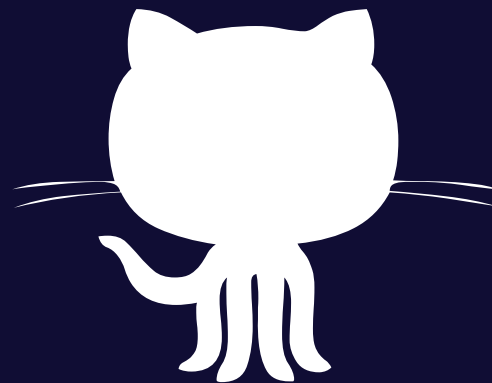
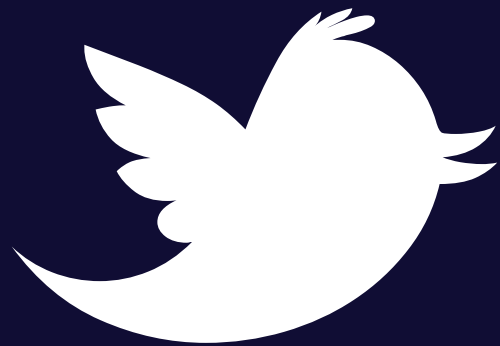


zomgscale!

with Celluloid and JRuby

Ben Lovell

benlovell







queen





dating platform

20m members

10k partner sites



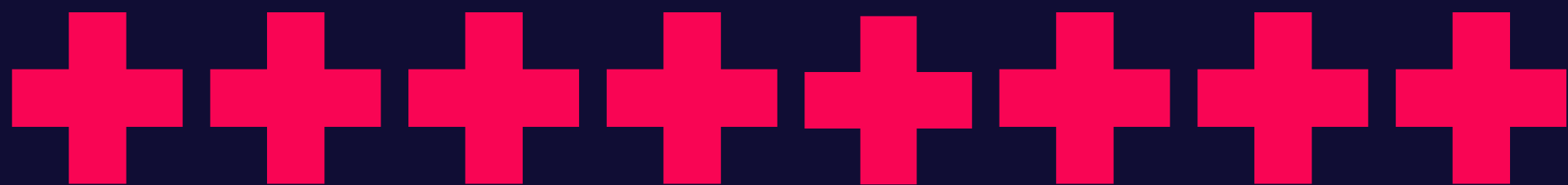
we're hiring
globaldev
isn't everyone?

Moore's Law

every few years
CPU clock speed
has doubled

but recently
the growth
has stalled

cores



the free lunch
is over
Herb Sutter

harness the
POWER
in those cores



concurrency

parallelism

so there's a
difference?



processes
or
threads

processes

x cores == x processes?

memory constraints

communication

processes

what about fork (2)
and CoW friendly GC
?



threads

sharing state

locks and granularity

races

hard to reason

zomg! I <3

multithreaded code

NOBODY. EVER.

but there are ways
to mitigate the madness

**don't communicate by
sharing memory...**

**...share memory
by communicating**

go





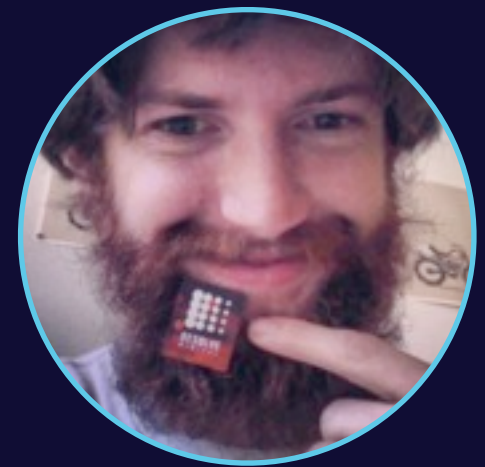
painless multithreaded programming for ruby

The Maintainers



Tony Arcieri
@bascule

Tim Carey-Smith
@halorgium



Ben Langfeld
@benlangfeld

**a concurrent object oriented
programming framework which
lets you build multithreaded
programs out of concurrent
objects just as easily as you
build sequential programs
out of regular objects**

