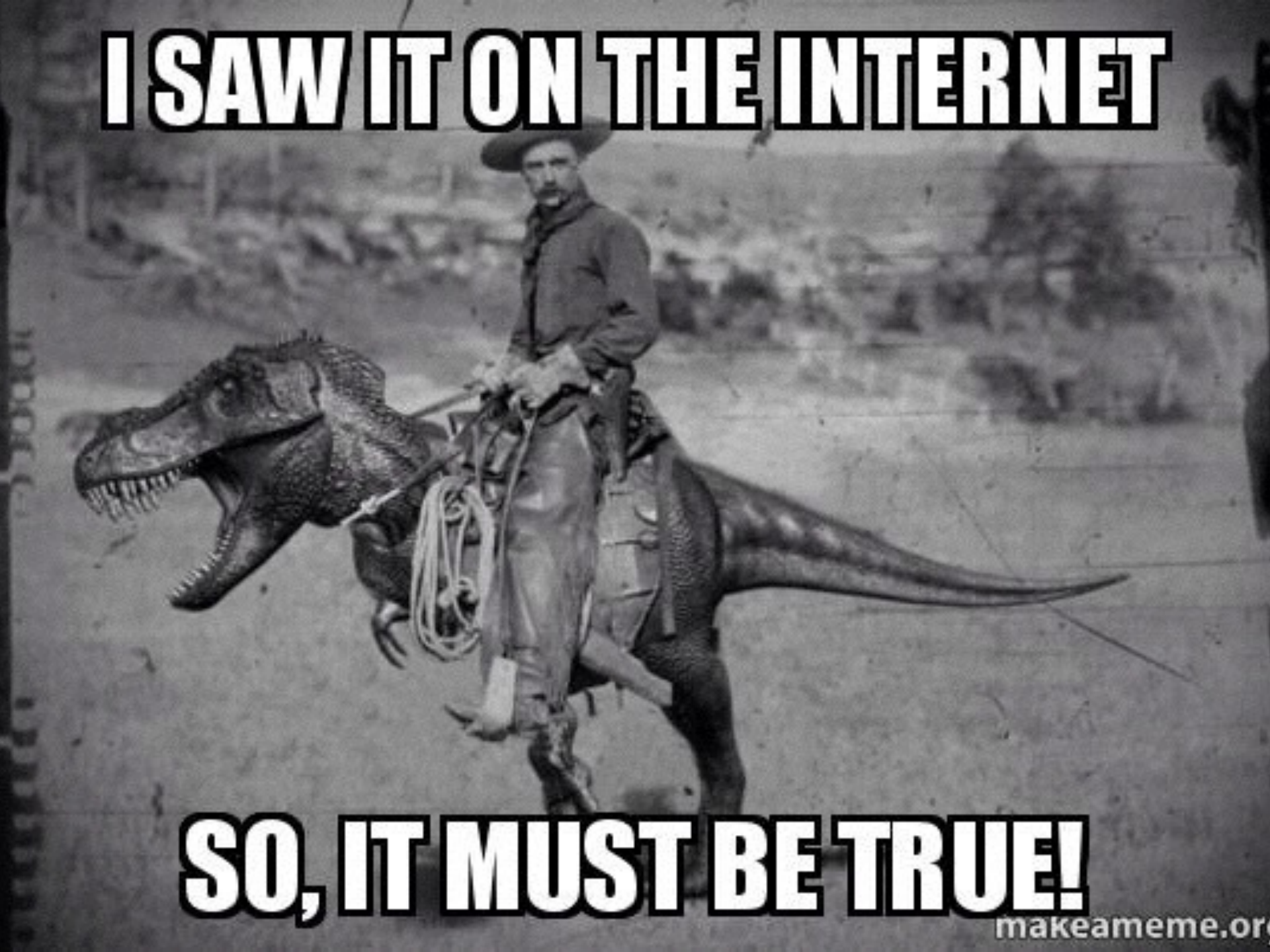
Everything You Know About the GIL is Wrong

Jerry D'Antonio
Software Developer – Test Double
@jerrydantonio

Creator – Concurrent Ruby

- concurrent-ruby.com
- Used by
  - Rails
  - Sidekiq
  - Logstash
  - Dynflow
  - Volt
  - Hamster
  - Pakyow
  - Microsoft Azure

