Ruby trunk - Feature #905

Add String.new(fixnum) to preallocate large buffer

12/19/2008 08:54 AM - headius (Charles Nutter)

Status:	Closed	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:		

Description

=begin
Because Strings are used in ruby as arbitrary byte buffers, and because the cost of growing a String increases as it gets larger (especially when it starts small), String.new should support a form that takes a fixnum and ensures the backing store will have at least that much room. This is analogous to Array.new(fixnum) which does the same thing.

The simple implementation of this would just add a Fixnum check to the String.new method, and the result would be an empty string with that size buffer. This would allow heavy string-appending algorithms and libraries (like ERb) to avoid doing so many memory copies while they run.
=end

Related issues:

Related to Ruby trunk - Feature #12024: Add String.buffer, for creating strin...	**Closed**

History

#1 - 12/19/2008 09:16 AM - brixen (Brian Shirai)

In Rubinius, we have found it useful to have String.pattern(size, value) where value can be a fixnum or a string. The string created will be size characters where value is repeated. For example:

```
String.pattern(5, ?a)     # => "aaaaa"
String.pattern(5, " ")    # => "     "
String.pattern(5, "110") # => "11011"
```

Something like "ab" * 5 then becomes String.pattern("ab".size * 5, "ab") behind the scenes in our implementation.

#2 - 12/19/2008 09:31 AM - matz (Yukihiro Matsumoto)

Hi

In message "Re: [ruby-core:20668] [Feature #905] Add String.new(fixnum) to preallocate large buffer"
on Fri, 19 Dec 2008 08:46:13 +0900, Charles Nutter redmine@ruby-lang.org writes:

> Because Strings are used in ruby as arbitrary byte buffers, and because the cost of growing a String increases as it gets larger (especially when it starts small), String.new should support a form that takes a fixnum and ensures the backing store will have at least that much room. This is analogous to Array.new(fixnum) which does the same thing.

I like the idea.

But I'd prefer adding a new class method for the purpose,
say. String#buffer(n), to adding new role to an argument by type,
or there may be a better name.

```
                              matz.
```

#3 - 12/19/2008 09:40 AM - pragdave (Dave Thomas)

On Dec 18, 2008, at 6:23 PM, Yukihiro Matsumoto wrote:

> But I'd prefer adding a new class method for the purpose,
> say. String#buffer(n), to adding new role to an argument by type,
> or there may be a better name.

Maybe

```
str = String.capacity(n)
```

or

```
str = String.sized(n)
```

(presumably n is in bytes, not characters)

#4 - 12/19/2008 10:13 AM - headius (Charles Nutter)

Yukihiro Matsumoto wrote:

> I like the idea.
>
> But I'd prefer adding a new class method for the purpose,
> say. String#buffer(n), to adding new role to an argument by type,
> or there may be a better name.

I thought String.new(1000) would be a nice equivalent to
Array.new(1000), since they both do essentially the same thing. But I'm
not opposed to a separate method. I would vote for something active and
descriptive like "allocate", but that's obviously not available.
"buffer" isn't bad.

I like "new" best. And of course Array.new changes behavior depending on
argument types too.

```
Array.new(size=0, obj=nil)
Array.new(array)
Array.new(size) {|index| block }
```

What's good for the goose...

#5 - 12/19/2008 10:35 AM - headius (Charles Nutter)

Brian Ford wrote:

> Issue #905 has been updated by Brian Ford.
>
> In Rubinius, we have found it useful to have String.pattern(size, value) where value can be a fixnum or a string. The string created will be size characters where value is repeated. For example:
>
> ```
> String.pattern(5, ?a)     # => "aaaaa"
> String.pattern(5, " ")    # => "     "
> String.pattern(5, "110") # => "11011"
> ```
>
> Something like "ab" * 5 then becomes String.pattern("ab".size * 5, "ab").

Yeah, seems like a reasonably good idea. The string formats seem a
little weird to me though...not what I'd expect. The analog with an
array would be to accept another array as the fill value, which seems
equally weird.

#6 - 12/19/2008 12:13 PM - headius (Charles Nutter)

Dave Thomas wrote:

> (presumably n is in bytes, not characters)

Yes, I'm thinking bytes myself. Presumably you either want just a byte
buffer or you know how many bytes you need to allocate for the character
encoding you intend.

#7 - 12/19/2008 05:11 PM - duerst (Martin Dürst)

At 09:08 08/12/19, Brian Ford wrote:

> Something like "ab" * 5 then becomes String.pattern("ab".size * 5, "ab").

I'm at a total loss to see why

```
String.pattern("ab".size * 5, "ab")
```

should be in any way better than

`"ab" * 5`

but maybe that's just me. Can somebody explain?

Regards,   Martin.

#-#-# Martin J. Du"rst, Assoc. Professor, Aoyama Gakuin University
#-#-# http://www.sw.it.aoyama.ac.jp   mailto:duerst@it.aoyama.ac.jp

#8 - 12/19/2008 05:34 PM - rklemme (Robert Klemme)

2008/12/19 Martin Duerst duerst@it.aoyama.ac.jp:

> At 09:08 08/12/19, Brian Ford wrote:
>
> > Something like "ab" * 5 then becomes String.pattern("ab".size * 5, "ab").
>
> I'm at a total loss to see why
>
> `String.pattern("ab".size * 5, "ab")`
>
> should be in any way better than
>
> `  "ab" * 5`
>
> but maybe that's just me. Can somebody explain?

It's probably not. But if you want "ab" repeated 5 times, then "ab" *
5 is probably the ideal solution.

Fixing the length in bytes is a new and different feature, i.e. your
pattern is cut off. With String.pattern("bo",3) you would get "bob"
or maybe "bo" but in a buffer of length 3 bytes.

We could also name it "resize" with this semantics:

String.resize(100, "foo") -> "foo" and buffer has 100 bytes
String.resize(100) -> "" and buffer has 100 bytes

Or maybe just "size". IMHO the idea was that you do not need to have
a String beforehand so all solution which are instance methods receive
a String as argument are probably suboptimal because they will
typically be invoked like show above, i.e. with a String constructor
which needs one object allocation (including GC bookkeeping overhead)
and a bit of memory.

A typical use case would be this:

```
File.open "foo" do |io|
  buffer = String.size(1024)
  while io.read(1024, buffer)
    # or even io.read(buffer.capacity, buffer)
    $defout.write(buffer)
  end
end
```

Kind regards

robert

---

remember.guy do |as, often| as.you_can - without end

#9 - 12/20/2008 12:41 AM - headius (Charles Nutter)

Jim Weirich wrote:

> There is a slight difference. Array.new(1000) creates an array with a
> thousand elements. The proposed String.new(1000) would create a string
> with zero characters, but the ability to grow to 1000 characters (umm,
> bytes) without internal reallocation.
>
> I think this difference is enough to warrant different names.

Yes, that's a good point; and the resulting array would << to the 1001st element.

I suppose then coming up with a common name that works for both would be a good idea. I'm back to liking "buffer" in both cases.

String.buffer(1000) produces an empty string that can grow to 1000 bytes without needing to resize/copy.

Array.buffer(1000) produces an empty array that can grow to 1000 elements without needing to resize/copy.