**based upon the
actor model**

actor model

first proposed

way back in **1970**

actor model

actors are isolated within
lightweight processes

actor model

actors possess

identity

actor model

absolutely

no shared state

actor model

**actors don't need to
compete for locks**

actor model

are sent messages

asynchronously

actor model

messages are

buffered by a mailbox

actor model

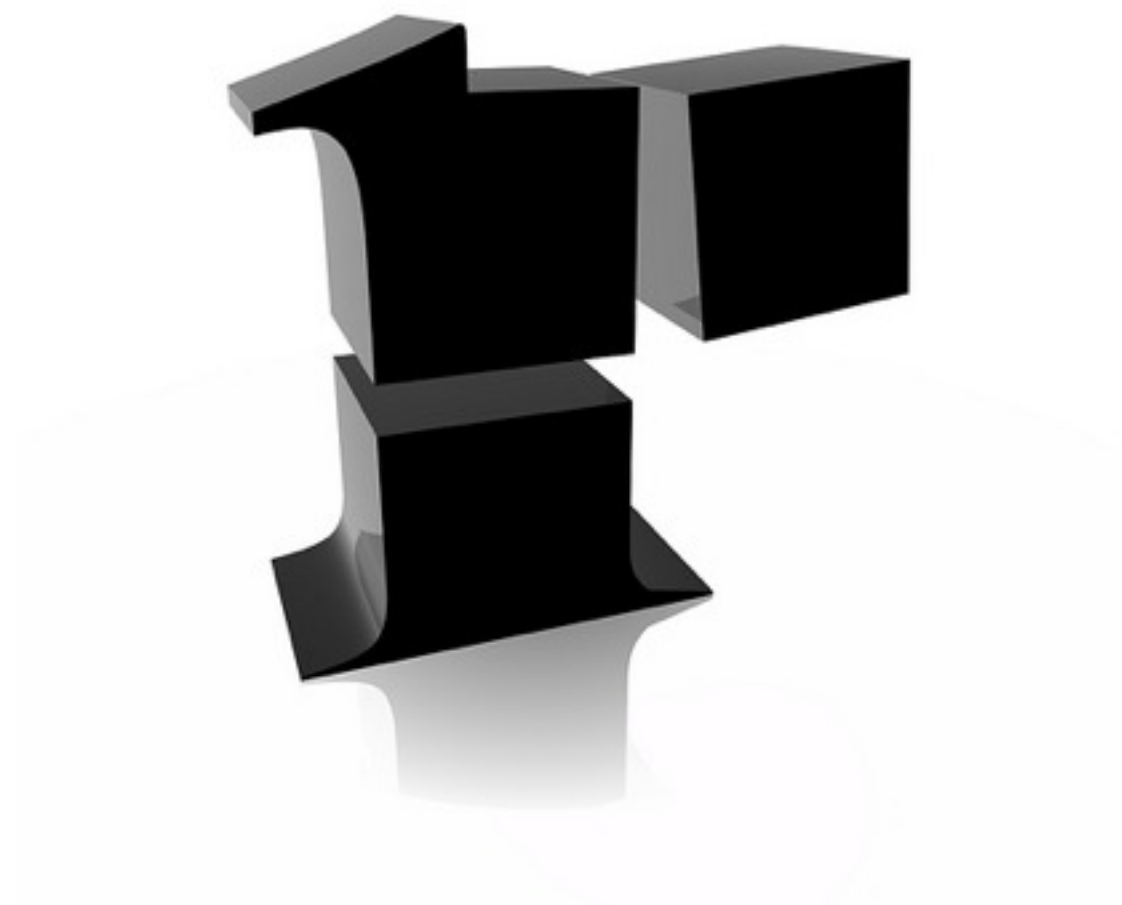
**the actor works off each
message sequentially**