Rumor has it, Ruby can't do concurrency


I SAW IT ON THE INTERNET
SO, IT MUST BE TRUE!
makeameme.org

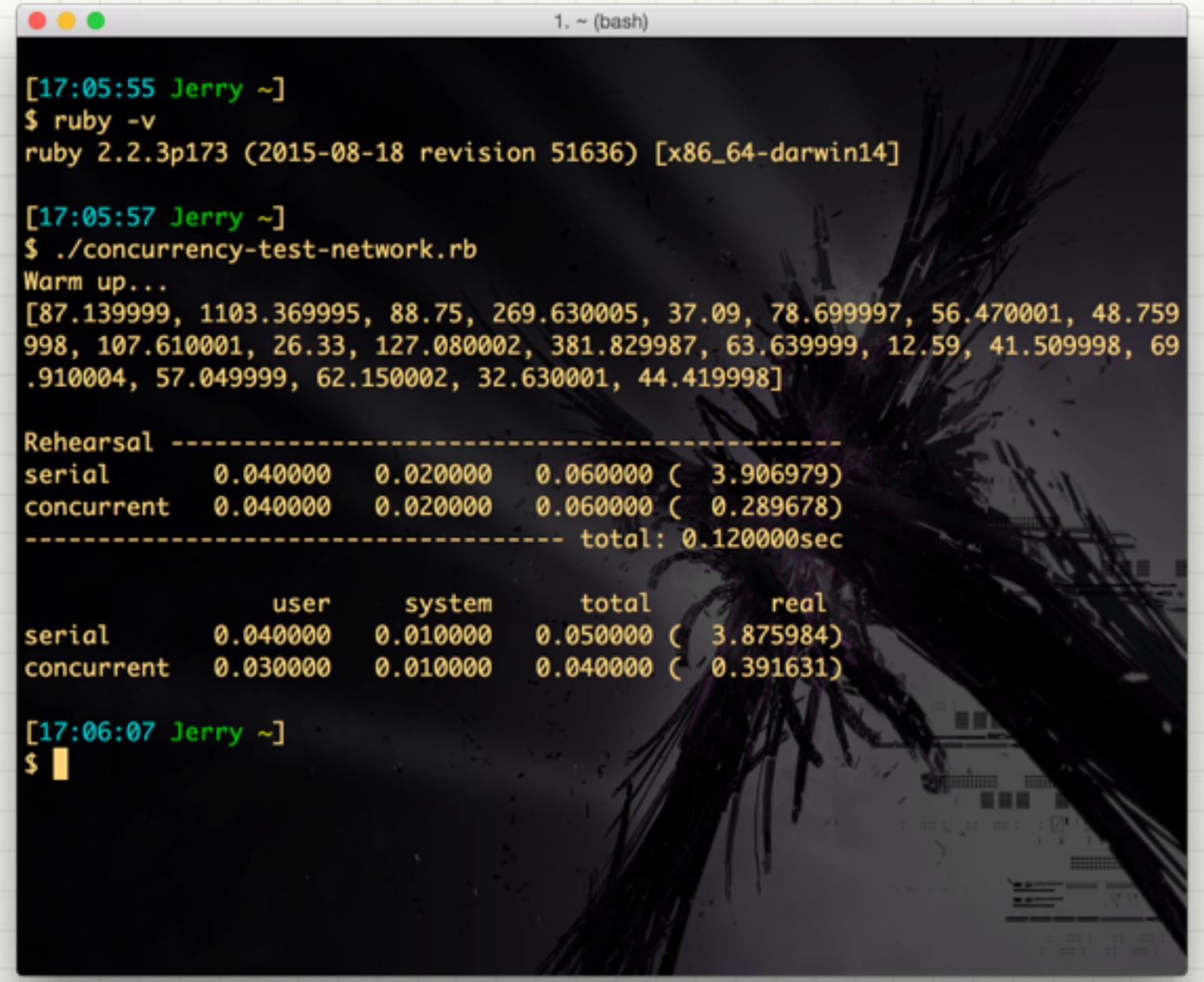
According to the Internet

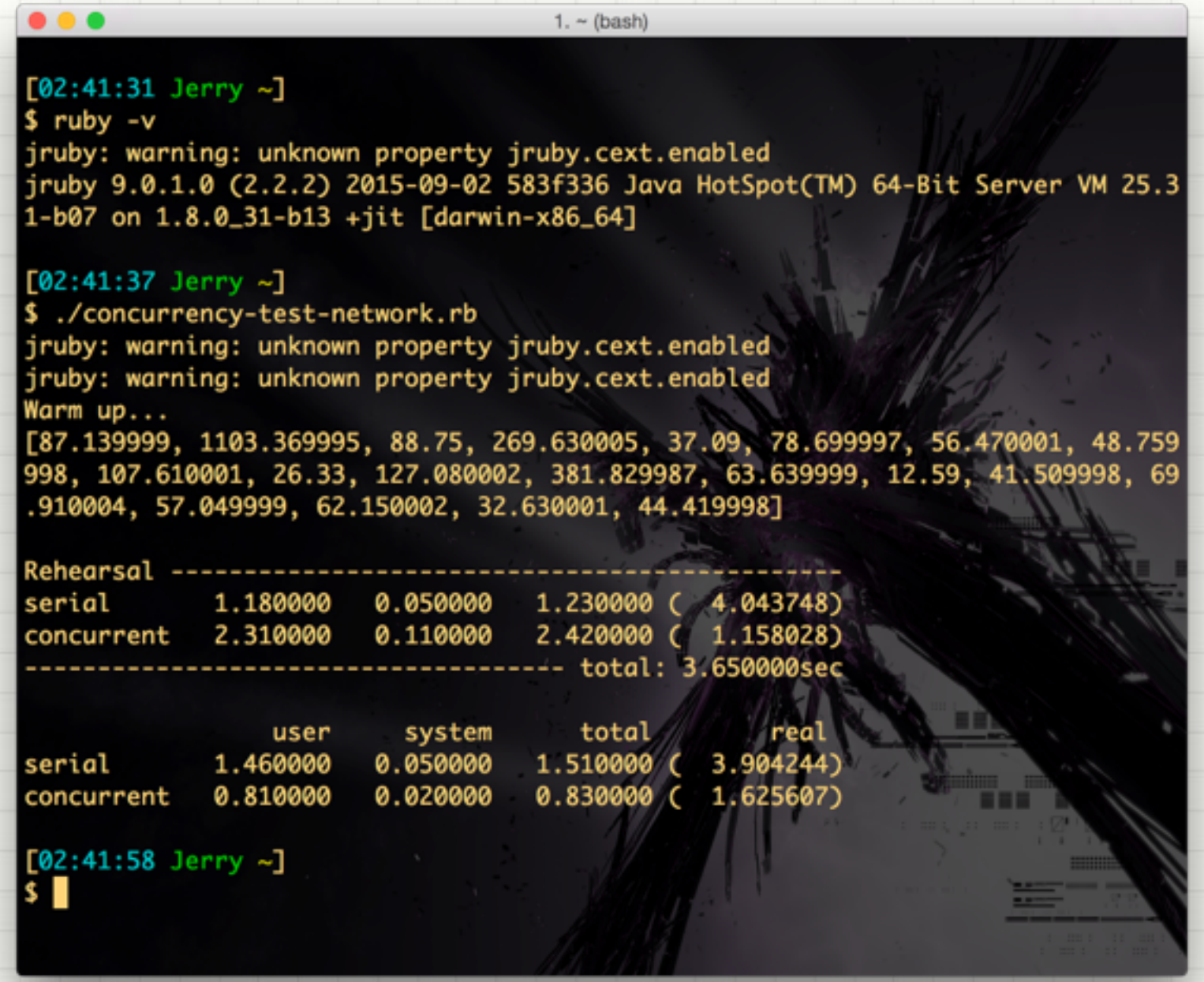
- Ruby has a Global Interpreter Lock (GIL), aka Global Virtual Machine Lock (GVL)
- The GIL is soulless, heartless, and pure evil
- The GIL hates you and wants you to be miserable
- The GIL eats babies for breakfast, kittens for desert, and puppies for a midnight snack
- The GIL is the sole cause of climate change
- If there was no GIL there would be no war

