<td>c. shareable!</td>
<td>allow to read</td>
<td>Yes</td>
</tr>
</tbody>
</table>

I want to introduce functionality of shareable!, but not sure the Ruby-level API.

I think (b) freeze for this semantics is good name because it only allows to read-only local variables. However, it is not enough to make a sharable Proc because read objects from the Proc should be also sharable.

Making freeze with (c) shareable! functionality is one idea, but I think freeze should not deep-freezing because it is very surprising that read objects become the sharable (== frozen) for usual Ruby users. Maybe Ractor.make_shareable(pr) makes pr sharable is no surprise because it is good declaration the pr should be shareable, even if the read objects from pr become sharable (== frozen).

Removing (a) isolate and using (c) shareable! at Ractor.new(&b) is one idea, but I think it is surprising that they can access outer local variables, but the they can not access newly assigned variables as usual blocks.

```ruby
a = 1
Ractor.new do
  p a # only 1
end

a = 2
```

(a) isolate does not have such issue because all outer lvars accesses are not allowed == easy to understand, easy to debug.

In practice, accessing outer variables with multi-ractor program is very useful because we need to declare same local variables if we want to access them from different ractors.

The following example is from [Feature #17261]:

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

r1 = Ractor.new tv1, tv2 do |tv1, tv2| # <-- here
  loop do
    Thread.atomiclly do
      v1, v2 = tv1.value, tv2.value
      raise if v1 != v2
    end
  end
end
```

With (c) shareable! semantics, it is easier to write:

```ruby
tv1 = Thread::TVar.new(0)
tv2 = Thread::TVar.new(0)
```
r1 = Ractor.new do
  loop do
    Thread.atomic do
      v1, v2 = tv1.value, tv2.value
      raise if v1 != v2
    end
  end
end

Above example is also enable to make more simple:

i, j, k = nil
pr = Proc.new do
  p i, j, k
end
r = Ractor.new do |task|
  task.call
end
r.send(pr)

However, using this semantics (shareable!) can freeze extra-variables in accidents:

a = [1, 2, 3]

Ractor.new do
  do_something if a.length > 0
end
a << 4 # raises FrozenError

It is clear that there is a syntax or method to apply shareable! functionality.

a = [1, 2, 3]
Ractor.new & (Ractor.make_shareable(Proc.new { a.length ... }))

It can be used with define_method which can invoke from ractors:

define_method(name, Ractor.make_shareable(Proc.new { ... }))

But it is too long.

There are implementations for (a), (b) and (c), but the API is not fixed, so there is no PR now.

I'm thinking to introduce (c)'s feature in Ractor.make_shareable(pr).
To use with define_method, maybe it should be more friendly. Ideally, new syntax is great.

There is no conclusion, and your comments are welcome.

Thanks,
Koichi

Associated revisions

Revision 5d97bdc2 - 10/29/2020 06:12 PM - ko1 (Koichi Sasada)
Ractor.make_shareable(a_proc)

Ractor.make_shareable() supports Proc object if
(1) a Proc only read outer local variables (no assignments)
(2) read outer local variables are shareable.

Read local variables are stored in a snapshot, so after making
shareable Proc, any assignments are not affect like that:

a = 1
pr = Ractor.make_shareable(Proc.new{p a})
pr.call #=> 1
a = 2
pr.call #=> 1 # `a = 2` doesn't affect

[Feature #17284]

**History**

---

#1 - 10/25/2020 05:06 PM - ko1 (Koichi Sasada)
- Description updated

#2 - 10/25/2020 07:33 PM - ko1 (Koichi Sasada)
https://github.com/ruby/ruby/pull/3700
for Ractor.make_shareable(a_proc) (a_proc becomes shareable Proc with (c) shareable! semantics).

#3 - 10/25/2020 08:04 PM - marcandre (Marc-Andre Lafortune)
I think (c) semantics are definitely the most useful.

For API: Ractor.make_shareable(proc) (and equivalently proc.deep_freeze)

I'm not sure of the use-cases for a) or b).

#4 - 10/26/2020 01:34 AM - ko1 (Koichi Sasada)
Thank you for your reply.

I'm not sure of the use-cases for a) or b).

Do you think Ractor.new() can call (c) instead of current (a)?
In other words, can we accept (1) and (2) in the following example?