Whatever is decided, I think it's going to be something people want (need) on 1.8, so I'll probably submit a backport request as well (and perhaps write up a simple extension people can use until then).

#10 - 12/20/2008 12:58 AM - pragdave (Dave Thomas)

On Dec 19, 2008, at 9:33 AM, Charles Oliver Nutter wrote:

> I suppose then coming up with a common name that works for both would be a good idea. I'm back to liking "buffer" in both cases.
>
> String.buffer(1000) produces an empty string that can grow to 1000 bytes without needing to resize/copy.
>
> Array.buffer(1000) produces an empty array that can grow to 1000 elements without needing to resize/copy.

I think the reason I dislike this is that you're creating methods that are polymorphic on the types of their arguments, and yet we generally don't do that in Ruby-level code. So by creating these methods, you're giving them a different flavor from methods that would be written in straight Ruby.

How about something more Ruby-like:

```
s = String.new(initial_capacity: 1000)
```

```
t = String.new(buffer, initial_capacity: 2*buffer.length)
```

Dave

#11 - 12/20/2008 03:56 AM - headius (Charles Nutter)

Dave Thomas wrote:

> I think the reason I dislike this is that you're creating methods that are polymorphic on the types of their arguments, and yet we generally don't do that in Ruby-level code. So by creating these methods, you're giving them a different flavor from methods that would be written in straight Ruby.

Neither of those methods are polymorphic on anything. They're both new methods that accept a Fixnum.

#12 - 12/20/2008 06:38 AM - pragdave (Dave Thomas)

On Dec 19, 2008, at 12:48 PM, Charles Oliver Nutter wrote:

> Neither of those methods are polymorphic on anything. They're both new methods that accept a Fixnum.

.new is

String.buffer is not a meaningful name for a constructor, in my opinion, whereas String.new has a pedigree. By adding a initial_size: n optional argument, you exactly express the meaning—you're asking for an initial allocation when String.new executes. Similarly,

```
Array.new([1,2,3], initial_size: 100)
```

lets you both initialize and allocation a new array.

Right now, we have File.open("fred", "w"), rather than
File.open_write("fred"). It seems like a good idea, particularly for
an interface that's likely to grow over time (I can forsee

```
String.new(initial_size: 1000, fill_with: " ", encoding: binary,
etc: ...)
```

Cheers

Dave

#13 - 12/20/2008 07:36 AM - headius (Charles Nutter)

=begin
Dave Thomas wrote:

> On Dec 19, 2008, at 12:48 PM, Charles Oliver Nutter wrote:
>
> > Neither of those methods are polymorphic on anything. They're both new
> > methods that accept a Fixnum.
>
> .new is

And already has multiple forms in Array, so there's precedent. Also,
adding multiple forms with different named arguments doesn't reduce the
complexity of that single method any.

> String.buffer is not a meaningful name for a constructor, in my opinion,
> whereas String.new has a pedigree. By adding a initial_size: n optional
> argument, you exactly express the meaning—you're asking for an initial
> allocation when String.new executes. Similarly,
>
> Array.new([1,2,3], initial_size: 100)

But Array.new(initial_size: 100).size would == 0. That's confusing...I
think buffer better expresses that it's the backing store being sized
than the outward expression of the String or Array itself, which is what
initial_size means.

I would also expect that the cost of allocating and populating an
arguments hash for this would negate some of the gain from adding the
new form. Array.buffer(100) adds almost no overhead on top of the
physical creation of the backing store and object to wrap it, where
Array.buffer(initial_size: 100) creates both a new Array and a new Hash.
An implementation detail, sure, but we I think we just need something
simple here. Perhaps buffer just doesn't express it clearly enough?

> Right now, we have File.open("fred", "w"), rather than
> File.open_write("fred"). It seems like a good idea, particularly for an
> interface that's likely to grow over time (I can forsee
>
> String.new(initial_size: 1000, fill_with: " ", encoding: binary, etc:
> ...)

It seems to me this is making the semantics of String.new much more
complicated, rather than simpler and more uniform. And at least encoding
is already available outside of "new", so this is little more than a
shortcut. But there's absolutely no way at present to allocate a string
with a guaranteed backing store size, and that's the sole intention of
this RFE.

=end

#14 - 12/20/2008 08:21 AM - pragdave (Dave Thomas)

=begin

On Dec 19, 2008, at 4:28 PM, Charles Oliver Nutter wrote:

> > String.buffer is not a meaningful name for a constructor, in my
> >
> > opinion, whereas String.new has a pedigree. By adding a
> >
> > initial_size: n optional argument, you exactly express the meaning—
> > you're asking for an initial allocation when String.new executes.
> >
> > Similarly,
> > Array.new([1,2,3], initial_size: 100)
>
> But Array.new(initial_size: 100).size would == 0. That's
>
> confusing...I think buffer better expresses that it's the backing
>
> store being sized than the outward expression of the String or Array
>
> itself, which is what initial_size means.
>
> I would also expect that the cost of allocating and populating an
>
> arguments hash for this would negate some of the gain from adding
>
> the new form. Array.buffer(100) adds almost no overhead on top of
>
> the physical creation of the backing store and object to wrap it,
>
> where Array.buffer(initial_size: 100) creates both a new Array and a
>
> new Hash. An implementation detail, sure, but we I think we just
>
> need something simple here. Perhaps buffer just doesn't express it
>
> clearly enough?

I don't think the cost of a hash is going to be significant--if it is,

then I'd hope that implementors find a way of optimizing these styles

of keyword hashes, because they're used more and more

(*cough*Rails*cough*).

I agree initial_size: is misleading. Perhaps String.new(preallocate: n)

Dave

=end

#15 - 12/20/2008 12:17 PM - headius (Charles Nutter)

=begin
Dave Thomas wrote:

> I don't think the cost of a hash is going to be significant--if it is,
> then I'd hope that implementors find a way of optimizing these styles of
> keyword hashes, because they're used more and more (*cough*Rails*cough*).

Hard to do, since it has to be a hash on the callee side. Constructing
the hash could perhaps be delayed, in case the callee was a C function,
but it still has to be something. In comparison, new(1000) is almost free.

=end

#16 - 12/20/2008 02:53 PM - rogerdpack (Roger Pack)

=begin

> Hard to do, since it has to be a hash on the callee side. Constructing the
> hash could perhaps be delayed, in case the callee was a C function, but it
> still has to be something. In comparison, new(1000) is almost free.

I suppose a clever implementation could optimize that out.
-=R

=end

#17 - 12/20/2008 03:26 PM - brixen (Brian Shirai)

=begin
On Dec 19, 12:02 am, Martin Duerst due...@it.aoyama.ac.jp wrote:

> At 09:08 08/12/19, Brian Ford wrote:
>
>> Something like "ab" * 5 then becomes String.pattern("ab".size * 5, "ab").
>
> I'm at a total loss to see why
>   String.pattern("ab".size * 5, "ab")
> should be in any way better than
>   "ab" * 5
> but maybe that's just me. Can somebody explain?

It's not better. It's not intended that you see it. Behind the scenes
it has been useful for us to have this method. This is one example of
its usage. The string is allocated in one step and filled in one loop.

There are other places it has been useful. The most useful aspect is
requesting a particular size. The initial contents is a lesser, but
still useful aspect.