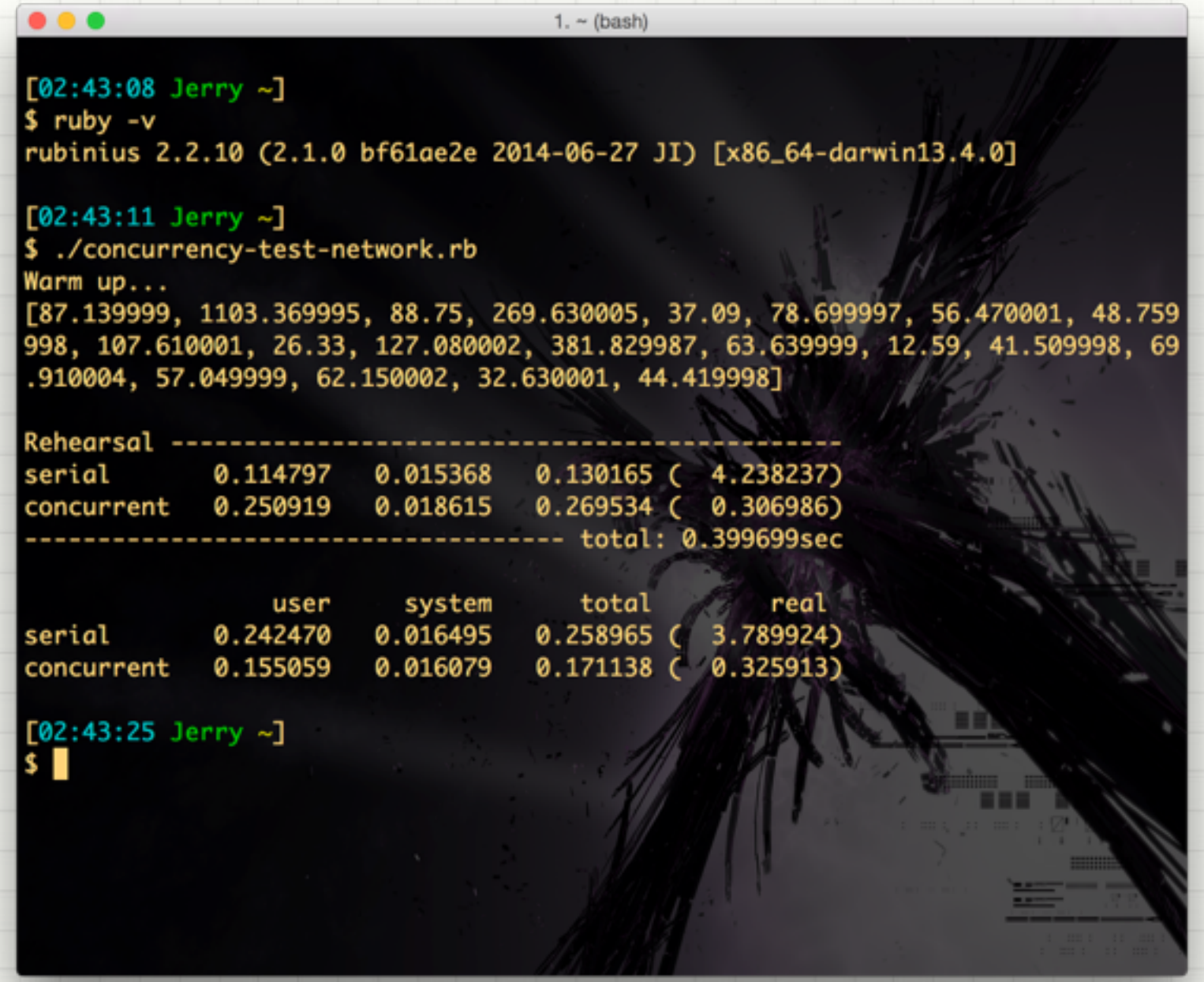
Let’s look at some code

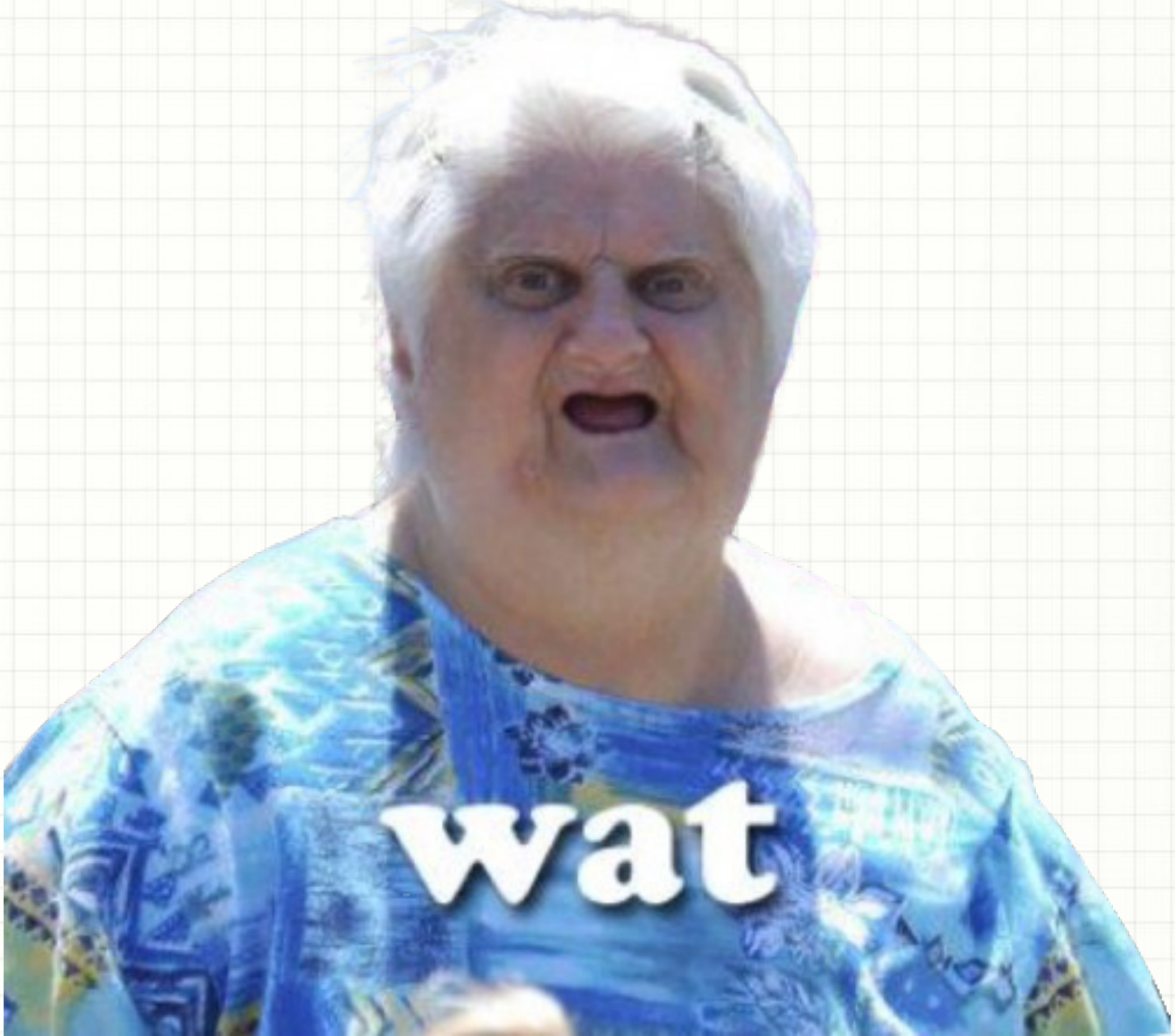
```ruby
#!/usr/bin/env ruby

require 'concurrent'
require 'benchmark'
require 'open-uri' # for open(uri)

# http://www.bloomberg.com/visual-data/best-and-worst/highest-risk-adjusted-returns-in-us-tech-companies
SYMBOLS = ['MA', 'PCLN', 'ADP', 'V', 'TSS', 'FISV', 'EBAY', 'PAYX', 'WDC', 'SYMC',
           'AAPL', 'AMZN', 'KLAC', 'FNFV', 'XLNX', 'MSI', 'ADI', 'VRSN', 'CA', 'YHOO']
YEAR = 2014

def get_year_end_closing(symbol, year)
  uri = "http://ichart.finance.yahoo.com/table.csv?s=#{symbol}&a=01&b=04&c=#{year}&d=01&e=14&f=#{year+1}&g
  data = open(uri) {|f| f.collect{|line| line.strip } }
  data[1].split(',')[4].to_f
end

def serial_get_year_end_closings(year)
  SYMBOLS.collect do |symbol|
    get_year_end_closing(symbol, year)
  end
end

def concurrent_get_year_end_closings(year)
  futures = SYMBOLS.collect do |symbol|
    Concurrent::Future.execute { get_year_end_closing(symbol, year) }
  end
  futures.collect { |future| future.value }
end
```

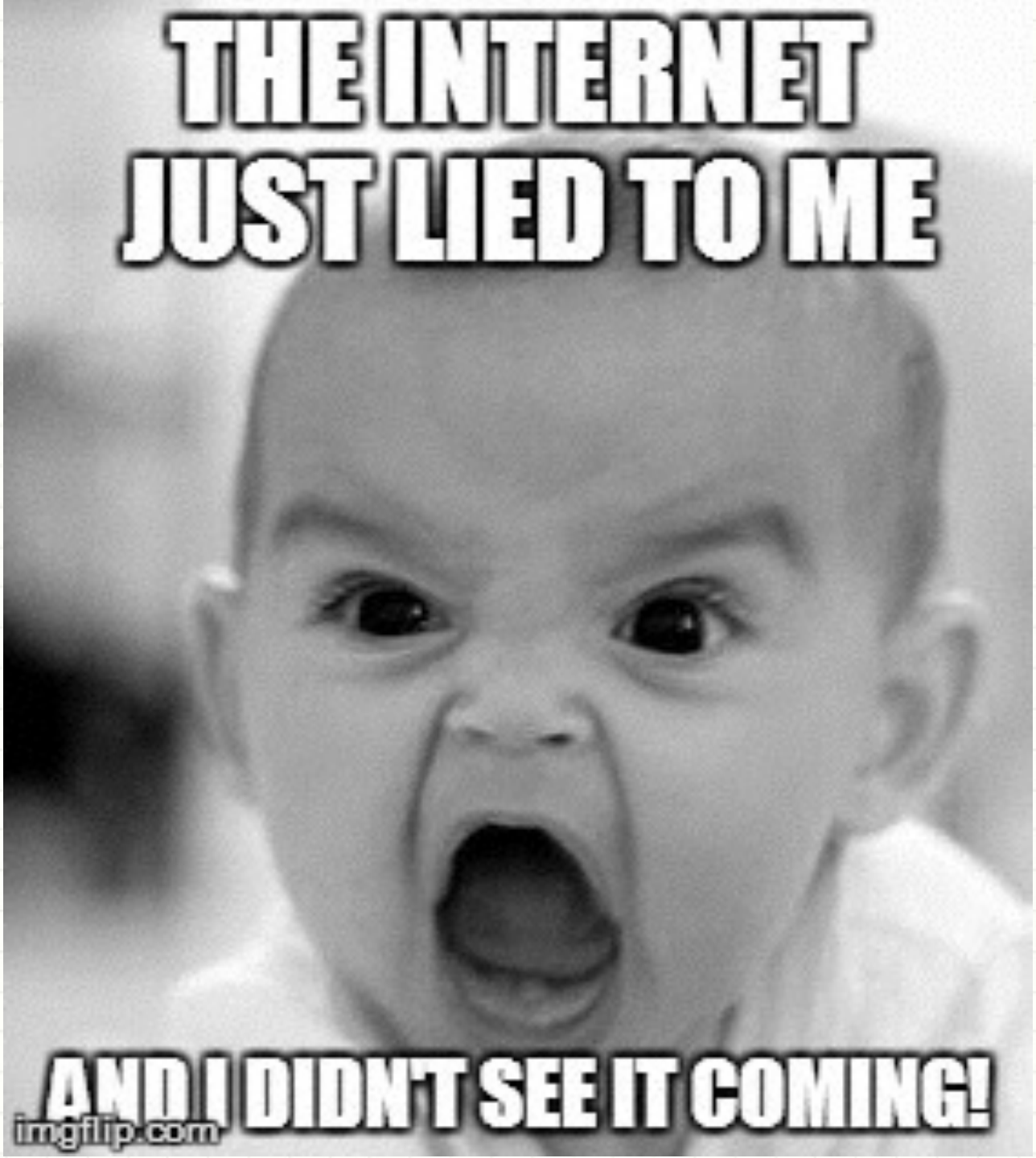
```ruby
# make sure there's no unfair I/O caching going on
# and also warm up the global thread pool
puts 'Warm up...'
p concurrent_get_year_end_closings(YEAR)
puts "\n"

Benchmark.bmbm do |bm|
  bm.report('serial') do
    serial_get_year_end_closings(YEAR)
  end

  bm.report('concurrent') do
    concurrent_get_year_end_closings(YEAR)
  end
end
```


1. ~ (bash)
[17:05:55 Jerry ~]
$ ruby -v
ruby 2.2.3p173 (2015-08-18 revision 51636) [x86_64-darwin14]
[17:05:57 Jerry ~]
$ ./concurrency-test-network.rb
Warm up...
[87.139999, 1103.369995, 88.75, 269.630005, 37.09, 78.699997, 56.470001, 48.759
998, 107.610001, 26.33, 127.080002, 381.829987, 63.639999, 12.59, 41.509998, 69
.910004, 57.049999, 62.150002, 32.630001, 44.419998]
Rehearsal ---
serial 0.040000 0.020000 0.060000 (3.906979)
concurrent 0.040000 0.020000 0.060000 (0.289678)
--------------------------------- total: 0.120000sec
user system total real
serial 0.040000 0.010000 0.050000 (3.875984)
concurrent 0.030000 0.010000 0.040000 (0.391631)
[17:06:07 Jerry ~]
$


