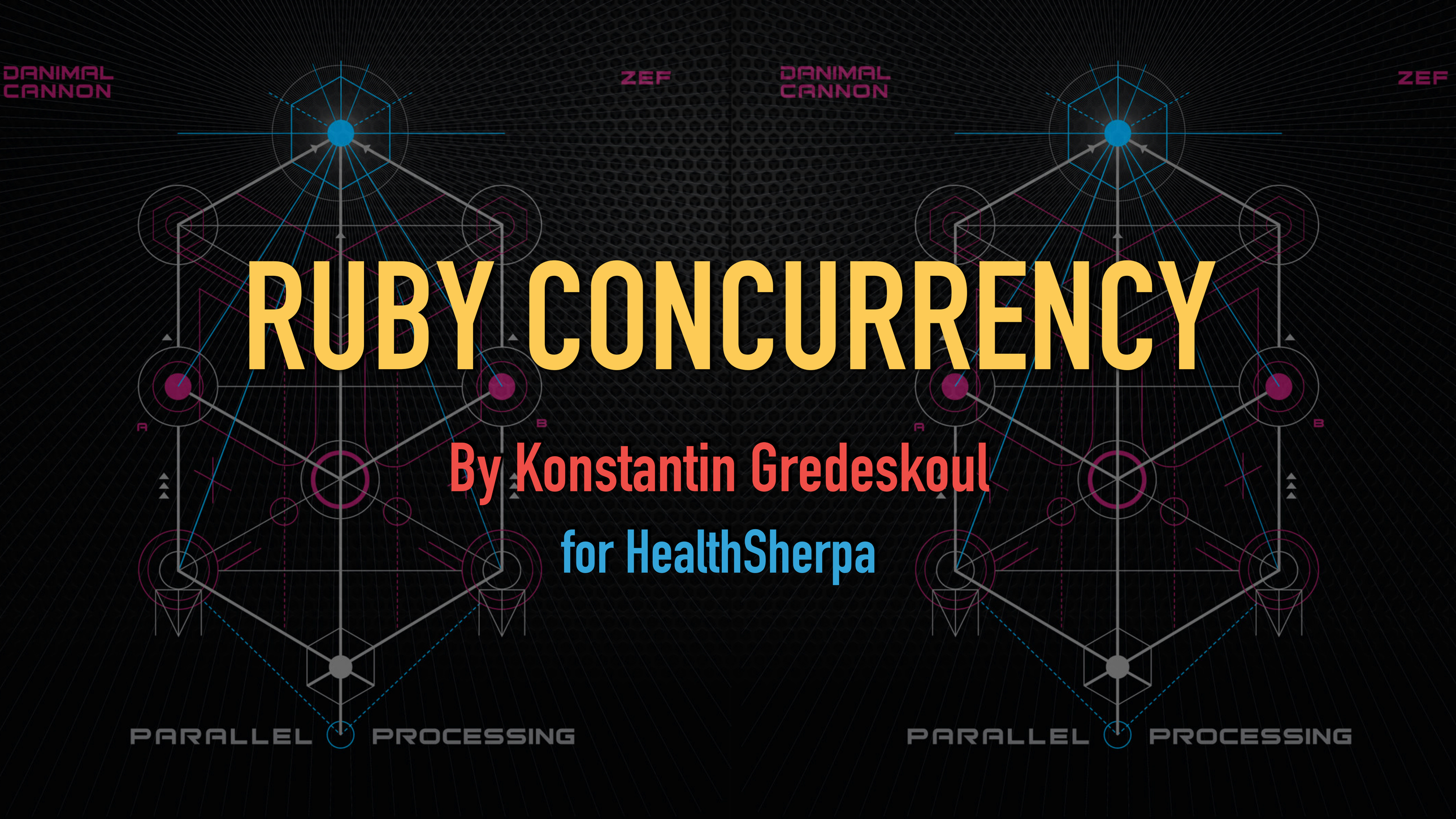


RUBY CONCURRENCY

By Konstantin Gredeskoul

for HealthSherpa

Who am I, and why should you care?

I've written production Ruby Code since 2007.

I've published over 40 Ruby Gems.

They've been downloaded about 20,000,000 times total.

I spoke at RubyConf in 2015.