```ruby
b = a = []
Ractor.new do
  p a #=> (1) shows [] even if a is replaced with :sym
end
a = :sym
# (2) frozen error because an array is sharable (deep frozen)
b << 1 # frozen error
```

#5 - 10/26/2020 05:25 AM - marcandre (Marc-Andre Lafortune)
ko1 (Koichi Sasada) wrote in [#note-4]:

Thank you for your reply.

Here's a longer reply.

I would like a method to make a Proc independent of the binding it was created in. I'm thinking of Proc#detach that would make a snapshot of the values needed (shallow copy):

```ruby
x = 1
a = ary = []
pr = Proc.new { ary << x; x += 1 }.detach
x = ary = nil # no effect on `p`, as binding is detached
pr.call #=> 2
pr.call #=> 2, same since always starts from snapshot
a #=> [1, 1], since snapshot is shallow copy
binding.local_variables #=> [:x, :a, :ary, :pr]
pr.binding.local_variables #=> [] # always empty; value are passed like arguments
pr.binding.snapshot #=> { ary: [], x: 1 } # not necessary, but at least for illustration
pr.binding.snapshot.frozen #=> true
```

This is always what I want when I call define_method and I have to jump through hoops to make sure I don't capture another value by mistake...

```ruby
def foo(text)
  text.each_line do |line|
    if special_line?(line)
      foo, bar = parse_line(line)
  end
```

---

11/04/2020
define_method(foo) {
  puts bar
}
end
end
end
# oops, 'text' might *never* be garbage collected, and last `line` will not be either :-(

I see Ractor.make_shareable(block) as equivalent to detach + make_shareable on the values of the snapshot.

This would make it easy to check if a block accesses non-shareable outer variables:

```ruby
n = 42
ary = []
Ractor.shareable?(Proc.new { do_something() }) #=> false, has binding
Ractor.shareable?(Proc.new { do_something(v).detach }) #=> true, snapshot shareable
Ractor.shareable?(Proc.new { do_something(ary).detach }) #=> false, because 'ary' not shareable
ary.freeze
Ractor.shareable?(Proc.new { do_something(ary).detach }) #=> true because 'ary' is shareable
ary2 = []
p = Ractor.make_shareable(Proc.new { do_something(ary2) })
ary2.frozen #=> true
Ractor.shareable?(p) #=> true
```

Do you think Ractor.new() can call (c) instead of current (a)?
In other words, can we accept (1) and (2) in the following example?

```ruby
b = a = []
Ractor.new do
  p a #=> (1) shows [] even if a is replaced with :sym
end
a = :sym
# (2) frozen error because an array is sharable (deep frozen)
```

b << 1 #=> frozen error

I think it could definitely call detach above, so 1) yes.

2) is trickier/riskier. I think there are better solutions.

Maybe a better way to resolve 2 is simply that:

```ruby
b = a = []
Ractor.new do
  p a
end
# equivalent to:
b = a = []
Ractor.new(a) do |a|
  p a
end
a = :sym # no effect
b << 1 # no effect, array was deeply copied
```

#17286 would allow for:

```ruby
b = a = []
Ractor.new(move: true) do
  p a
end
a = :sym # no effect
b << 1 # Ractor::MovedError
```

I hope I'm not missing something obvious, it's getting late here :-)

#6 - 10/26/2020 08:07 AM - Eregon (Benoit Daloze)

(c) sounds the most useful and general.

While reading the description, I thought Proc#deep_freeze is a good name.
That clearly says it will freeze transitively the closure (stopping at shareable objects).

I think it would be good if such Proc copying the closure have an explicit .deep_freeze call, or some other syntax, including for Ractor.new.
That way it's clear they behave differently than usual, and that they snapshot the closure.
Maybe we can use some new syntax inside the block's parameters?