1. ~ (bash)
[02:41:31 Jerry ~]
$ ruby -v
jruby: warning: unknown property jruby.cext.enabled
jruby 9.0.1.0 (2.2.2) 2015-09-02 583f336 Java HotSpot(TM) 64-Bit Server VM 25.3
1-b07 on 1.8.0_31-b13 +jit [darwin-x86_64]
[02:41:37 Jerry ~]
$ ./concurrency-test-network.rb
jruby: warning: unknown property jruby.cext.enabled
jruby: warning: unknown property jruby.cext.enabled
Warm up...
[87.139999, 1103.369995, 88.75, 269.630005, 37.09, 78.699997, 56.470001, 48.759
998, 107.610001, 26.33, 127.080002, 381.829987, 63.639999, 12.59, 41.509998, 69
.910004, 57.049999, 62.150002, 32.630001, 44.419998]
Rehearsal ---
serial 1.180000 0.050000 1.230000 (4.043748)
concurrent 2.310000 0.110000 2.420000 (1.158028)
--------------------------------- total: 3.650000sec
user system total real
serial 1.460000 0.050000 1.510000 (3.904244)
concurrent 0.810000 0.020000 0.830000 (1.625607)
[02:41:58 Jerry ~]
$


1. ~ (bash)
[02:43:08 Jerry ~]
$ ruby -v
rubinius 2.2.10 (2.1.0 bf61ae2e 2014-06-27 JI) [x86_64-darwin13.4.0]
[02:43:11 Jerry ~]
$ ./concurrency-test-network.rb
Warm up...
[87.139999, 1103.369995, 88.75, 269.630005, 37.09, 78.699997, 56.470001, 48.759998, 107.610001, 26.33, 127.080002, 381.829987, 63.639999, 12.59, 41.509998, 69.910004, 57.049999, 62.150002, 32.630001, 44.419998]
Rehearsal ---
serial 0.114797 0.015368 0.130165 (4.238237)
concurrent 0.250919 0.018615 0.269534 (0.306986)
---------------------------------- total: 0.399699sec
user system total real
serial 0.242470 0.016495 0.258965 (3.789924)
concurrent 0.155059 0.016079 0.171138 (0.325913)
[02:43:25 Jerry ~]
$


wat


THE INTERNET
JUST LIED TO ME
AND I DIDN'T SEE IT COMING!
imgflip.com

The Obligatory “Concurrency vs. Parallelism” Talk

TL;DR Concurrency is NOT Parallelism

Channeling my inner Rob Pike

- Concurrency: Programming as the *composition* of independently executing processes.

- Parallelism: Programming as the *simultaneous* execution of (possibly related) computations.

Rob Pike
"Concurrency is not Parallelism"
https://talks.golang.org/2012/waza.slide#1

More Rob Pike

- Concurrency is about *dealing* with lots of things at once.
- Parallelism is about *doing* lots of things at once.

Rob Pike
"Concurrency is not Parallelism"
https://talks.golang.org/2012/waza.slide#8

Concurrency vs. Parallelism

- Parallelism *requires* two processor cores
  - No matter the language/runtime
  - A processor core can only execute one instruction at a time
- Concurrency can happen when there is only one core
  - Concurrency is about design
  - Improved performance is a side effect

Non-concurrent programs gain no benefit from running on multiple processors

Concurrent programs get parallelism for free when the runtime supports it

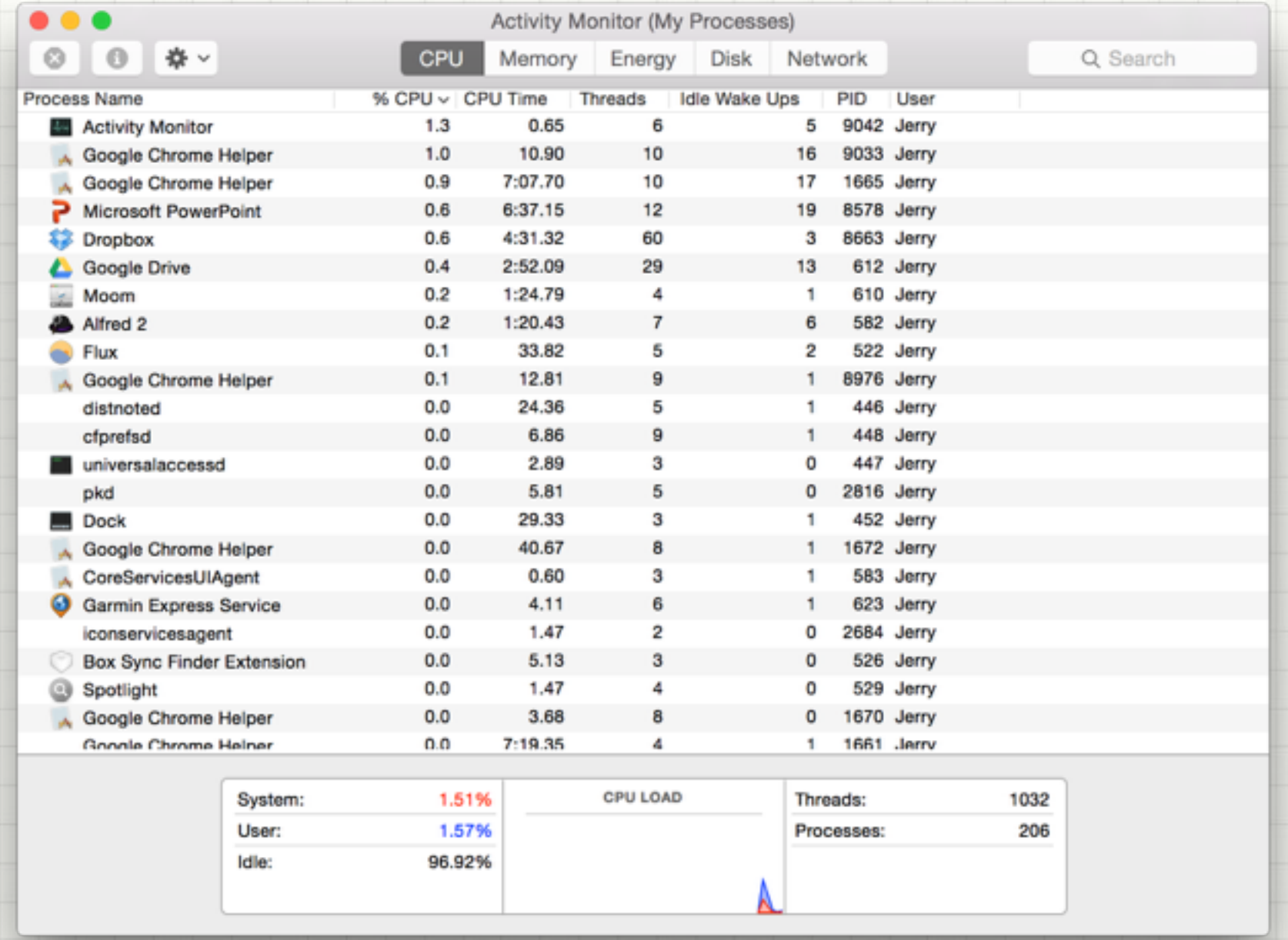
Let’s Talk About the GIL

The “L” in GIL means “lock”

- A “lock” in computer science is a synchronization mechanism which ensures that only one thread can access a protected resource at any given time
- A thread needing to access a protected resource must request the lock
  - If the lock is available it is acquired
  - If not, the thread blocks and waits for the lock to become available

What is a thread?

- A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler – Wikipedia
- Most modern operating systems use threads for enabling concurrent tasks within individual programs
- The number of threads running at any given time, across all programs, vastly exceeds the number of processor cores on the syste


Activity Monitor (My Processes)
CPU
Memory
Energy
Disk
Network
Search
Process Name % CPU CPU Time Threads Idle Wake Ups PID User
Activity Monitor 1.3 0.65 6 5 9042 Jerry
Google Chrome Helper 1.0 10.90 10 16 9033 Jerry
Google Chrome Helper 0.9 7:07.70 10 17 1665 Jerry
Microsoft PowerPoint 0.6 6:37.15 12 19 8578 Jerry
Dropbox 0.6 4:31.32 60 3 8663 Jerry
Google Drive 0.4 2:52.09 29 13 612 Jerry
Moom 0.2 1:24.79 4 1 610 Jerry
Alfred 2 0.2 1:20.43 7 6 582 Jerry
Flux 0.1 33.82 5 2 522 Jerry
Google Chrome Helper 0.1 12.81 9 1 8976 Jerry
distnoted 0.0 24.36 5 1 446 Jerry
cfprefsd 0.0 6.86 9 1 448 Jerry
universalaccessd 0.0 2.89 3 0 447 Jerry
pkd 0.0 5.81 5 0 2816 Jerry
Dock 0.0 29.33 3 1 452 Jerry
Google Chrome Helper 0.0 40.67 8 1 1672 Jerry
CoreServicesUIAgent 0.0 0.60 3 1 583 Jerry
Garmin Express Service 0.0 4.11 6 1 623 Jerry
iconservicesagent 0.0 1.47 2 0 2684 Jerry
Box Sync Finder Extension 0.0 5.13 3 0 526 Jerry
Spotlight 0.0 1.47 4 0 529 Jerry
Google Chrome Helper 0.0 3.68 8 0 1670 Jerry
Google Chrome Helper 0.0 7:19.35 4 1 1661 Jerry
System: 1.51%
User: 1.57%
Idle: 96.92%
CPU LOAD
Threads: 1032
Processes: 206

What is a thread?