actor model

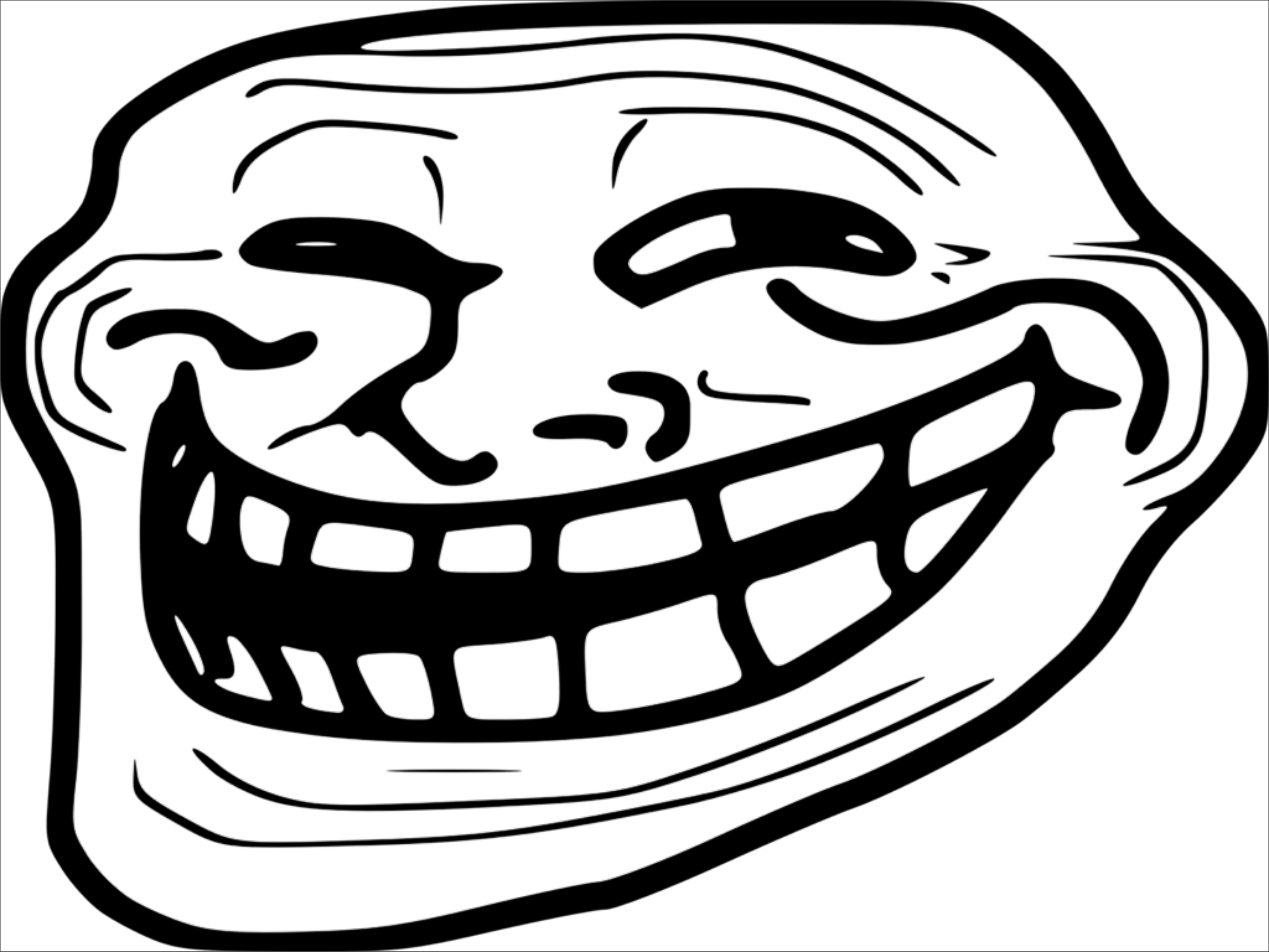
has implementations
in many languages













celluloid actors

automatically

synchronize state

```
1 class Actor
2   attr_reader :counter
3
4   def initialize
5     @counter = 0
6     @mutex = Mutex.new
7   end
8
9   def increment
10    @mutex.synchronize do
11      @counter += 1
12    end
13  end
14 end
```

with celluloid

the same example...

```
1 require "celluloid"
2
3 class Actor
4   include Celluloid
5   attr_reader :counter
6
7   def initialize
8     @counter = 0
9   end
10
11   def increment
12     @counter += 1
13   end
14 end
```



celluloid actors
are active objects
living within threads

```
1 require "celluloid"
2
3 class Actor
4   include Celluloid
5 end
6
7 actor = Actor.new
8 actor.inspect
9 #=> <Celluloid::ActorProxy(Actor:0x3feaecbb38e0)>
10
11 Thread.main
12 #=> <Thread:0x007f86290b8ce8 run>
13
14 actor.thread
15 #=> <Thread:0x007f862ad27a78 sleep>
```