The mileage for other implementations may vary.

Cheers,
Brian

> Regards,   Martin.
>
> #-#-# Martin J. Du"rst, Assoc. Professor, Aoyama Gakuin University
> #-#-# http://www.sw.it.aoyama.ac.jp   mailto:due...@it.aoyama.ac.jp

=end

#18 - 12/20/2008 04:35 PM - headius (Charles Nutter)

=begin
Gary Wright wrote:

> How about:
>
> String.reserve(100)
> Array.reserve(100)

"reserve" is pretty good. I'll abstain from commenting on any other
forms since I really just want the single-param fixnum version myself.

=end

#19 - 12/23/2008 01:46 AM - headius (Charles Nutter)

=begin
I guess the relative silence on this issue means there's not much more
to discuss. Here's a summary up to now:

- Everyone seems to agree it's a good idea to add, so we should add it.
  And I would like to see it backported to 1.8.6/7.

- Everyone likes the flat fixnum form except Dave Thomas, who would like
  it to be a keyword argument. But that would not support backporting and
  no core methods currently accept keyword arguments, plus it would create
  a throw-away hash in all current implementations.

- Several names have been suggested: overload 'new', buffer,
  preallocate, capacity, sized, reserve. I prefer 'buffer' and 'reserve',

with a strong lean toward 'buffer' because it mimics a well-known idiom
in the Java world: "String.buffer(1000)" == "new StringBuffer(1000)".

- Other forms have been suggested that accept a fill fixnum or fill
string; however I believe we should skip these cases for now since we're
not actually creating a string of a certain size (and content), we're
creating an empty string with a backing store of a certain size. The
expectation is that the contents of that backing store are unimportant
(perhaps \000s), and so fill params are meaningless.

So for me, the solution is String.buffer(1000). I rest my case, your honor.
=end

#20 - 12/23/2008 01:46 AM - headius (Charles Nutter)

=begin
I guess the relative silence on this issue means there's not much more
to discuss. Here's a summary up to now:

- Everyone seems to agree it's a good idea to add, so we should add it.
And I would like to see it backported to 1.8.6/7.

- Everyone likes the flat fixnum form except Dave Thomas, who would like
it to be a keyword argument. But that would not support backporting and
no core methods currently accept keyword arguments, plus it would create
a throw-away hash in all current implementations.

- Several names have been suggested: overload 'new', buffer,
preallocate, capacity, sized, reserve. I prefer 'buffer' and 'reserve',
with a strong lean toward 'buffer' because it mimics a well-known idiom
in the Java world: "String.buffer(1000)" == "new StringBuffer(1000)".

- Other forms have been suggested that accept a fill fixnum or fill
string; however I believe we should skip these cases for now since we're
not actually creating a string of a certain size (and content), we're
creating an empty string with a backing store of a certain size. The
expectation is that the contents of that backing store are unimportant
(perhaps \000s), and so fill params are meaningless.

So for me, the solution is String.buffer(1000). I rest my case, your honor.

=end

#21 - 12/23/2008 03:01 AM - pragdave (Dave Thomas)

=begin

On Dec 22, 2008, at 10:37 AM, Charles Oliver Nutter wrote:

> no core methods currently accept keyword arguments, plus it would
>
> create a throw-away hash in all current implementations.

File.open...

=end

#22 - 02/03/2009 10:44 AM - shyouhei (Shyouhei Urabe)

- Assignee set to matz (Yukihiro Matsumoto)

=begin

=end

#23 - 03/04/2010 01:26 AM - mame (Yusuke Endoh)

- Status changed from Open to Feedback

=begin
Hi,

> This would allow heavy string-appending algorithms and libraries (like ERb) to avoid doing so many memory copies while they run.

Is it really a bottleneck? Please make an experiment and show us
the result.

We can continue API discussion after we confirm the feature really
makes sense.

--
Yusuke Endoh mame@tsg.ne.jp
=end

#24 - 03/04/2010 03:11 PM - coatl (caleb clausen)

=begin
Do we really need a benchmark to confirm that copying large strings is expensive? Pre-sized buffers are a well-known performance win on other systems, so why not for ruby as well?

I would like to try to create a benchmark to prove this would help, but it may be some time before I can get to it.
=end

#25 - 03/04/2010 06:03 PM - now (Nikolai Weibull)

=begin
On Thu, Mar 4, 2010 at 07:11, caleb clausen redmine@ruby-lang.org wrote:

> Issue #905 has been updated by caleb clausen.
>
> Do we really need a benchmark to confirm that copying large strings is expensive? Pre-sized buffers are a well-known performance win on other systems, so why not for ruby as well?

Doesn't this unnecessarily expose implementation details about String?
Preallocation doesn't make as much sense if Strings were implemented
using, for example, Ropes [1].

[1] http://en.wikipedia.org/wiki/Rope_(computer_science)

=end

#26 - 03/04/2010 09:26 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/3/4 caleb clausen redmine@ruby-lang.org:

> Pre-sized buffers are a well-known performance win on other systems, so why not for ruby as well?

Indeed, it will bring speed up to Ruby, but if the speed up is
negligibly-small, it is not only actually useless but also bad
for code maintenance.

If we confirm the performance up is significant and a patch is
present, we'll be strongly encouraged to discuss the feature
actively and to import the patch.

I have forgotten another matter. I also wonder how many case
we can expect a precise length that ERB will generate.
If we cannot in many cases, the feature may be still useless.

Well, it may be good only if the feature can be used in Rails...

--
Yusuke ENDOH mame@tsg.ne.jp

=end

#27 - 03/05/2010 01:39 AM - murphy (Kornelius Kalnbach)

=begin
Doesn't Ruby allocate already using a "double memory if you run out"
rule? That makes string concatenation (amortized) linear, even if the

string must be moved in the memory.

I doubt that there are real-world use cases that would be much faster
with preallocation. As Yusuke said, ERb is more of a counter-example.

Even with this API extension, we wouldn't have control over the
generation of the string buffer in many use cases, as in Array#join,
String#% or in literals using #{}. Its use would be limited to String#<<.

[murphy]

=end

#28 - 03/05/2010 02:13 AM - hgs (Hugh Sasse)

=begin
On Fri, 5 Mar 2010, Kornelius Kalnbach wrote:

> Doesn't Ruby allocate already using a "double memory if you run out"
> rule? That makes string concatenation (amortized) linear, even if the
> string must be moved in the memory.

Yes (last time I looked), but while this sort of thing is
being looked at I'd like to remind people of the cunning code inside
Lua for handling large string concatenations:

http://www.lua.org/pil/11.6.html

It seems relevant in terms of moving data about.

```
    HTH
    Hugh
```

=end

#29 - 03/05/2010 02:29 AM - mame (Yusuke Endoh)

=begin
Hi,

2010/3/5 Kornelius Kalnbach murphy@rubychan.de:

> Doesn't Ruby allocate already using a "double memory if you run out"
> rule? That makes string concatenation (amortized) linear, even if the
> string must be moved in the memory.

Yes, it does. This is why I think experiment is needed.

Because the suggested feature can be used to omit first some
expansions, it will actually reduce time. But I guess if the
reduced time is not so much.