- Many programming languages (Ruby, Java) map language constructs directly to operating system threads
- Some programming languages (Erlang, Go) place an abstraction over native threads and provide their own scheduler
- Regardless of the language, you still have threads within the operating system

Multithreading in the OS

- The operating system must manage threads of execution across *all* running programs
  - Remember, each core can only run one thread
- When the operating system pauses the execution of one thread to resume execution of another it's called a "context switch"
- No programming language can preempt an operating system context switch
  - Even a single-thread program
  - At best, we can give it suggestions and hints

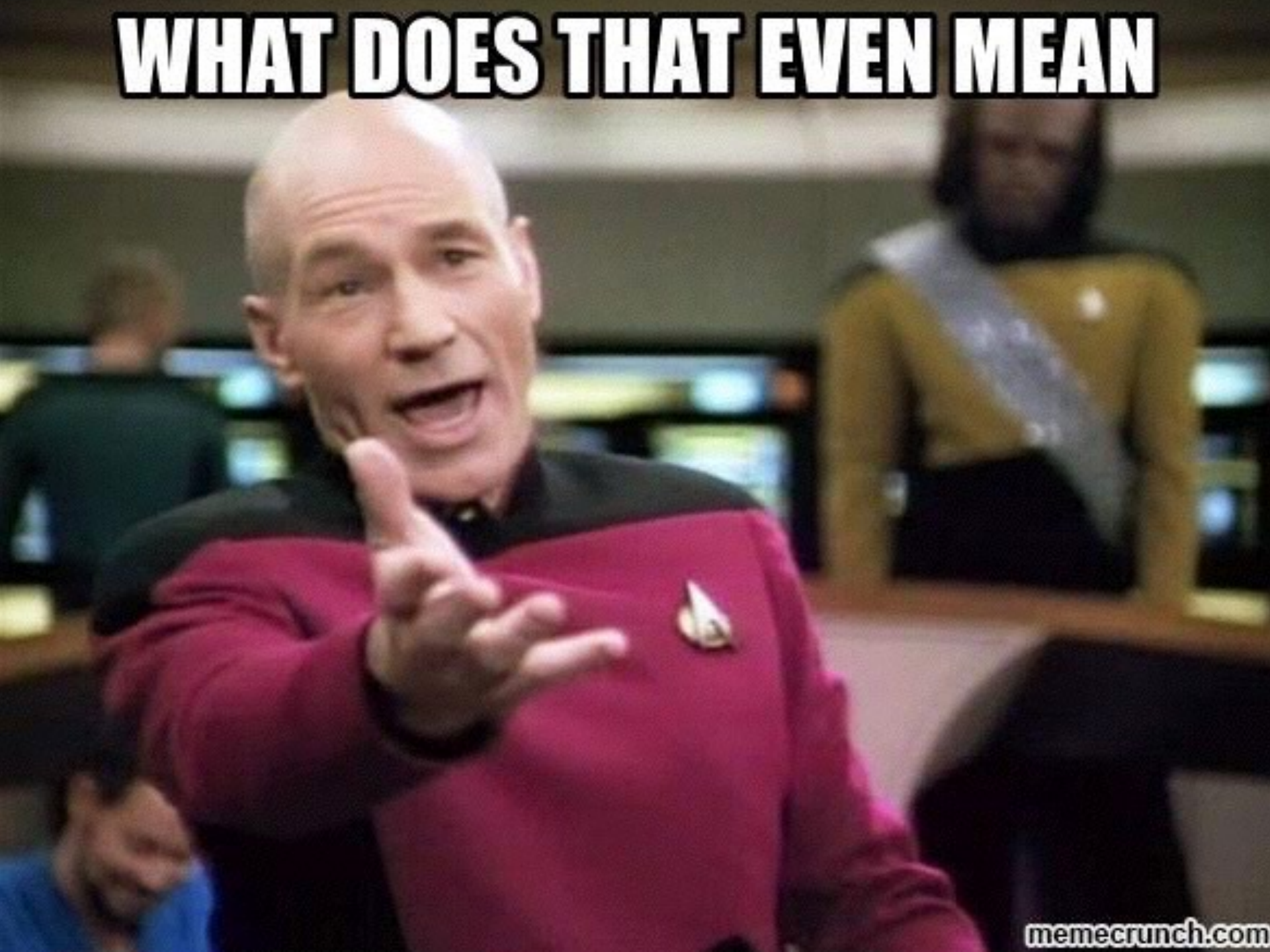
Getting back to the GIL

- Every programming language/runtime must have internal logic to manage itself across operating system context switches and across its own concurrency constructs
- Some languages run one thread per processor and handle context switching internally
- Other languages let the operating system manage all concurrency and context switching

Ruby uses the GIL to protect its internal state across OS context switches

How the GIL works (simplified)

- Thread A begins some work
  - Ruby locks the GIL to protect its internal state
- A context switch occurs and Thread B runs
  - Thread B attempts to acquire the GIL, but fails
  - Thread B signals the OS that it's done for now
- Eventually Thread A is resumed by the OS
  - When done, Thread A releases the GIL
- On the next context switch Thread B can acquire the GIL


WHAT DOES THAT EVEN MEAN
memecrunch.com

Implications of the GIL

- Only one “unit” of Ruby code can execute at any given time
  - Although there may be multiple threads and multiple processors, executing code will regularly be blocked by the GIL
- When given multiple cores Ruby is unable to experience true parallelism
- The Ruby runtime guarantees that *it* will always be in a consistent internal state
  - But it makes no guarantees about *your* code

The obligatory word definition

guar·an·tee
/ˌgerənˈtē/

noun

1. a formal promise or assurance (typically in writing) that certain conditions will be fulfilled, especially that a product will be repaired or replaced if not of a specified quality and durability.

verb

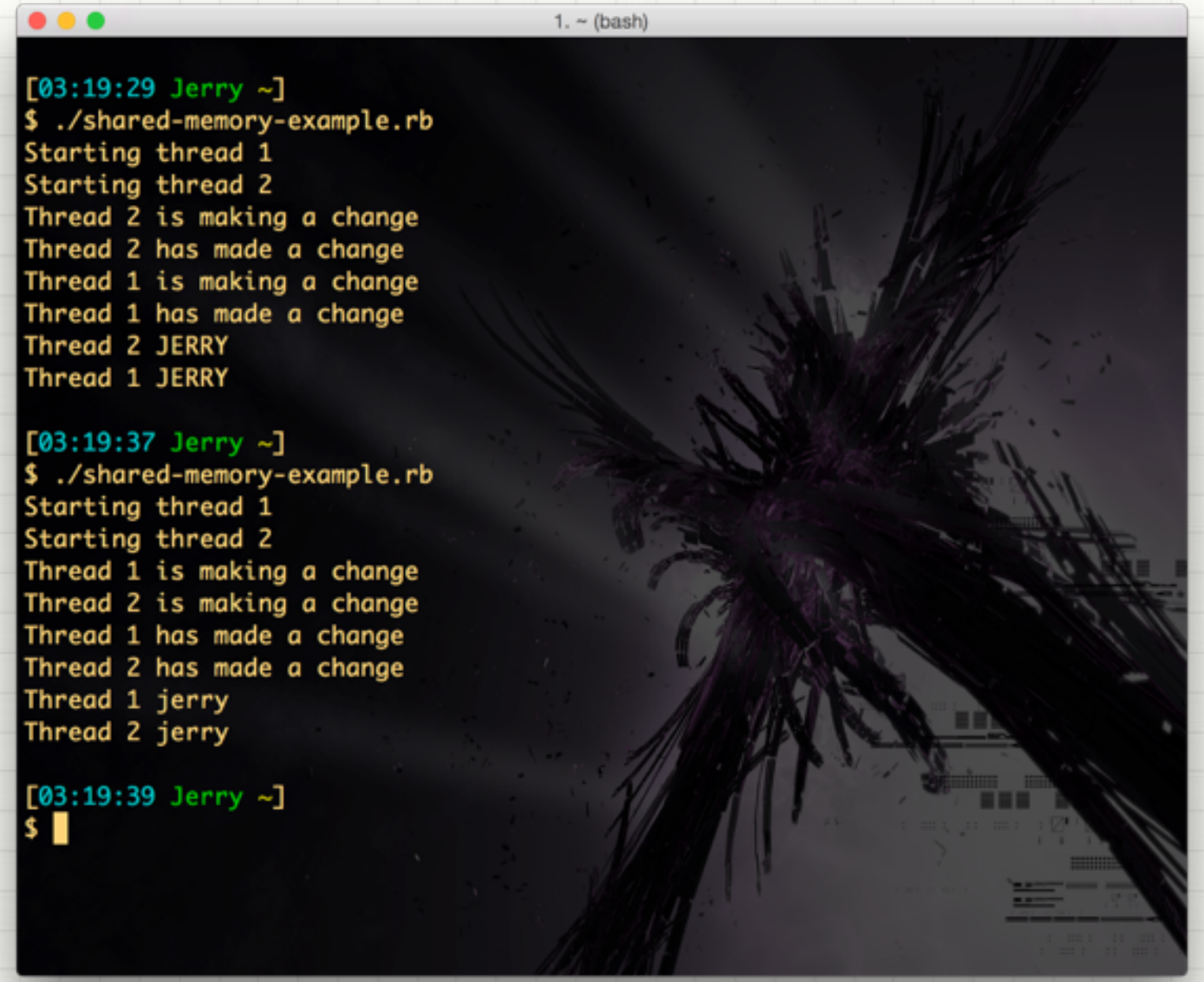
1. provide a formal assurance or promise, especially that certain conditions shall be fulfilled relating to a product, service, or transaction.