```ruby
a = []
b = []
Ractor.new(a) do |a, ^deep_freeze|
  p [a,b]
end
a << 1 # OK, does not affect the Ractor
b << 2 # FrozenError
b = Object.new # seems worth an error (or warning), which we can do if we use new syntax on the block
They is probably better syntax for this, but this illustrates the idea.
```

I think C++ lambda copy/move specifiers are too complicated. Deep-copy only works if the Proc is called once and passed to a single Ractor, so I would only have a specifier to deep freeze all variables, and Ractor.new can still take arguments to explicitly deep copy them.

#7 - 10/26/2020 01:26 PM - Dan0042 (Daniel DeLorme)

(a) The method name `isolate` sounds like it will convert the proc to make it isolated, but it seems all it does is raise an error if the proc is not already isolated from the outer scope?

(b) If we can read an outer lvar but it is not frozen/made shareable, I guess that can only mean it is deep-copied?

```ruby
a = b = [1,2]
Ractor.new do
  a << 3
  p a # [1,2,3]
  p b # [1,2] or [1,2,3]?
end.take
b << 4
p a # [1,2,4]
p b # [1,2,4]
```

marcandre (Marc-Andre Lafortune) wrote in #note-3:

> I think c) semantics are definitely the most useful. For API: Ractor.make_shareable(proc) (and equivalently proc.deep_freeze)

+1

#8 - 10/26/2020 03:55 PM - ko1 (Koichi Sasada)

Today's meeting, there are comments:

- (c) is too danger to freeze reachable objects from reachable local variables.

```ruby
def foo(&b) b.shareable!; end
a = [1]
foo { p a }
a << 2 #=> frozen error
```

This example is more worse because the block writer can not know the application of .shareable!.

- There is another idea Proc.shareable[ a ] makes shareable Proc if readable variables (a in this case) refers shareable objects.

```ruby
a = [1, 2]
Proc.shareable[ a ] # raise an error
```

```ruby
b = [1, 2].freeze
Proc.shareable[ a ] # ok.
```

It is mild.

With this idea, we can write a ractor aware `define_method` like:

```ruby
n.times{|i| define_method("m#{i}", Proc.shareable{ p i })}
```

or if we choose the definition of Ractor.make_shareable(a_proc):

```ruby
n.times{|i| define_method("m#{i}", Ractor.make_shareable{proc{ p i }})}
```

#9 - 10/26/2020 04:12 PM - ko1 (Koichi Sasada)
I have several question about Proc#detach to understand your idea.

```ruby
x = 1
a = ary = []
pr = Proc.new { ary << x; x += 1}.detach
x = ary = nil # no effect on `p’, as binding is detached # ko1: what is `p’? `pr’?
pr.call # => 2
pr.call
# => 2, same since always starts from snapshot # ko1: does `x’ is initialized with 1 at every Proc#call?
a
# => [1, 1], since snapshot is shallow copy # ko1: I'm not sure why `a’ is affected because of shallow copy?
binding.local_variables # => [:x, :a, :ary, :pr]
pr.binding.local_variables # => [] # always empty; value are passed like arguments
pr.binding.snapshot # => { ary: [], x: 1 } # not necessary, but at least for illustration
pr.binding.snapshot.frozen # => true
```

This is always what I want when I call define_method and I have to jump through hoops to make sure I don't capture another value by mistake...

I'm not sure why you can avoid mis-capturing with Proc#detach... Ah, I got it. Only objects from variables in Proc are marked.

#10 - 10/26/2020 04:15 PM - ko1 (Koichi Sasada)

(b) If we can read an outer lvar but it is not frozen/made shareable, I guess that can only mean it is deep-copied?

in dev-meeting, there is same comment: they should copy everything and freeze them instead of marked as frozen and sharable. However this approach does not work on some objects, IO for example.

#11 - 10/26/2020 05:44 PM - marcandre (Marc-Andre Lafortune)

ko1 (Koichi Sasada) wrote in #note-8:

Today's meeting, there are comments:

- (c) is too danger to freeze reachable objects from reachable local variables.