My presentations have been viewed by over 200,000 people.

I sometimes switch to Ruby in a social conversation.

But mostly, because I love to spread Ruby Luv, and NOT Coronavirus.

LET'S START YOU UP WITH A
SHORT SELF-TEST
OF HOW WELL YOU UNDERSTAND
RUBY'S CONCURRENCY MODEL BEFORE THE TALK.

QUESTION:
TO FULLY UTILIZE ALL AVAILABLE CORES ON A MULTI-CORE SERVER, WE SHOULD...

A) RUN ONE RUBY PROCESS WITH MANY THREADS

B) RUN SEVERAL SINGLE-THREADED RUBY PROCESSES

C) RUN AS MANY RUBY PROCESSES AS THE NUMBER OF CPU CORES AVAILABLE

D) RUN ONE RUBY PROCESS THAT FORKS THOUSANDS OF WORKERS

WHAT SHOULD YOU TAKE AWAY FROM THIS TALK?

1. You'll learn where it is appropriate to use Threads in MRI Ruby applications and when it is not.
2. Likewise, when is it appropriate to use multiple Ruby Processes, how many should you start, and how?
3. How to determine the optimal number of threads per process.
4. List of tools that are available to you in addition to the raw Thread class
5. And finally, how to write thread-safe code and detect unsafe code.

EXAMPLE 1.

CONCURRENT:

TWO QUEUES AND ONE COFFEE MACHINE.

PARALLEL:

TWO QUEUES AND TWO COFFEE MACHINES.

Joe Armstrong – “Erlang and other stuff”

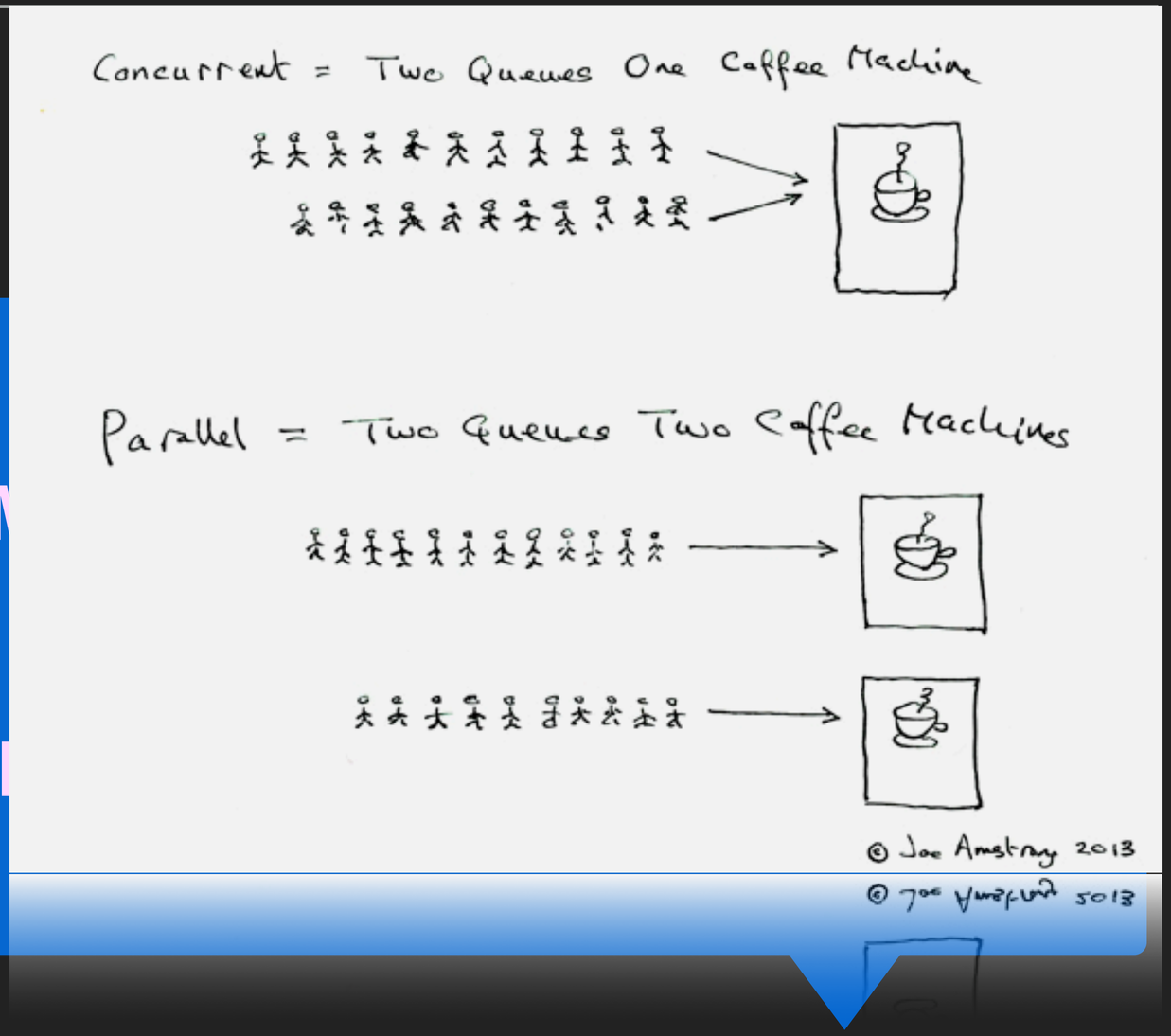
EXAMPLE 1.