Google

Share and share alike

- Ruby is a “shared memory” language
- All variables are references
  - The variable itself is simply a memory address
  - The data is stored at the references memory location
- Two variables may reference the same memory address
- Two threads may share data by simultaneously accessing the same memory location

```ruby
#!/usr/bin/env ruby

str = "Jerry"

t1 = Thread.new(str) do |s|
  print "Starting thread 1\n"
  sleep rand(2)
  print "Thread 1 is making a change\n"
  s.upcase!
  print "Thread 1 has made a change\n"
  sleep rand(2)
  print "Thread 1 #{s}\n"
end

t2 = Thread.new(str) do |s|
  print "Starting thread 2\n"
  sleep rand(2)
  print "Thread 2 is making a change\n"
  s.downcase!
  print "Thread 2 has made a change\n"
  sleep rand(2)
  print "Thread 2 #{s}\n"
end

[t1, t2].each(&:join)
```

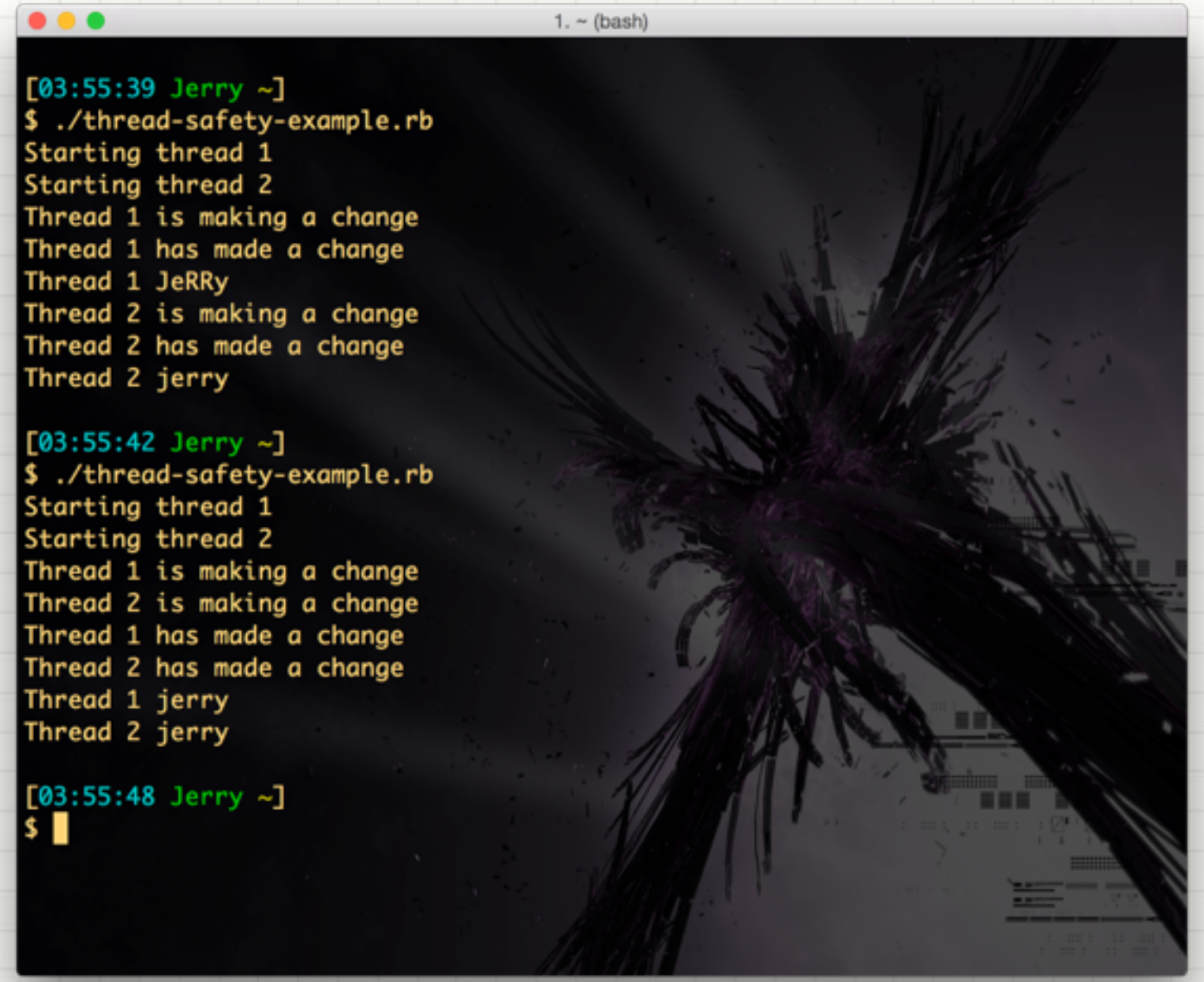

1. ~ (bash)
[03:19:29 Jerry ~]
$ ./shared-memory-example.rb
Starting thread 1
Starting thread 2
Thread 2 is making a change
Thread 2 has made a change
Thread 1 is making a change
Thread 1 has made a change
Thread 2 JERRY
Thread 1 JERRY
[03:19:37 Jerry ~]
$ ./shared-memory-example.rb
Starting thread 1
Starting thread 2
Thread 1 is making a change
Thread 2 is making a change
Thread 1 has made a change
Thread 2 has made a change
Thread 1 jerry
Thread 2 jerry
[03:19:39 Jerry ~]
$

Was that code thread safe?
Was it correct?

Correct vs. safe

- In a fully parallel, shared memory system it is possible for two or more threads to simultaneously access the same memory
- In a concurrent, shared memory system it is possible for a context switch to occur while one thread is in the process of performing complex memory altering operations
- The ordering of these operations is important

```ruby
#!/usr/bin/env ruby

str = "Jerry"

t1 = Thread.new(str) do |s|
  print "Starting thread 1\n"
  s1 = s.dup
  sleep rand(2)
  s1.gsub!(/r/, 'R')
  sleep rand(2)
  print "Thread 1 is making a change\n"
  s.gsub!(/^.*$/, s1)
  print "Thread 1 has made a change\n"
  sleep rand(2)
  print "Thread 1 #{s}\n"
end

t2 = Thread.new(str) do |s|
  print "Starting thread 2\n"
  s2 = s.dup
  sleep rand(2)
  s2.gsub!(/J/, 'j')
  sleep rand(2)
  print "Thread 2 is making a change\n"
  s.gsub!(/^.*$/, s2)
  print "Thread 2 has made a change\n"
  sleep rand(2)
  print "Thread 2 #{s}\n"
end

[t1, t2].each(&:join)
```


1. ~ (bash)
[03:55:39 Jerry ~]
$ ./thread-safety-example.rb
Starting thread 1
Starting thread 2
Thread 1 is making a change
Thread 1 has made a change
Thread 1 JeRRy
Thread 2 is making a change
Thread 2 has made a change
Thread 2 jerry
[03:55:42 Jerry ~]
$ ./thread-safety-example.rb
Starting thread 1
Starting thread 2
Thread 1 is making a change
Thread 2 is making a change
Thread 1 has made a change
Thread 2 has made a change
Thread 1 jerry
Thread 2 jerry
[03:55:48 Jerry ~]
$

Ruby is selfish

- Ruby is an interpreted language
  - Ruby is compiled to bytecode within the interpreter
  - Ruby is free to *optimize* and *reorder* your code
- Every Ruby operation is implemented in C
- The Ruby runtime is just another program; it is under the control of the compiler and the operating system
  - The C compiler is free to *optimize* and *reorder* instructions during compilation
  - An operating system context switch can occur at *any point* in the running C code
- The GIL protects *Ruby*, not your code

Ruby is thread safe, your code isn't

- Every individual read and write to memory is guaranteed to be thread-safe in Ruby
  - The GIL prevents interleaved access to memory used by the runtime
  - The GIL prevents interleaved access to individual variables
  - Ruby itself will never become corrupt
- Ruby makes *no* guarantees about your code