> Even with this API extension, we wouldn't have control over the
> generation of the string buffer in many use cases, as in Array#join,
> String#% or in literals using #{}. Its use would be limited to String#<<.

Absolutely. The feature is hard to use.
Even if we pre-allocated a string, calling some method on the
string may shrink it.

I think we should call the feature just "optimization hint"
rather than API. It is better to think the hint may be even
ignored.

--
Yusuke ENDOH mame@tsg.ne.jp

=end

#30 - 03/05/2010 02:39 AM - mame (Yusuke Endoh)

=begin

Hi,

2010/3/5 Hugh Sasse hgs@dmu.ac.uk:

> Yes (last time I looked), but while this sort of thing is
> being looked at I'd like to remind people of the cunning code inside
> Lua for handling large string concatenations:
>
> http://www.lua.org/pil/11.6.html

At first glance, the document explains the difference of destructive
and non-destructive concatenations, like String#+ and #<<.

It is absolutely different topic from pre-allocation.

--
Yusuke ENDOH mame@tsg.ne.jp

=end

#31 - 03/05/2010 02:51 AM - hgs (Hugh Sasse)

=begin

On Fri, 5 Mar 2010, Yusuke ENDOH wrote:

> Hi,
>
> 2010/3/5 Hugh Sasse hgs@dmu.ac.uk:
>
>> Yes (last time I looked), but while this sort of thing is
>> being looked at I'd like to remind people of the cunning code inside
>> Lua for handling large string concatenations:
>>
>> http://www.lua.org/pil/11.6.html
>
> At first glance, the document explains the difference of destructive
> and non-destructive concatenations, like String#+ and #<<.
>
> It is absolutely different topic from pre-allocation.

It is related: the algorithm constructs large strings from smaller
ones in an elegant way using a "tower of Hanoi", and if the top
string concatenation gets bigger than the one below it, only then
are they joined together. Result is less copying and merging.
Admittedly, it is less applicable with mutable strings, but while
only the top of the tower is modified, there'd be less churn in
memory.

```
        Hugh
```

=end

#32 - 03/05/2010 05:08 AM - coatl (caleb clausen)

=begin
If String#<< is really O(1), there would seem to be little reason to change anything. I still want to investigate this myself when I get a chance.

I do like the tower of hanoi algorithm hugh pointed out. But it seems like a big change from where ruby's String class is now.
=end

#33 - 03/05/2010 06:09 AM - murphy (Kornelius Kalnbach)

=begin
On 04.03.10 21:08, caleb clausen wrote:

> If String#<< is really O(1), there would seem to be little reason to
> change anything. I still want to investigate this myself when I get a
> chance.
> O(n), where n is the size of the appended string.

But I think it's always worth to look into speedups even if we can't
expect to change the complexity class. The O-factor may not interesting
to theorists, but it matters greatly to programmers. JRuby, for example,
concats strings almost twice as fast in this benchmark:

```
require 'benchmark'

N = 10_000_000
Benchmark.bm 20 do |results|
results.report 'loop' do
N.times { }
end
results.report "'' <<" do
s = ''
N.times { s << '.' << 'word' }
end
end
```

```
ruby19 string_buffer.rb
user     system      total        real
loop                 1.240000   0.010000   1.250000 (  1.255154)
'' <<               5.820000   0.060000   5.880000 (  5.889959)

jruby string_buffer.rb
user     system      total        real
loop                 0.584000   0.000000   0.584000 (  0.488000)
'' <<               2.900000   0.000000   2.900000 (  2.900000)
```

So, there is room for optimization somewhere.

[murphy]

=end

#34 - 03/05/2010 09:12 AM - kstephens (Kurt Stephens)

=begin
+1

Preallocation of String would be immensely useful in large ERB templates.

So much so, I was looking to patching into rb_str_resize(str, len) with a method, to get around related performance issues. Ruby Strings already support the difference between the string length and the allocated buffer size -- we need to expose it and ensure that Strings do not automatically "shrink" the internal String buffers. There should probably be a method to explicitly shrink the internal buffer, if needed.

From what I can tell string growth is roughly O(log2 N) because of the power-of-2 buffer resizing.   For large buffers making this O(1) for large strings helps performance and reduces malloc() memory fragmentation.

=end

#35 - 03/05/2010 11:58 AM - murphy (Kornelius Kalnbach)

=begin
On 05.03.10 01:13, Kurt Stephens wrote:

> Preallocation of String would be immensely useful in large ERB
> templates.
> How big would the buffer size have to be for this template?

<%= link_to @record.name, @record %>

> So much so, I was looking to patching into rb_str_resize(str, len)
> with a method, to get around related performance issues. Ruby
> Strings already support the difference between the string length and
> the allocated buffer size -- we need to expose it and ensure that
> Strings do not automatically "shrink" the internal String buffers.
> There should probably be a method to explicitly shrink the internal
> buffer, if needed.
> This sounds like C to me.
>
> From what I can tell string growth is roughly O(log2 N) because of
> the power-of-2 buffer resizing.
> You probably mean O(N * log2 N). But even in the worst case (smallest
> possible steps, string data must be relocated for each buffer

> extension), it's still just O(N) where N is the length of the final
> string. Example:

```
s = ''
s << '1' # allocate 1 byte, relocate 0 bytes, write 1 byte
s << '2' # allocate 2 bytes, relocate 1 byte, write 1 byte
s << '3' # allocate 4 bytes, relocate 2 bytes, write 1 byte
s << '4' # write 1 byte (buffer is long enough)
s << '5' # allocate 8 bytes, relocate 4 bytes, write 1 byte
...
```

So it's exactly n bytes for the writes, and O(n) bytes must be relocated
in total (about 2*n since sum[i=0..k] $2^i < 2^{k+1}$). Allocation itself
is O(1) for each step.

But I don't say it can't be further optimized in the real world.

> For large buffers making this O(1)
> for large strings helps performance and reduces malloc() memory
> fragmentation.
> Ropes have been mentioned, they provide constant time concatenation, but
> have slower iteration and indexing. They also use more memory.

Is Array#join optimized for the case where all entries are strings? As in:

```
if array.all? { |obj| obj.is_a? String }
buffer_size = array.map { |str| str.size }.sum
else
buffer_size = whatever
end
result = allocate(buffer_size)
array.each { |str| result << str }
```

We could have rope-like performance for concatenation then by using
Arrays, and #join them in linear time to get the final result. Wouldn't
change the complexity, but is probably faster.

[murphy]

=end

#36 - 03/05/2010 05:20 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/3/5 Hugh Sasse hgs@dmu.ac.uk:

>> At first glance, the document explains the difference of destructive
>> and non-destructive concatenations, like String#+ and #<<.
>>
>> It is absolutely different topic from pre-allocation.

> It is related: the algorithm constructs large strings from smaller
> ones in an elegant way using a "tower of Hanoi", and if the top
> string concatenation gets bigger than the one below it, only then
> are they joined together. Result is less copying and merging.

Ah, sorry. I had to read all more carefully.

The algorithm itself is interesting, but I understand it is
just workaround to implement efficient string buffer by using
immutable strings (because Lua String seems always immutable).

But Ruby String is mutable. Is it also more efficient with
mutable string than current direct concatenation? I wonder
if the algorithm needs more memcpy than the current.