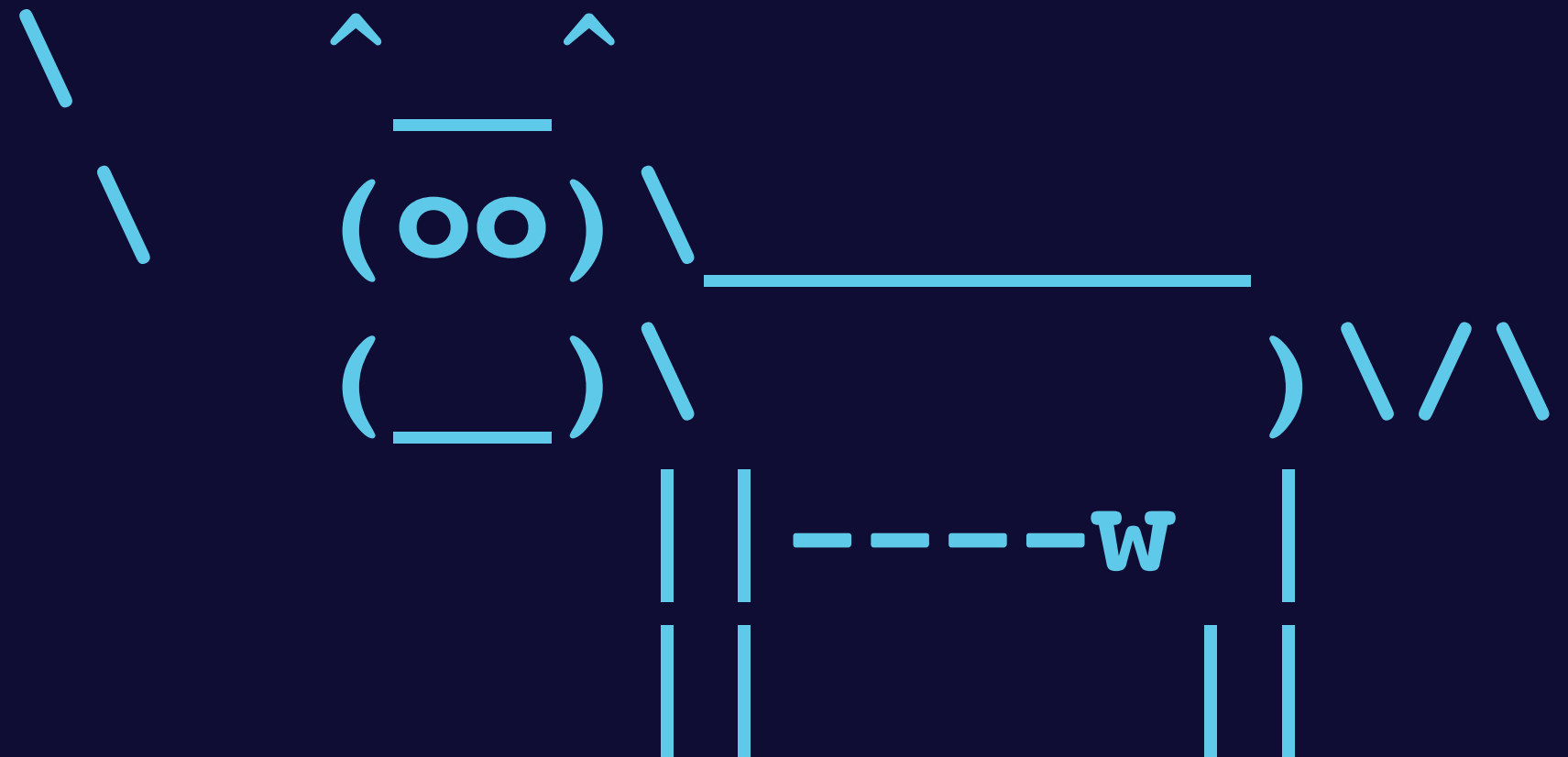
```
1 module Celluloid
2   module ClassMethods
3     # Create a new actor
4     def new(*args, &block)
5       proxy = Actor.new(allocate, actor_options).proxy
6       proxy._send_(:initialize, *args, &block)
7       proxy
8     end
9     # ...
10  end
11  # ...
12 end
```

celluloid actors

**messages you send
are buffered via the
actor's mailbox...**

celluloid actors

... until the **actor** is
ready to act upon them

[illegible]

celluloid actors

there is no

pattern matching

just regular **method calls**

celluloid actors
poll their mailbox
via a message loop

```
1 class Actor
2   # Wrap the given subject with an Actor
3   def initialize(subject, options = {})
4     @subject = subject
5     @mailbox = options[:mailbox] || Mailbox.new
6     @running = true
7
8     @thread = ThreadHandle.new(:actor) do
9       setup_thread
10      run
11    end
12    #...
13  end
14  #...
15 end
```

```
1 class Actor
2   def run
3     #...
4     while @running
5       if message = @mailbox.receive(timeout_interval)
6         handle_message message
7       else
8         # No message indicates a timeout
9         @timers.fire
10        @receivers.fire_timers
11      end
12    end
13    #...
14    shutdown
15  end
16 end
```

celluloid actors

act upon messages

sequentially

what about ordering?