Memory model

- “In computing, a memory model describes the interactions of threads through memory and their shared use of the data.” Wikipedia
- Defines visibility, volatility, atomicity, and synchronization barriers
- Java’s current memory model was adopted in 2004 as part of Java 5
- The C and C++ memory models were adopted in 2011 with C11 and C++11

Ruby does NOT have a documented memory model

The GIL provides an implied memory model but no guarantees

But what about this?

```
Rehearsal ------------------------------------------
serial       0.040000   0.020000   0.060000 (  3.906979)
concurrent   0.040000   0.020000   0.060000 (  0.289678)
--------------------------------- total: 0.120000sec

                 user     system      total        real
serial       0.040000   0.010000   0.050000 (  3.875984)
concurrent   0.030000   0.010000   0.040000 (  0.391631)
```

When your program does a lot by doing nothing at all

I/O, I/O, it's off to work I go

- Modern computers support both blocking and asynchronous (non-blocking) I/O
  - Blocking: the process must wait for I/O to complete before it continues
  - Asynchronous: request an I/O operation from the OS then do something else while I/O is in progress
- I/O in Ruby programs is *blocking*
- I/O within Ruby is *asynchronous*

All Ruby I/O calls unlock the GIL, as do backtick and `system` calls

When Ruby thread is waiting on I/O it does not block other threads

You can’t spell GIL without I/O

- The GIL exists to maintain the internal consistency of the Ruby runtime
- I/O operations are slow, which is why asynchronous I/O was invented
- While I/O is in progress the Ruby thread is blocked so it *cannot* change the internal state
- So Ruby allows other threads to do useful work

Ruby programs which perform significant I/O generally benefit from concurrency

You may be concurrent if…

- Does your program do these things?
  - Read/write from files (such as logs)
  - Interact with databases
  - Listen for inbound network requests (say, HTTP)
  - Connect to external HTTP APIs
  - Send email
- If so, then your program may benefit from concurrency

WHERE'S THE LOVE?

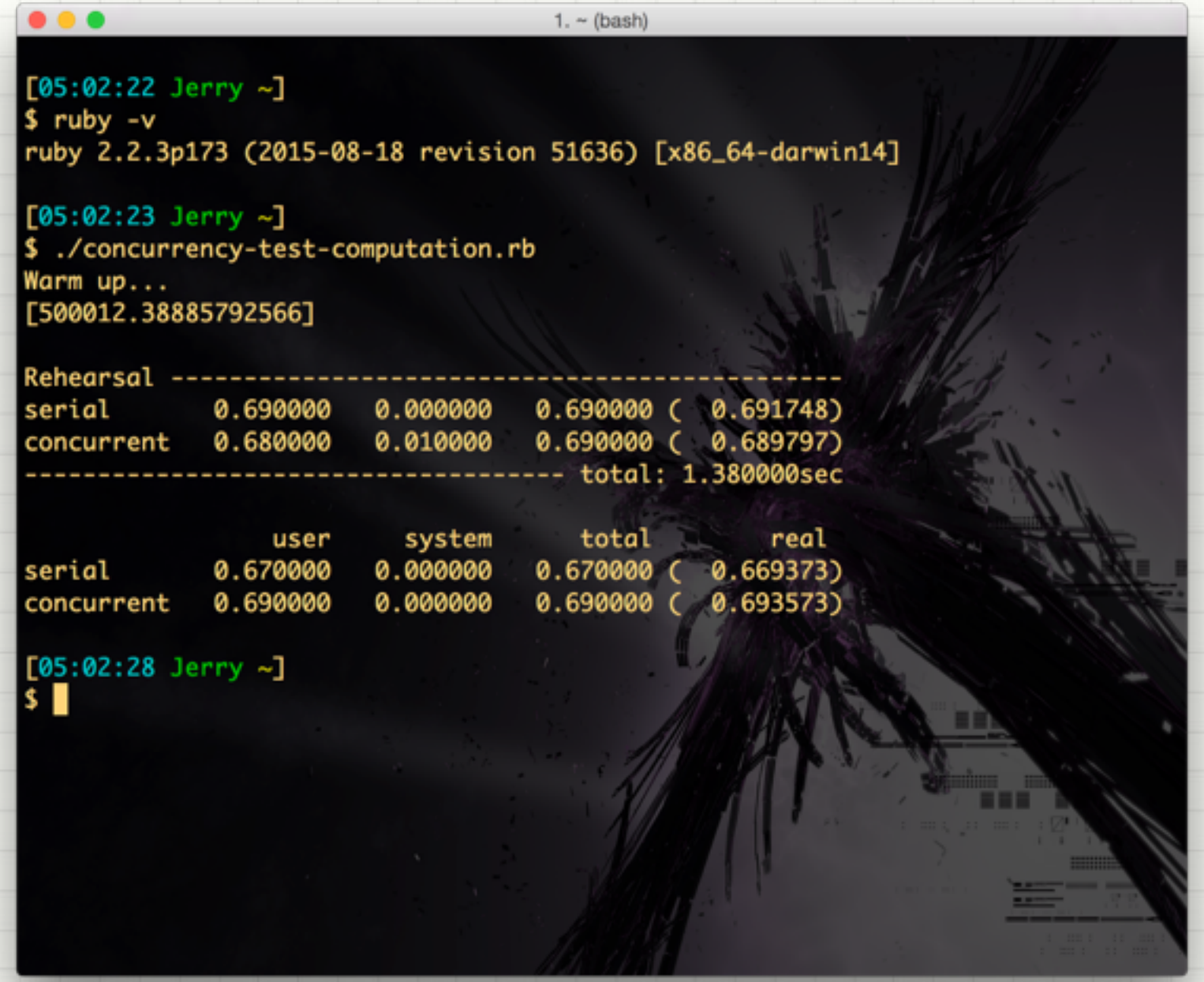
Lack of knowledge

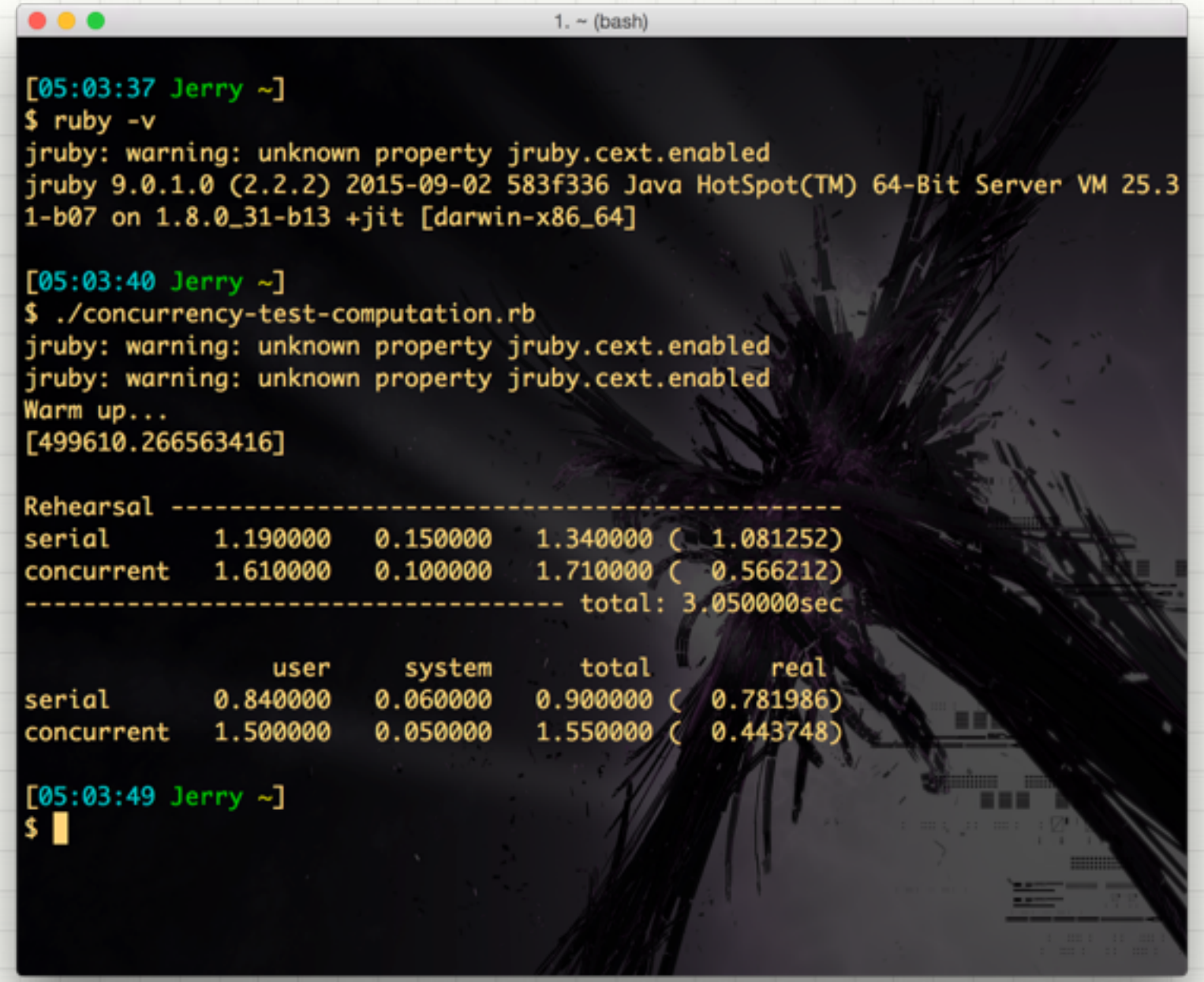
- Most Ruby programmers never write concurrent code
- Often concurrency is managed by the frameworks we use (Rails)
- Many of the domains requiring highly concurrent code also need high performance so they are written in other languages
- So learning about concurrency simply isn’t necessary for many in the Ruby community