--
Yusuke ENDOH mame@tsg.ne.jp

=end

#37 - 03/05/2010 05:58 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/3/5 Kornelius Kalnbach murphy@rubychan.de:

> > Preallocation of String would be immensely useful in large ERB
> > templates.
> > How big would the buffer size have to be for this template?
>
> <%= link_to @record.name, @record %>

Yes, it is generally difficult to determine the size.

We may be able to estimate it by using domain knowledge in some cases.
(e.g., certain page size is empirically known as about 10KB, etc.)
But if the expectation is disappointed, it will cause wasteful memory
allocation or no speed up.

> > So much so, I was looking to patching into rb_str_resize(str, len)
> > with a method, to get around related performance issues. Ruby
> > Strings already support the difference between the string length and
> > the allocated buffer size -- we need to expose it and ensure that
> > Strings do not automatically "shrink" the internal String buffers.
> > There should probably be a method to explicitly shrink the internal
> > buffer, if needed.
> > This sounds like C to me.

Agreed. It is too easy to waste memory.

> But I don't say it can't be further optimized in the real world.

Agreed. So, we need a benchmark to discuss this.

> > For large buffers making this O(1)
> > for large strings helps performance and reduces malloc() memory
> > fragmentation.
> > Ropes have been mentioned, they provide constant time concatenation, but
> > have slower iteration and indexing. They also use more memory.
>
> Is Array#join optimized for the case where all entries are strings?

I think Array#join already does so.

Thank you very much for saying almost all I want to say :-)

--
Yusuke ENDOH mame@tsg.ne.jp

=end

#38 - 03/05/2010 07:07 PM - hgs (Hugh Sasse)

=begin
On Fri, 5 Mar 2010, Yusuke ENDOH wrote:

> Hi,
>
> 2010/3/5 Hugh Sasse hgs@dmu.ac.uk:
>
> > At first glance, the document explains the difference of destructive
> > and non-destructive concatenations, like String#+ and #<<.
> >
> > It is absolutely different topic from pre-allocation.

> > It is related: the algorithm constructs large strings from smaller
> > ones in an elegant way using a "tower of Hanoi", and if the top
> > string concatenation gets bigger than the one below it, only then
> > are they joined together. Result is less copying and merging.
>
> Ah, sorry. I had to read all more carefully.
>
> The algorithm itself is interesting, but I understand it is
> just workaround to implement efficient string buffer by using
> *immutable* strings (because Lua String seems always immutable).
>
> But Ruby String is mutable. Is it also more efficient with
> *mutable* string than current direct concatenation? I wonder
> if the algorithm needs more memcpy than the current.

Possibly. I've not gone into this in much depth. I thought it
might be helpful to raise it in case this would give significant
help to garbage collection. I'm thinking that as the strings get
longer they fill up space in the heap so need to be moved to the
newly allocated space. Dealing with only the top of the "tower of
Hanoi" would be handling smaller chunks. I think this would need to
be tested, but could be worth exploring. Lua is rather quick, and
the article talks about a big speed increase.

On the other hand, it is difficult to decide when to invoke this
algorithm. It is probably too heavy for just joining two strings,
but for reading in lots of chunks and appending them, it could be a
big help. I don't know how to detect that distinction in user code.
It might be too much work.

```
      Hugh
```

> --
> Yusuke ENDOH mame@tsg.ne.jp

=end

#39 - 03/06/2010 02:25 AM - now (Nikolai Weibull)

=begin
On Fri, Mar 5, 2010 at 17:25, Caleb Clausen vikkous@gmail.com wrote:

> On 3/5/10, Yusuke ENDOH mame@tsg.ne.jp wrote:
>
> > 2010/3/5 Kornelius Kalnbach murphy@rubychan.de:
> >
> > > How big would the buffer size have to be for this template?
> > >
> > > <%= link_to @record.name, @record %>
> >
> > Yes, it is generally difficult to determine the size.
> >
> > We may be able to estimate it by using domain knowledge in some cases.
> > (e.g., certain page size is empirically known as about 10KB, etc.)
> > But if the expectation is disappointed, it will cause wasteful memory
> > allocation or no speed up.
>
> Generally, a given template should expand to about the same size every
> time.

I'm getting the feeling thath the only real use case that we've got
for this so far is ERb. Wouldn't it make more sense to change the way
ERb (and similar "string concatenators") creates its result?

=end

#40 - 03/06/2010 07:44 AM - murphy (Kornelius Kalnbach)

=begin
On 05.03.10 18:25, Nikolai Weibull wrote:

> I'm getting the feeling thath the only real use case that we've got
> for this so far is ERb. Wouldn't it make more sense to change the way
> ERb (and similar "string concatenators") creates its result?
> How about an optimized StringBuffer class in stdlib that's optimized for
> this kind of stuff? But only if we really find a way to speed it up.

[murphy]

=end

#41 - 03/06/2010 10:26 AM - murphy (Kornelius Kalnbach)

=begin
On 06.03.10 01:31, Kurt Stephens wrote:

> ERB template rendering is one of my greatest performance issues right now.
> Have you really identified String concatenation as the primary issue?
> There's so much more going on when building a template (especially in
> Rails).

Somehow, my feeling is that the actual concatenation of a small string
takes even less time than the calling overhead of String#<< (accessing
self, method lookup, checking arguments, returning the recipient, ...)
We could be talking about, say, 2% of the time your template needs to
compile.

By the way, fact check: ERb really uses String#<<, right?
[murphy]

=end

#42 - 03/06/2010 11:49 AM - murphy (Kornelius Kalnbach)

- File string_buffer.diff added

=begin
Here's a patch that doesn't work. I don't know what I'm doing wrong here: RESIZE_CAPA seemed just right.

Any hints?
=end

#43 - 03/07/2010 05:44 AM - mame (Yusuke Endoh)

=begin
Hi,

2010/3/6 Kornelius Kalnbach redmine@ruby-lang.org:

> Here's a patch that doesn't work. I don't know what I'm doing wrong here: RESIZE_CAPA seemed just right.

Thank you for your writing a patch!
It seems to work on my environment. What made you think it does
not work?

I confirmed it by the following program:

```
opt = false
s = ""
t = "x" * 1_000_000
s.buffer(100_000_000) if opt
100.times { s << t }
p s.size
```

The above program takes 0.205 sec. when opt is false, and takes
0.195 sec. when opt is true.

But this is artificial example with very big string (1 GB).
The following more realistic case (with 100 KB):

```
opt = false
1000.times do
s = ""
s.buffer(opt ? 100_001 : 100)
x = "x" * 1000
100.times { s << x }
end
```

takes 0.115 sec. when opt is false, 0.130 sec. when opt is true.
I don't know why it becomes slower, but the story seems not to be
so simple.

Anyway, the overhead of concatenation seems not so big. I doubt
if it is the bottleneck.