CONCURRENT:

TWO QUEUES AND ONE COFFEE M

PARALLEL:

TWO QUEUES AND TWO COFFEE M



Joe Armstrong – “Erlang and other stuff”

EXAMPLE 2.

CONCURRENCY

CAN HAPPEN WHEN THERE IS ONLY ONE CORE – CONCURRENCY IS ABOUT DESIGN – IMPROVED PERFORMANCE IS A SIDE EFFECT

PARALLELISM

REQUIRES TWO PROCESSOR CORES – NO MATTER THE LANGUAGE/RUNTIME – A PROCESSOR CORE CAN ONLY EXECUTE ONE INSTRUCTION AT A TIME

Rob Pike "Concurrency is not Parallelism"

EXAMPLE 3 —— RUBY PRIMITIVES CATEGORIZED

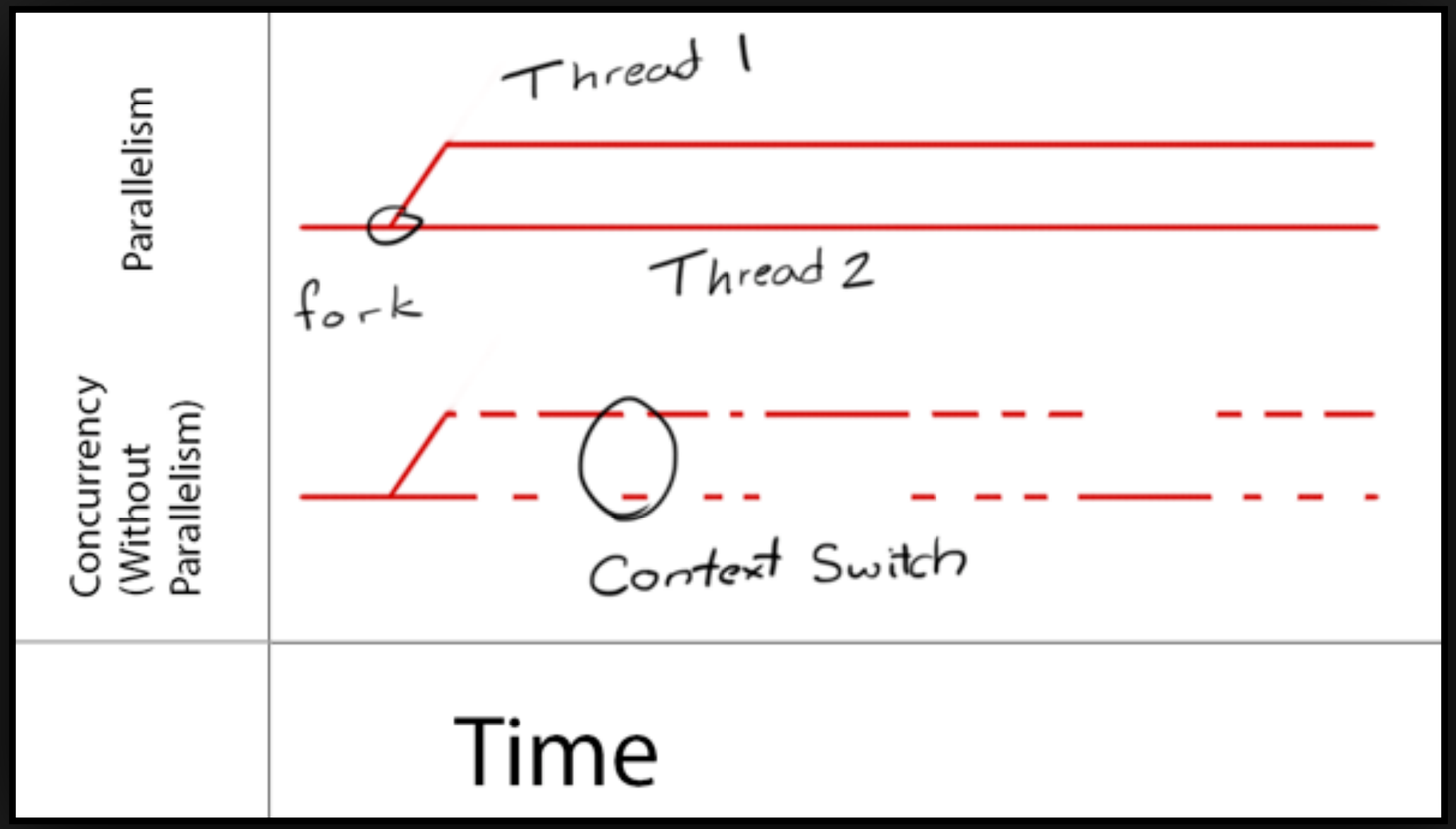
CONCURRENT

RUBY THREADS, FIBERS, CELLULOID, GUILD (RUBY 3.0)