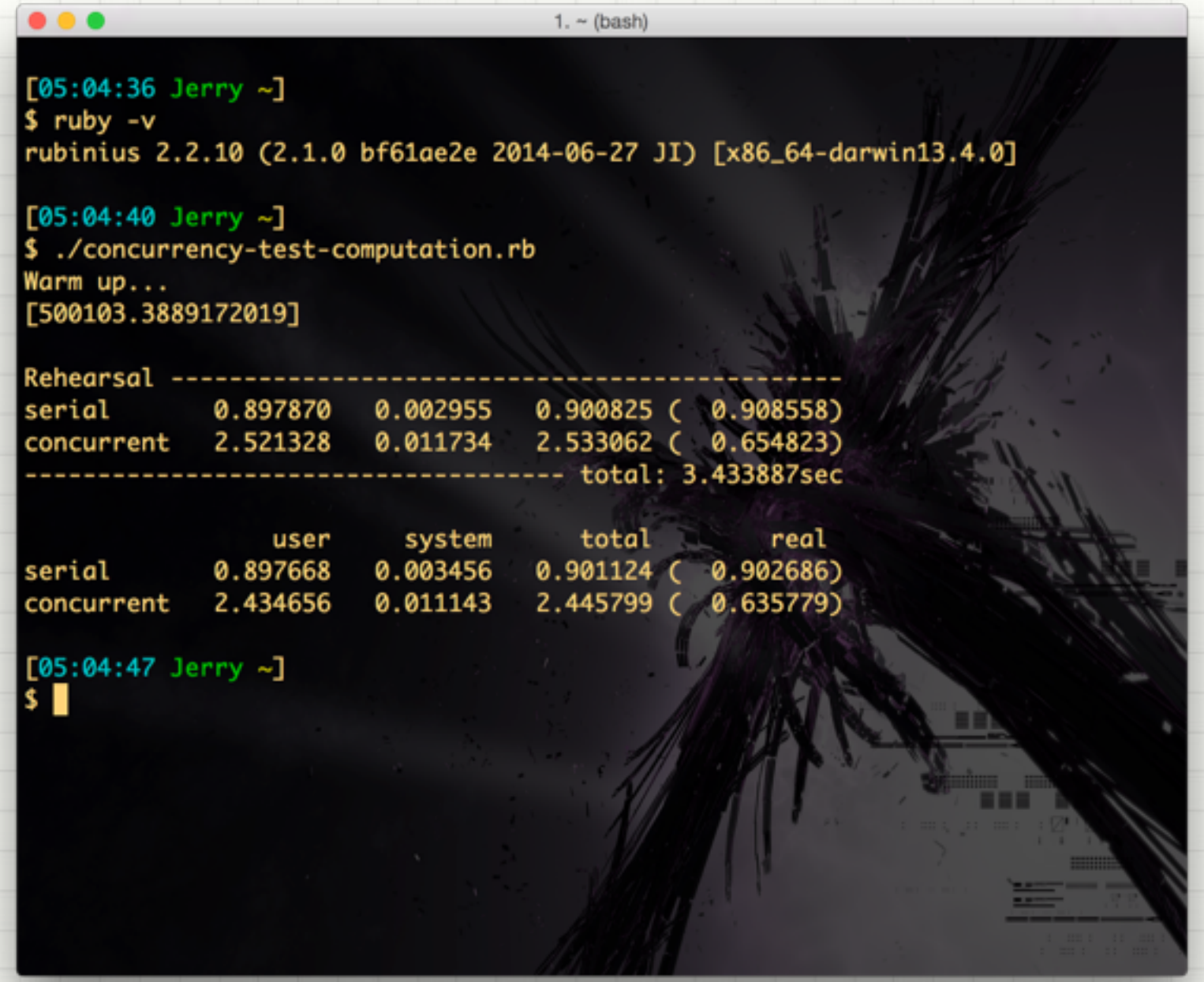
Concurrency in Ruby isn't perfect

- Ruby is good at concurrent I/O; not so much for processor intensive operations
- The GIL prevents full parallelism
  - The operating system will still multiplex across multiple Ruby threads
  - But many context switches will result in a no-op because the GIL is locked
- Some programs will gain no performance benefit from concurrency

```ruby
#!/usr/bin/env ruby

require 'concurrent'
require 'benchmark'

COUNT = 10
NUMBERS = 1_000_000.times.collect { rand }

# obviously contrived example
def sum(numbers)
  total = 0
  numbers.size.times {|i| total += numbers[i]}
  total
end

def serial_sum(numbers, count)
  count.times { sum(numbers) }
end

def concurrent_sum(numbers, count)
  futures = count.times.collect do
    Concurrent::Future.execute { sum(numbers) }
  end
  futures.collect { |future| future.value }
end
```


1. ~ (bash)
[05:02:22 Jerry ~]
$ ruby -v
ruby 2.2.3p173 (2015-08-18 revision 51636) [x86_64-darwin14]
[05:02:23 Jerry ~]
$ ./concurrency-test-computation.rb
Warm up...
[500012.38885792566]
Rehearsal --
serial 0.690000 0.000000 0.690000 (0.691748)
concurrent 0.680000 0.010000 0.690000 (0.689797)
------------------------------------ total: 1.380000sec
user system total real
serial 0.670000 0.000000 0.670000 (0.669373)
concurrent 0.690000 0.000000 0.690000 (0.693573)
[05:02:28 Jerry ~]
$


1. ~ (bash)
[05:03:37 Jerry ~]
$ ruby -v
jruby: warning: unknown property jruby.cext.enabled
jruby 9.0.1.0 (2.2.2) 2015-09-02 583f336 Java HotSpot(TM) 64-Bit Server VM 25.3
1-b07 on 1.8.0_31-b13 +jit [darwin-x86_64]
[05:03:40 Jerry ~]
$ ./concurrency-test-computation.rb
jruby: warning: unknown property jruby.cext.enabled
jruby: warning: unknown property jruby.cext.enabled
Warm up...
[499610.266563416]
Rehearsal ---
serial 1.190000 0.150000 1.340000 (1.081252)
concurrent 1.610000 0.100000 1.710000 (0.566212)
--------------------------------- total: 3.050000sec
user system total real
serial 0.840000 0.060000 0.900000 (0.781986)
concurrent 1.500000 0.050000 1.550000 (0.443748)
[05:03:49 Jerry ~]
$


1. ~ (bash)
[05:04:36 Jerry ~]
$ ruby -v
rubinius 2.2.10 (2.1.0 bf61ae2e 2014-06-27 JI) [x86_64-darwin13.4.0]
[05:04:40 Jerry ~]
$ ./concurrency-test-computation.rb
Warm up...
[500103.3889172019]
Rehearsal --
serial 0.897870 0.002955 0.900825 (0.908558)
concurrent 2.521328 0.011734 2.533062 (0.654823)
------------------------------------ total: 3.433887sec
user system total real
serial 0.897668 0.003456 0.901124 (0.902686)
concurrent 2.434656 0.011143 2.445799 (0.635779)
[05:04:47 Jerry ~]
$

Tools of the trade

- Ruby's concurrency tools are pretty basic: Thread, Fiber, Mutex, ConditionVariable
- Java: *java.util.concurrent*
- Go: Goroutine, channel, ticker, timer, mutex, atomic variables…
- Clojure: future, promise, delay, ref, atom, agent, core-async…
- Erlang: spawn, gen_server, gen_event…
- Scala: Executor, future, promise, actor (Akka)…

Ruby concurrency needs two things: better tools and a better publicist

Summary

- Concurrency is not parallelism
- The GIL protects Ruby's internal state when the operating system context switches
  - The GIL does not provide thread safety guarantees to user code
  - But it imposes an implicit memory model
- The GIL prevents true parallelism in Ruby
- But Ruby is pretty good at multiplexing threads performing blocking I/O


KEEP
CALM
AND
DON'T SWEAT
THE GIL

Jerry D'Antonio

@jerrydantonio
github.com/jdantonio
concurrent-ruby.com

td

Hire @testdouble
If your team needs some help!