--
Yusuke ENDOH mame@tsg.ne.jp

=end

#44 - 03/07/2010 08:29 AM - murphy (Kornelius Kalnbach)

=begin
On 06.03.10 21:44, Yusuke ENDOH wrote:

> 2010/3/6 Kornelius Kalnbach redmine@ruby-lang.org:
>
>> Here's a patch that doesn't work. I don't know what I'm doing wrong here: RESIZE_CAPA seemed just right.
>> Thank you for your writing a patch!
>> It seems to work on my environment. What made you think it does
>> not work?
>> The fact that the memory taken by the Ruby process didn't change in top.
>> I requested a 200MB buffer, and the process was still at 2.8MB.
>
> Anyway, the overhead of concatenation seems not so big. I doubt
> if it is the bottleneck.
> That's my conclusion, too. But the JRuby team seems to have seen some
> 10% speedup:

http://gist.github.com/323431 - without and with preset buffer

Maybe the question is, is it worth it?

[murphy]

=end

#45 - 03/07/2010 02:57 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/3/7 Kornelius Kalnbach murphy@rubychan.de:

> On 06.03.10 21:44, Yusuke ENDOH wrote:
>
>> 2010/3/6 Kornelius Kalnbach redmine@ruby-lang.org:
>>
>>> Here's a patch that doesn't work. I don't know what I'm doing wrong here: RESIZE_CAPA seemed just right.
>>> Thank you for your writing a patch!
>>> It seems to work on my environment. What made you think it does
>>> not work?
>>> The fact that the memory taken by the Ruby process didn't change in top.
>>> I requested a 200MB buffer, and the process was still at 2.8MB.

Hmm, I guess you saw physical memory size allocated.
On many platform, physical memory is not allocated until
writing into the page actually occurs.

If you use Linux, see virtual memory size (VSZ column of

ps command), instead of %MEM. It would reflect your huge
allocation.

The performance may be improved by using madvise, but I
don't think it should be supported by ruby core.

--
Yusuke ENDOH mame@tsg.ne.jp

=end

#46 - 03/07/2010 06:38 PM - kosaki (Motohiro KOSAKI)

=begin
Hi

At least, Linux madvise doesn't improve the performance in such case. current cruby + linux(glibc) realloc implementation makes very optimal behavior.
a big size string makes a big size realloc() and a big size realloc() is using mremap(2) internally. Then, realloc() doesn't makes string copy at all.

IOW, the main benefit of string.buffer() is to reduce realloc() cost. but it is already zero. so I don't think it is worth method. sadly almost developers never use such no improve method, I expect.

Instead, I would propose improve JRuby's internal string representation and string concat implementation.

Thanks.
=end

#47 - 03/07/2010 06:47 PM - wanabe (_ wanabe)

=begin
Hi,

```
opt = false
1000.times do
s = ""
s.buffer(opt ? 100_001 : 100)
x = "x" * 1000
100.times { s << x }
end

takes 0.115 sec. when opt is false, 0.130 sec. when opt is true.
```

I tried too.
Interestingly, it gets faster on my environment.

```
$ cat test.rb
require 'benchmark'
opt = ARGV[0]
list = Array.new(10) do
Benchmark.realtime do
1000.times do
s = ""
s.buffer(opt ? 100_001 : 100)
x = "x" * 1000
100.times { s << x }
end
end
end
list.sort!
p list.first, list.last

$ ./ruby -v -Ilib test.rb opt
ruby 1.9.2dev (2010-03-07 trunk 26843) [i386-mingw32]
0.1780099868774414
0.18601107597351074

$ ./ruby -v -Ilib test.rb
ruby 1.9.2dev (2010-03-07 trunk 26843) [i386-mingw32]
0.21401190757751465
0.22301316261291504
```

But, I guess, the patch may not work as expected in some cases.
Some methods (String#succ!, sub!, []=, and so on) can let CAPA shrink.

=end

#48 - 03/07/2010 07:58 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/3/5 Kornelius Kalnbach murphy@rubychan.de:

> JRuby, for example,
> concats strings almost twice as fast in this benchmark:
>
> ```
> require 'benchmark'
>
> N = 10_000_000
> Benchmark.bm 20 do |results|
> results.report 'loop' do
> N.times { }
> end
> results.report "'' <<" do
> s = ''
> N.times { s << '.' << 'word' }
> end
> end
>
> ruby19 string_buffer.rb
> user     system      total        real
> loop                 1.240000   0.010000   1.250000 (  1.255154)
> '' <<                5.820000   0.060000   5.880000 (  5.889959)
>
> jruby string_buffer.rb
> user     system      total        real
> loop                 0.584000   0.000000   0.584000 (  0.488000)
> '' <<                2.900000   0.000000   2.900000 (  2.900000)
> ```

I wonder why such a simple loop is slower than jruby...?

I retested.

```
ruby19
user     system      total        real
loop                 2.100000   0.000000   2.100000 (  2.095623)
'' <<               11.720000   0.040000  11.760000 ( 11.768111)

jruby
user     system      total        real
loop                 2.263000   0.000000   2.263000 (  2.228000)
'' <<               10.193000   0.000000  10.193000 ( 10.193000)
```

Ko1 told me that GC makes the second benchmark slower than JRuby.
In MRI, a string literal is duplicated whenever evaluated.
I moved the literals out of the loop:

```
results.report "'' <<" do
s = ''
s1, s2 = '.', 'word'
N.times { s << s1 << s2 }
end

ruby19
user     system      total        real
'' <<                6.810000   0.040000   6.850000 (  6.851979)

jruby
user     system      total        real
'' <<                7.159000   0.000000   7.159000 (  7.126000)
```

Indeed, there is room for optimization in MRI, but in this case,
it is not in string concatenation, I guess.

--
Yusuke ENDOH mame@tsg.ne.jp

=end

#49 - 03/07/2010 08:46 PM - kosaki (Motohiro KOSAKI)

=begin

```
$ cat test.rb
require 'benchmark'
opt = ARGV[0]
list = Array.new(10) do
Benchmark.realtime do
1000.times do
s = ""
s.buffer(opt ? 100_001 : 100)
x = "x" * 1000
100.times { s << x }
end
end
end
list.sort!
p list.first, list.last

$ ./ruby -v -Ilib test.rb opt
ruby 1.9.2dev (2010-03-07 trunk 26843) [i386-mingw32]
0.1780099868774414
0.18601107597351074

$ ./ruby -v -Ilib test.rb
ruby 1.9.2dev (2010-03-07 trunk 26843) [i386-mingw32]
0.21401190757751465
0.22301316261291504
```

Ah, yes. "x" * 1000 is not so big string. then, its realloc() doesn't use mremap.
It mean string concat(i.e. "<<" operator) cause string copy on each time. but is this real issue? Does small string copy makes big peformance issue? when? So, I think we need good realistic benchmark.

Thanks.
=end

#50 - 03/07/2010 10:21 PM - murphy (Kornelius Kalnbach)

=begin
On 07.03.10 06:57, Yusuke ENDOH wrote:

Hmm, I guess you saw physical memory size allocated.
On many platform, physical memory is not allocated until
writing into the page actually occurs.
I didn't know that. Thanks!
[murphy]

=end

#51 - 03/08/2010 12:34 AM - headius (Charles Nutter)

=begin
On Sun, Mar 7, 2010 at 4:58 AM, Yusuke ENDOH mame@tsg.ne.jp wrote:

Ko1 told me that GC makes the second benchmark slower than JRuby.
In MRI, a string literal is duplicated whenever evaluated.
I moved the literals out of the loop:

JRuby behaves the same, since literal strings are still separate objects and mutable.

```
 results.report "'' <<" do
   s = ''
   s1, s2 = '.', 'word'
   N.times { s << s1 << s2 }
 end

 ruby19
                  user     system      total        real
 '' <<        6.810000   0.040000   6.850000 (  6.851979)
```

```
jruby
                      user     system      total        real
'' <<             7.159000   0.000000   7.159000 (  7.126000)
```

Indeed, there is room for optimization in MRI, but in this case,
it is not in string concatenation, I guess.