no guarantees

celluloid actors

dispatch calls

within **fibers**

fibers?

cooperative

lightweight

user space

some gotchas...

celluloid actors
can dispatch
synchronously

```
1 require "celluloid"
2
3 class Actor
4   include Celluloid
5
6   def compute_all_the_things
7     sleep 2
8     puts "42"
9   end
10 end
11
12 actor = Actor.new
13 actor.compute_all_the_things
14 puts "done!"

#=> 42
#=> done!
```

blocking



celluloid actors

can dispatch

asynchronously

```
1 require "celluloid"
2
3 class Actor
4   include Celluloid
5
6   def compute_all_the_things
7     sleep 2
8     puts "42"
9   end
10 end
11
12 actor = Actor.new
13 actor.async.compute_all_the_things
14 puts "done!"
15
16 #=> done!
17 #=> 42
```

returns
immediately



celluloid actors
can perform tasks
in futures

```
1 require "celluloid"
2
3 class Actor
4   include Celluloid
5
6   def compute_all_the_things
7     sleep 2
8     "42"
9   end
10 end
11
12 actor = Actor.new
13 future = actor.future.compute_all_the_things
14 puts "done!"
15 puts future.value
16
17 #=> done!
18 #=> 42
```

returns immediately



blocks until a value is yielded



celluloid actors
are accessible by
reference or name

```
1 require "celluloid"
2
3 class Actor
4   include Celluloid
5
6   def compute_all_the_things
7     sleep 2
8     puts "42"
9   end
10 end
11
12 actor = Actor.new
13 Celluloid::Actor[:foo] = actor
14
15 actor.inspect
16 #=> <Celluloid::ActorProxy(Actor:0x3feb3ec11308)>
17 Celluloid::Actor[:foo].inspect
18 #=> <Celluloid::ActorProxy(Actor:0x3feb3ec11308)>
```

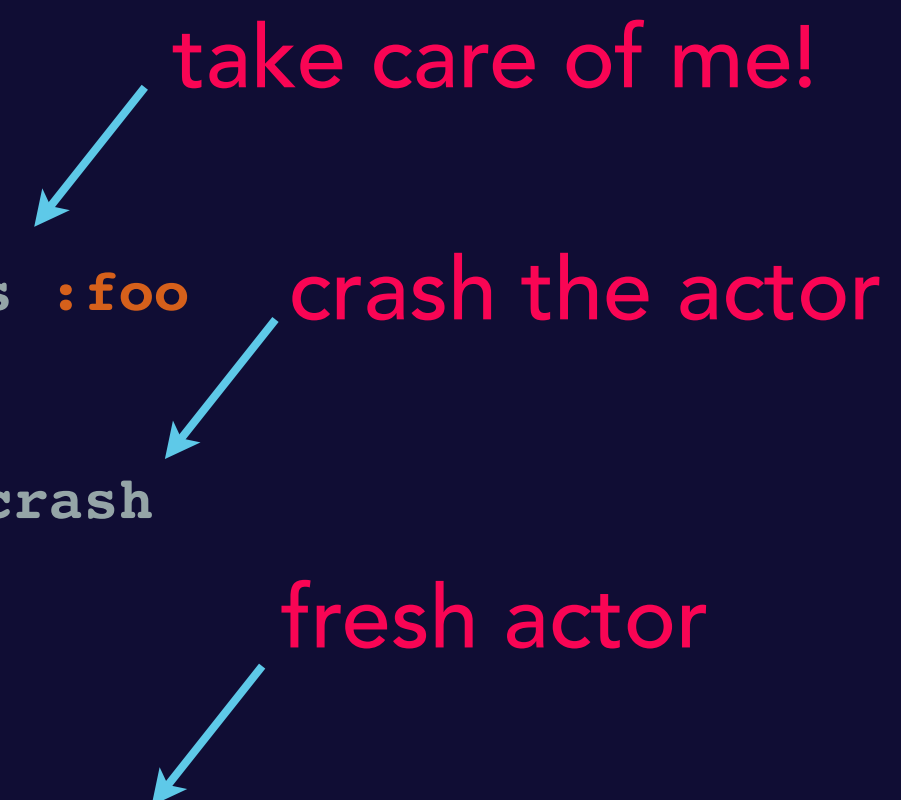
celluloid actors
are fault tolerant
... let it crash!

```
1 require "celluloid/autostart"
2
3 class Actor
4   include Celluloid
5
6   def compute_all_the_things
7     puts "42"
8   end
9
10  def zomg_crash
11    raise "derp!"
12  end
13 end
14
15 supervisor = Actor.supervise_as :foo
16
17 begin
18   Celluloid::Actor[:foo].zomg_crash
19 rescue
20   puts "whoops"
21 end
22
23 Celluloid::Actor[:foo].compute_all_the_things
24
25 #=> whoops
26 #=> 42
```

take care of me!