PARALLEL

OS PROCESSES: UNICORN WORKERS, SIDEKIQ WORKERS

CONCURRENCY VS PARALLELISM



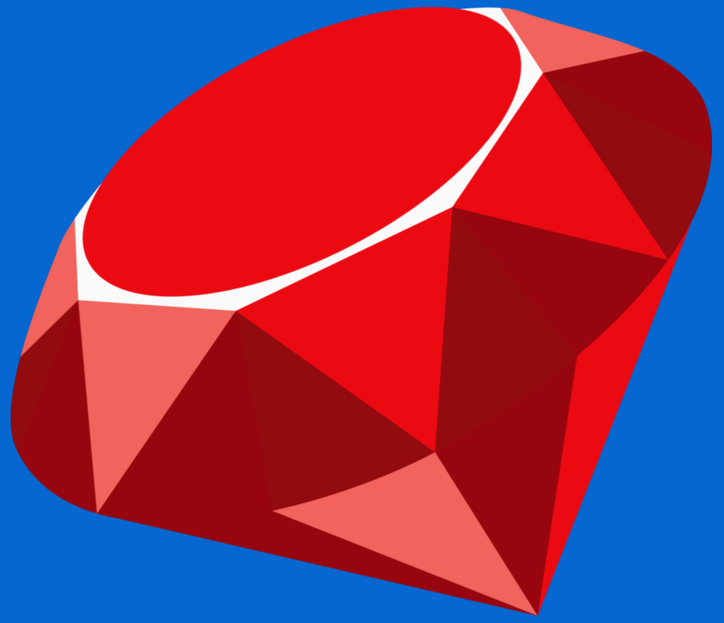
MULTITHREADING IN THE UNIX OS

- UNIX OS manages threads of execution across all running processes
- 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.
- You can monitor the number of context switches per process using system tools, such as:

`dtrace, latency, fs_usage`

Critically, each core can only run one thread at a time 🚀

SO HOW DOES THIS AFFECT RUBY?



GLOBAL INTERPRETER LOCK

"GIL"

WHAT IS THIS "GLOBAL INTERPRETER LOCK"?