My numbers came out somewhat differently. Make sure you're running
with the JVM's "server" mode if you run on Hotspot (Sun/OpenJDK):

```
~/projects/jruby  jruby --server string_bench.rb
user     system      total        real
loop              0.572000   0.000000   0.572000 (  0.523000)
'' <<             1.470000   0.000000   1.470000 (  1.470000)
```

```
~/projects/jruby  ruby1.9 string_bench.rb
user     system      total        real
loop              0.810000   0.000000   0.810000 (  0.838414)
'' <<             2.670000   0.040000   2.710000 (  2.733041)
```

Here's numbers with a prototypical String.buffer implementation:

```
~/projects/jruby  jruby --server string_bench.rb
user     system      total        real
loop              0.655000   0.000000   0.655000 (  0.606000)
'' <<             1.390000   0.000000   1.390000 (  1.390000)
user     system      total        real
loop              0.321000   0.000000   0.321000 (  0.321000)
'' <<             1.241000   0.000000   1.241000 (  1.241000)
user     system      total        real
loop              0.314000   0.000000   0.314000 (  0.314000)
'' <<             1.229000   0.000000   1.229000 (  1.229000)
```

Of course, this 10-15% improvement could simply be because the JVM
does not provide a "realloc" for its arrays (for various reasons, some
of them presumably because it moves objects around in memory a lot).
In order to grow a string, we have to allocate a new array and copy
its contents. Under those circumstances, String.buffer makes a lot of
sense, since the copying can get expensive at large sizes.

I don't know enough about MRI internals to implement an equivalent
String.buffer, but here's the patch to JRuby:

diff --git a/src/org/jruby/RubyString.java b/src/org/jruby/RubyString.java
index 71e6b63..e618ec8 100644
--- a/src/org/jruby/RubyString.java
+++ b/src/org/jruby/RubyString.java
@@ -451,6 +451,11 @@ public class RubyString extends RubyObject
implements EncodingCapable {
public static RubyString newStringLight(Ruby runtime, int size) {
return new RubyString(runtime, runtime.getString(), new
ByteList(size), false);
}
+

- @JRubyMethod(meta = true)
- public static IRubyObject buffer(ThreadContext context, IRubyObject self, IRubyObject size) {
- return newStringLight(context.getRuntime(), (int)size.convertToInteger().getLongValue());

- }

  public static RubyString newString(Ruby runtime, CharSequence str) {
  return new RubyString(runtime, runtime.getString(), str);

=end

#52 - 03/08/2010 12:40 PM - mame (Yusuke Endoh)

=begin
Hi,

2010/3/8 Charles Oliver Nutter headius@headius.com:

> > Indeed, there is room for optimization in MRI, but in this case,
> > it is not in string concatenation, I guess.
>
> My numbers came out somewhat differently. Make sure you're running
> with the JVM's "server" mode if you run on Hotspot (Sun/OpenJDK):

Ah, I didn't specify the option:

```
                           user      system       total         real
```

```
loop            1.471000   0.000000   1.471000 (  1.248000)
" <<            5.906000   0.000000   5.906000 (  5.906000)
```

JRuby is great :-)

> Here's numbers with a prototypical String.buffer implementation:
>
> *snip*
>
> Of course, this 10-15% improvement could simply be because the JVM
> does not provide a "realloc" for its arrays (for various reasons, some
> of them presumably because it moves objects around in memory a lot).
> In order to grow a string, we have to allocate a new array and copy
> its contents. Under those circumstances, String.buffer makes a lot of
> sense, since the copying can get expensive at large sizes.

Ok, we finally grasped the situation. To sum up:

- This feature is meaningless with MRI, at least, on Linux.
- But it serves as a workaround for slow string concatenation of JRuby that cannot be optimized due to JVM.
- Does MRI provide the feature just for script compatibility?

I cannot make the judgment. Please wait for matz.

--
Yusuke ENDOH mame@tsg.ne.jp

=end

#53 - 04/02/2010 08:10 AM - znz (Kazuhiro NISHIYAMA)

- Target version set to 2.0.0

=begin

=end

#54 - 02/08/2012 03:15 AM - kosaki (Motohiro KOSAKI)

Matz, should we close this ticket?

#55 - 05/26/2012 04:25 AM - Anonymous

Uh oh, this discussion is already a pain to read.

#56 - 09/19/2012 04:17 PM - headius (Charles Nutter)

Trying to wake this beast up...

mame: I don't think we can say it would not help MRI without testing an implementation, can we? I misunderstood realloc in my comment from two years (!!!) ago According to realloc docs:

```
 The realloc() function tries to change the size of the allocation pointed to by ptr to size, and returns ptr.
  If there is not enough room to enlarge the memory allocation pointed
 to by ptr, realloc() creates a new allocation, copies as much of the old data pointed to by ptr as will fit t
o the new allocation, frees the old allocation, and returns a pointer to
 the allocated memory.
```

This seems to indicate that except under rare circumstances where the memory after the pointer is known to be free, realloc will behave exactly like the JVM, creating a new pointer, copying data, and freeing the old pointer.

To me, this means that a pre-allocated String construction method is most definitely useful.

It also occurred to me recently that String.new does not accept an integer argument. Perhaps all we need to do is add a String.new form that takes Integer, and possibly an optional fill byte/codepoint/single-char string?

#57 - 09/19/2012 04:36 PM - shyouhei (Shyouhei Urabe)

Just a technical comment, not for the feature itself:

headius (Charles Nutter) wrote:

```
 to by ptr, realloc() creates a new allocation, copies as much of the old data
```

This "copy" is done by mremap(2) system call, which just reassembles OS's process-private virtual memory map to move a region of memory to another, in O(1). That is what mame said in "This feature is meaningless with MRI, at least, on Linux."

#58 - 10/15/2012 05:02 AM - headius (Charles Nutter)

I do not believe for a moment that realloc or mremap can in all cases perform the operation in O(1) time, and the docs seem to agree with me...first based on the doc above for realloc, and then for this doc on mremap:

```
   MREMAP_MAYMOVE
          By default, if there is not sufficient space to expand a mapping at its current location, then mremap() fails.  If this  flag
          is specified, then the kernel is permitted to relocate the mapping to a new virtual address, if necessary.  If the mapping is
          relocated, then absolute pointers into the old mapping location become invalid (offsets relative to the starting  address  of
          the mapping should be employed).
```

It seems to me that preallocation is most definitely useful, even in the presence of realloc and mremap. I would like to see it added.

#59 - 10/27/2012 04:39 AM - ko1 (Koichi Sasada)

Who can judge this ticket?
I can't understand this issue because there is long discussion.
Could anyone summarize a conclusion?