crash the actor

fresh actor

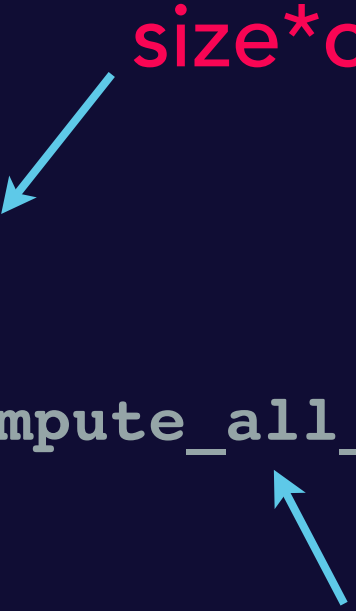


celluloid actors
can be arranged
as **pooled workers**

```
1 require "celluloid"
2
3 class Actor
4   include Celluloid
5
6   def compute_all_the_things
7     sleep 1
8     puts "42"
9   end
10 end
11
12 pool = Actor.pool
13
14 4.times { pool.compute_all_the_things }
15
16 #=> 42
17 #=> 42 and so on...
```

size*cores

load up the workers



there's more

timers

links

supervision groups

pub/sub

conditions

didn't your proposal
mention JRuby?

so what's wrong with
MRI?

well, nothing
but...

GLOBAL INTERPRETER LOCK

we got this far without a mention

MRI

not so bad when you're
I/O bound

MRI

but what about
computation?

JRuby

has no such lock

rubinius too!

that low hanging fruit?

yeah, about that...



but there is one tip!

blocking I/O...

don't



**an event-driven IO system for
building fast, scalable network
applications that integrate
directly with celluloid actors**

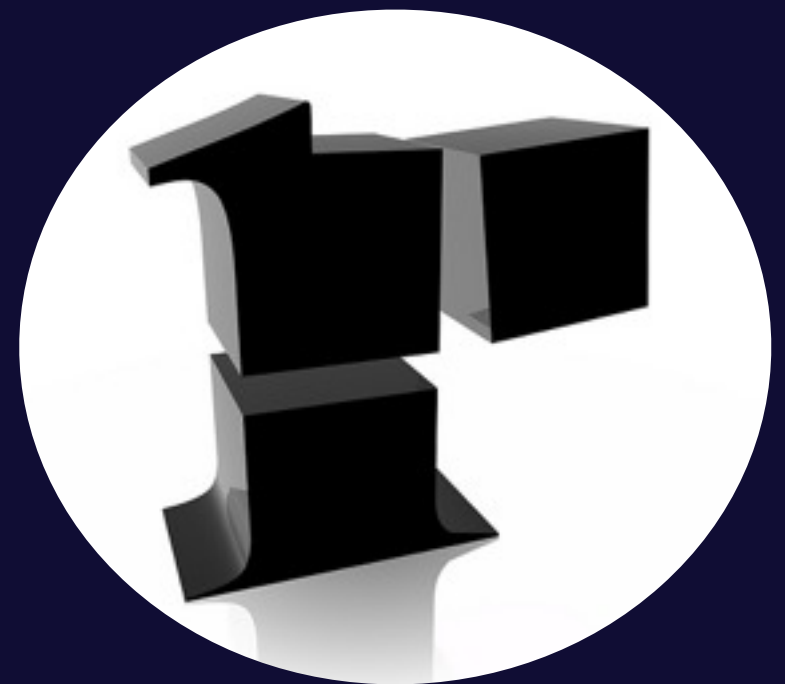
unlike certain other evented
I/O systems which limit you to a
single event loop per process
Celluloid::IO lets you make
as many actors as you want
system resources permitting

the future of ruby

concurrency

and parallelism?

the future of ruby
true thread-level
parallelism is available
right now!



the future of ruby
will **MRI** reconsider
the **GIL**?

the future of ruby

ask Matz!

(title collector)

thanks!
@benlovell