```ruby
def foo(&b) b.shareable!; end

a = [1]
foo{ p a }
a << 2 #=> frozen error
```

This example is more worse because the block writer can not know the application of .shareable!.

This is already possible, (even if the block does not refer to a):

```ruby
def foo(&b) b.binding.local_variable_get(:a).freeze; end

a = [1]
foo{}
a << 2 #=> frozen error
```

Yet it does not happen in real life, because people know what they are doing.

There are many ways to shoot yourself in the foot in Ruby, that is usually not a problem.

- There is another idea Proc.shareable{ a } makes shareable Proc if readable variables (a in this case) refers shareable objects.

```ruby
a = [1, 2]
Proc.shareable{ a } # raise an error
b = [1, 2].freeze
Proc.shareable{ a } # ok.
```

It is mild.

If I'm not mistaken, this is:

```ruby
def Proc.shareable(&b)
```
I have several question about Proc#detach to understand your idea. what is p? pr?

Yes, sorry
does x is initialized with 1 at every Proc#call?

Yes
I'm not sure why a is affected because of shallow copy?
a and ary, and ary in the snapshot all refer to the same Array.

Only objects from variables in Proc are marked.

Yes

marcandre (Marc-Andre Lafortune) wrote in #note-11:

This is already possible, (even if the block does not refer to a):

```ruby
def foo(&b) b.binding.local_variable_get(:a).freeze; end

a = [1]
foo{}
a << 2 #=> frozen error
```

Yet it does not happen in real life, because people know what they are doing.

There are many ways to shoot yourself in the foot in Ruby, that is usually not a problem.

It is true. But the degree is not same. It is easy to freeze them accidentally. So Matz and other attendees showed concern about it.

- There is another idea Proc.shareable{ a } makes shareable Proc if readable variables (a in this case) refers shareable objects.

```ruby
a = [1, 2]
Proc.shareable{ a } # raise an error
```

```ruby
b = [1, 2].freeze
Proc.shareable{ a } # ok.
```

It is mild.

If I'm not mistaken, this is:

```ruby
def Proc.shareable(&b)
  b = b.detach
  raise unless Ractor.shareable?(b)
  b
end
```

Ractor.shareable?(b) for Proc is not defined well, but maybe yes.

I have several question about Proc#detach to understand your idea. what is p? pr?

Yes, sorry
does x is initialized with 1 at every Proc#call?
Yes

But you can set it to another value (x = 2 for example).

My understanding:

Proc#detach do
  1. allocate a snapshot area
  2. copy object (which can be referred from Proc's variables) references to snapshot
Proc#call do
  1. allocate outer-lvars area for outer variables
  2. copy snapshot refs to outer-lvars area

Proc#freeze in this ticket is similar to Proc#detach, but does not do special at Proc#call.

I think outer-variables are used to store the information cross Proc#call so I doubt Proc#detach is useful on many cases.

  I'm not sure why a is affected because of shallow copy?

  a and ary, and ary in the snapshot all refer to the same Array.

Now I understand.

#13 - 10/27/2020 02:49 AM - marcandre (Marc-Andre Lafortune)

ko1 (Koichi Sasada) wrote in #note-12:

  My understanding:
  
  Proc#detach do
    1. allocate a snapshot area
    2. copy object (which can be referred from Proc's variables) references to snapshot
  
  Proc#call do
    1. allocate outer-lvars area for outer variables
    2. copy snapshot refs to outer-lvars area

  Proc#freeze in this ticket is similar to Proc#detach, but does not do special at Proc#call.

Indeed. But freeze must do special check for reassignments.

What I dislike about Proc#freeze is that it does not make intuitive sense to me. A Proc is not mutable per say. Calling Proc.new { ... } does not change the Proc.

Also, if after Proc#freeze you can reassign a outside the block and has no effect inside the block, then they are different local variables. It is not intuitive for me to disallow reassigning one and not the other.

I will agree that in general, these variables will not be reassigned anyways so it won't matter much. I just think it is easier to understand if you are allowed to reassigning it. Do you think there would be a noticeable difference in performance either way?

#14 - 10/27/2020 04:25 AM - ko1 (Koichi Sasada)

marcandre (Marc-Andre Lafortune) wrote in #note-13:

  ko1 (Koichi Sasada) wrote in #note-12:

    My understanding:
    
    Proc#detach do
      1. allocate a snapshot area
      2. copy object (which can be referred from Proc's variables) references to snapshot
    
    Proc#call do
1. allocate outer-lvars area for outer variables
2. copy snapshot refs to outer-lvars area

Proc#freeze in this ticket is similar to Proc#detach, but does not do special at Proc#call.

Indeed. But freeze must do special check for reassignments.

Is "reassignments" $a = 1$?
Yes. It is checked at #freeze timing (if there is assignments to outer variables, raise an error).

What I dislike about Proc#freeze is that it does not make intuitive sense to me. A Proc is not mutable per say. Calling Proc.new { ... } does not change the Proc.
Also, if after Proc#freeze you can reassign a outside the block and has no effect inside the block, then they are different local variables. It is not intuitive for me to disallow reassigning one and not the other.

There is two positions, environments (lvar space) is belong to a Proc, or a Proc only refers to environments.
Proc#freeze terminology uses the position "environments (lvar space) is belong to a Proc".
And in fact, environments are different object in implementation. So your intuition is also correct.

I use Proc#freeze terminology to explain the design to discuss this ticket, and I don't care to change (or remove) this name.

I will agree that in general, these variables will not be reassigned anyways so it won't matter much.

Do you have any useful example of outer-variable reassignment on Proc#detach semantics?

I just think it is easier to understand if you are allowed to reassigning it.

You say it is easy to explain the feature, right? It can be.
But I also think if I can set outer-variables, I expect they are shared with other Procs (bindings).

So freezing semantics is easy/no-misunderstanding feature I think.

Do you think there would be a noticeable difference in performance either way?

mmm. copying overhead and memory overhead? not so big difference I think.
Actually, current Proc#freeze implementation is slow and memory consuming because it copies all readable variables.

#15 - 10/29/2020 03:47 PM - ko1 (Koichi Sasada)
By discussion with Matz and several MRI committers, we decided that making shareable Proc should be more conservative.

- Ractor.make_shareable(read_values) should be danger to freeze objects unexpectedly.
- if all read variables are shareable, it is easy to make a Proc shareable.

Here is modified implementation: https://github.com/ruby/ruby/pull/3722

With this implementation, we can invoke a cross ractor Proc.

```ruby
class C
  a = 1
  define_method "foo", Ractor.make_shareable(Proc.new{ a })
a = 2
end
Ractor.new( C.new.foo ).take #=> 1
```

If we found more reasonable specification, we can relax this specification (maybe making strict is more difficult).

Also I want to provide a way to switch this specification to test the usability (by ractor gem?).

#16 - 10/29/2020 06:12 PM - ko1 (Koichi Sasada)
- Status changed from Open to Closed

Applied in changeset git|5d97bdc2dcb835c877010daa033cc2b1dfeb86d6.

Ractor.make_shareable(a_proc)
Ractor.make_shareable() supports Proc object if
(1) a Proc only read outer local variables (no assignments)
(2) read outer local variables are shareable.

Read local variables are stored in a snapshot, so after making
shareable Proc, any assignments are not affect like that:

```ruby
a = 1
pr = Ractor.make_shareable(Proc.new{p a})
pr.call #=> 1
a = 2
pr.call #=> 1 # `a = 2` doesn't affect
```

[Feature #17284]

#17 - 10/29/2020 07:49 PM - Eregon (Benoit Daloze)

Ractor.make_shareable does traverse reachable and not-already-shareable objects for other objects than Procs, it seems bad that it behaves
differently for Proc.

I think the intention is clear with Ractor.make_shareable(Proc.new(p a)), i.e., I think it's OK to Ractor.make_shareable(a) (no matter what type is a),
and it should be expected given Ractor.make_shareable is transitive.