#60 - 10/27/2012 10:44 AM - mame (Yusuke Endoh)

Hello headius,

headius (Charles Nutter) wrote:

> Trying to wake this beast up...
>
> mame: I don't think we can say it would not help MRI without testing an implementation, can we? I misunderstood realloc in my comment from two years (!!!) ago According to realloc docs:

Linux's realloc(3) man-page does NOT say that.

http://www.kernel.org/doc/man-pages/online/pages/man3/malloc.3.html

Perhaps you saw os x's realloc?
I wonder this issue is valid on os x.
Anyone can conduct a quantitative investigation?

headius (Charles Nutter) wrote:

> I do not believe for a moment that realloc or mremap can in all cases perform the operation in O(1) time, and the docs seem to agree with me...first based on the doc above for realloc, and then for this doc on mremap:

Looks irrelevant. I guess realloc(3) just uses mremap with MREMAP_MAYMOVE
internally.

--
Yusuke Endoh mame@tsg.ne.jp

#61 - 10/27/2012 10:50 AM - mame (Yusuke Endoh)

ko1 (Koichi Sasada) wrote:

> Who can judge this ticket?
> I can't understand this issue because there is long discussion.

> Could anyone summarize a conclusion?

Not concluded, but currently we know:

- This feature provides "a Ruby-level workaround" for a poor realloc implementation on some runtime, such as JVM, and possibly os x.
- But at least, Linux (precisely, libc?)'s realloc is well implemented. So this feature is meaningless in practice, in such environment.

--
Yusuke Endoh mame@tsg.ne.jp

#62 - 10/27/2012 12:02 PM - mame (Yusuke Endoh)

- Target version changed from 2.0.0 to 2.6

#63 - 10/27/2012 03:51 PM - headius (Charles Nutter)

mame: I do not understand how there's any way Linux would be different from any other platform. If there's no room in contiguous memory to expand a pointer, the data must be moved elsewhere in memory. Am I missing something?

#64 - 10/27/2012 07:33 PM - mame (Yusuke Endoh)

headius (Charles Nutter) wrote:

> mame: I do not understand how there's any way Linux would be different from any other platform. If there's no room in contiguous memory to expand a pointer, the data must be moved elsewhere in memory. Am I missing something?

Almost all recent practical operating systems are using the virtual memory mechanism.

http://en.wikipedia.org/wiki/Virtual_memory

In the OS based on the mechanism, there is a mapping from virtual memory addresses to physical ones.
By changing the map, contiguous virtual memory addresses can be (re)assigned without moving physical memory data.
(This is why the system call in question is named "remap", I think)

--
Yusuke Endoh mame@tsg.ne.jp

#65 - 11/01/2012 05:06 AM - headius (Charles Nutter)

So we have something like this:

Platforms known to not support any sort of O(1) realloc: JVM

Platforms that may not support O(1) realloc: OS X, others?

Platforms that do (should?) support O(1) realloc: Linux

In any case, I still see that there's value in this feature:

- It would help JRuby and all runtimes that run on non-efficient-realloc platforms.
- It does no harm and matches Array.new behavior.
- For folks doing crypto stuff that want to know exactly how big the buffer is right away, this provides a way to do so.

I won't try to argue whether realloc is consistently efficient across platforms or not. It seems like it's not guaranteed to be on any platform.

It's also such a tiny addition...why not?

#66 - 11/01/2012 05:53 AM - kosaki (Motohiro KOSAKI)

> So we have something like this:
>
> Platforms known to not support any sort of O(1) realloc: JVM
>
> Platforms that may not support O(1) realloc: OS X, others?
>
> Platforms that do (should?) support O(1) realloc: Linux

> In any case, I still see that there's value in this feature:
>
> - It would help JRuby and all runtimes that run on non-efficient-realloc platforms.
> - It does no harm and matches Array.new behavior.
> - For folks doing crypto stuff that want to know exactly how big the buffer is right away, this provides a way to do so.
>
> I won't try to argue whether realloc is consistently efficient across platforms or not. It seems like it's not guaranteed to be on any platform.
>
> It's also such a tiny addition...why not?

I don't imagine a lot of people take a string.buffer game for
optimization if it doesn't
have big benefit. now, this feature is unclear how much useful out of
jvm and how
much useful on jvm. afaik, nobody show realistic benchmark result nor
encompassing
affect platform lists. I'm not incline to agree guess game.

If the benefit is not so much, the feature will be dead and forgotten quickly.

#67 - 11/01/2012 05:53 AM - normalperson (Eric Wong)

"headius (Charles Nutter)" headius@headius.com wrote:

> - For folks doing crypto stuff that want to know exactly how big the buffer is right away, this provides a way to do so.

I'm not sure exactly what you mean. Do you mean to avoid leaving
sensitive data in the heap from realloc()? Yes it would help, but
I think this is a poor API for that purpose.

Perhaps special methods like String#secure_cat and String#secure_wipe
is more obvious for security-concious users.

> I won't try to argue whether realloc is consistently efficient across
> platforms or not. It seems like it's not guaranteed to be on any
> platform.

I absolutely agree this can help performance regardless of platform,
however...

> It's also such a tiny addition...why not?

I'm not a VM expert, but shouldn't it be possible for the VM to track
the growth of strings allocated at different call sites and
automatically optimize preallocations as time goes on?

#68 - 11/01/2012 09:30 AM - headius (Charles Nutter)

On Wed, Oct 31, 2012 at 3:43 PM, Eric Wong normalperson@yhbt.net wrote:

> "headius (Charles Nutter)" headius@headius.com wrote:
>
> > - For folks doing crypto stuff that want to know exactly how big the buffer is right away, this provides a way to do so.
>
> I'm not sure exactly what you mean. Do you mean to avoid leaving
> sensitive data in the heap from realloc()? Yes it would help, but
> I think this is a poor API for that purpose.

For security, you don't want strings to be growing and copying stuff
around in memory, so being able to allocate a specific size ahead of
time is useful.

> Perhaps special methods like String#secure_cat and String#secure_wipe
> is more obvious for security-concious users.

And if secure_cat didn't use realloc (because it could leave sensitive
data on the heap) you'd *still* have a need to preallocate what you
need. That doesn't solve anyhting.

> > I won't try to argue whether realloc is consistently efficient across
> > platforms or not. It seems like it's not guaranteed to be on any
> > platform.
>
> I absolutely agree this can help performance regardless of platform,
> however...
>
> > It's also such a tiny addition...why not?
>
> I'm not a VM expert, but shouldn't it be possible for the VM to track
> the growth of strings allocated at different call sites and
> automatically optimize preallocations as time goes on?

A sufficiently smart compiler can do anything, of course. However I
know of no VMs that track the eventual size of objects allocated at a
given call site and eagerly allocate that memory, and such an
optimization would be very tricky to do right.

- Charlie

#69 - 01/26/2016 11:31 PM - mame (Yusuke Endoh)

- Related to Feature #12024: Add String.buffer, for creating strings with large capacities added

#70 - 07/21/2016 07:22 PM - headius (Charles Nutter)

I accept String.new(capacity: size) as an acceptable implementation of this request.

#71 - 07/26/2016 07:33 AM - shyouhei (Shyouhei Urabe)

- Status changed from Feedback to Closed

Closing. Please use String.new with capacity.

Files

string_buffer.diff	846 Bytes	03/06/2010	murphy (Kornelius Kalnbach)