- Every programming language/runtime must have internal logic to deal with 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

FACT:

RUBY USES THE GIL TO PROTECT ITS INTERNAL STATE ACROSS OS CONTEXT SWITCHES

IMPLICATIONS OF “GIL" IN PRACTICE

- **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 MRI Ruby is unable to experience true parallelism (this is not the case with jRuby and Rubinius)**
- **The Ruby runtime guarantees that it will always be in a consistent internal state – but it makes no guarantees about your code**

LET'S LOOK AT A COUPLE OF EXAMPLES TO DIG A BIT DEEPER INTO GIL'S IMPLICATIONS....

TO USE CONCURRENCY OR NOT TO USE? THAT IS THE QUESTION.

ASSUMING MRI RUBY...

```ruby
# EXAMPLE 1: COMPUTE CHECKSUMS

require 'digest/md5'

10.times.map do
  Thread.new do
    Digest::MD5.hexdigest(rand)
  end
end.each(&:value)
```

```ruby
# EXAMPLE 2: LOAD REMOTE URL
require 'open-uri'

10.times.map do
  Thread.new do
    open('http://zombo.com')
  end
end.each(&:value)
```

EXAMPLE 1: COMPUTING CHECKSUMS – SOLUTION

- Let's say we we need to compute a checksum on each object in an array.
- We know that computing checksum is an operation on CPU and it requires no IO.

```ruby
# EXAMPLE 1: COMPUTE CHECKSUMS

require 'digest/md5'

10.times.map do
  Thread.new do
    Digest::MD5.hexdigest(rand)
  end
end.each(&:value)
```

- **Question:**
  Should our implementation use threads (or a thread pool) to parallelize our computation and complete the task faster?
- **Answer:**
  Absolutely not. If each checksum computation requires no IO and only CPU, there is zero benefit in using threads in the MRI Ruby.

EXAMPLE 2: CRAWLING THE WEB

- Let's say we want download remote content given an array of URLs.
- We know that reading from a remote URL over the network is an operation on IO

```ruby
# EXAMPLE 2: LOAD REMOTE URL
require 'open-uri'

10.times.map do
  Thread.new do
    open('http://zombo.com')
  end
end.each(&:value)
```

- **Question:**
  Should our implementation create threads to parallelize our computation so that we can fetch URLs a lot faster?
- ✔
  Definitely!
  Use a **Thread Pool** to prevent creating too many threads Each thread will spend some time waiting on IO: network is slow.

CONCURRENCY TOOLBOX

3.1 CONCURRENCY IN RUBY STANDARD LIBRARY

RUBY SUPPORTS THREE CORE FORMS OF CONCURRENCY

- Multi-Process (Puma, Unicorn, Sidekiq + sidekiq-pool)
- Multi-Threaded (Puma, Sidekiq). Fibers and Guild are here too.
- Evented (Thin)

BUILT-IN PRIMITIVES

- Ruby offers ONLY ONE thread-safe class: Queue!
- One or more threads can add work to the queue:

```
require 'thread'
@queue = Queue.new
@queue << job
```

- One or more threads can retrieve the work concurrently with other threads:

```
url = @queue.pop
```

- It's a blocking call!
- If Queue is insufficient for your needs, there are gems that provide thread-safe versions of Array, Hash, and other standard data structures.

BUILT-IN PRIMITIVES

THREAD LOCAL, MUTEXES AND CONDITIONAL VARIABLES

- Thread Local: variables stored here are globally scoped to the current thread.

```
Thread.current[:redis] = Redis.new
```

- Mutex: protect shared mutable data against race conditions
- Conditional variable: elegant notification mechanism to avoid infinite loops.

BUILT-IN PRIMITIVES

MUTEX IN ACTION

- Mutexes provide a mechanism for multiple threads to synchronize access to a critical portion of code. In other words, they help bring some order, and some guarantees, to the world of multi-threaded chaos.
- The name 'mutex' is shorthand for 'mutual exclusion.'
- If you wrap some section of your code with a mutex, you guarantee that no two threads can enter that section at the same time.

```ruby
require 'thread'

class BlockingQueue
  def initialize
    @storage = Array.new
    @mutex = Mutex.new
  end

  def push(item)
    @mutex.synchronize do
      @storage.push(item)
    end
  end
```

BUILT-IN PRIMITIVES

CONDITIONAL VARIABLE IN ACTION

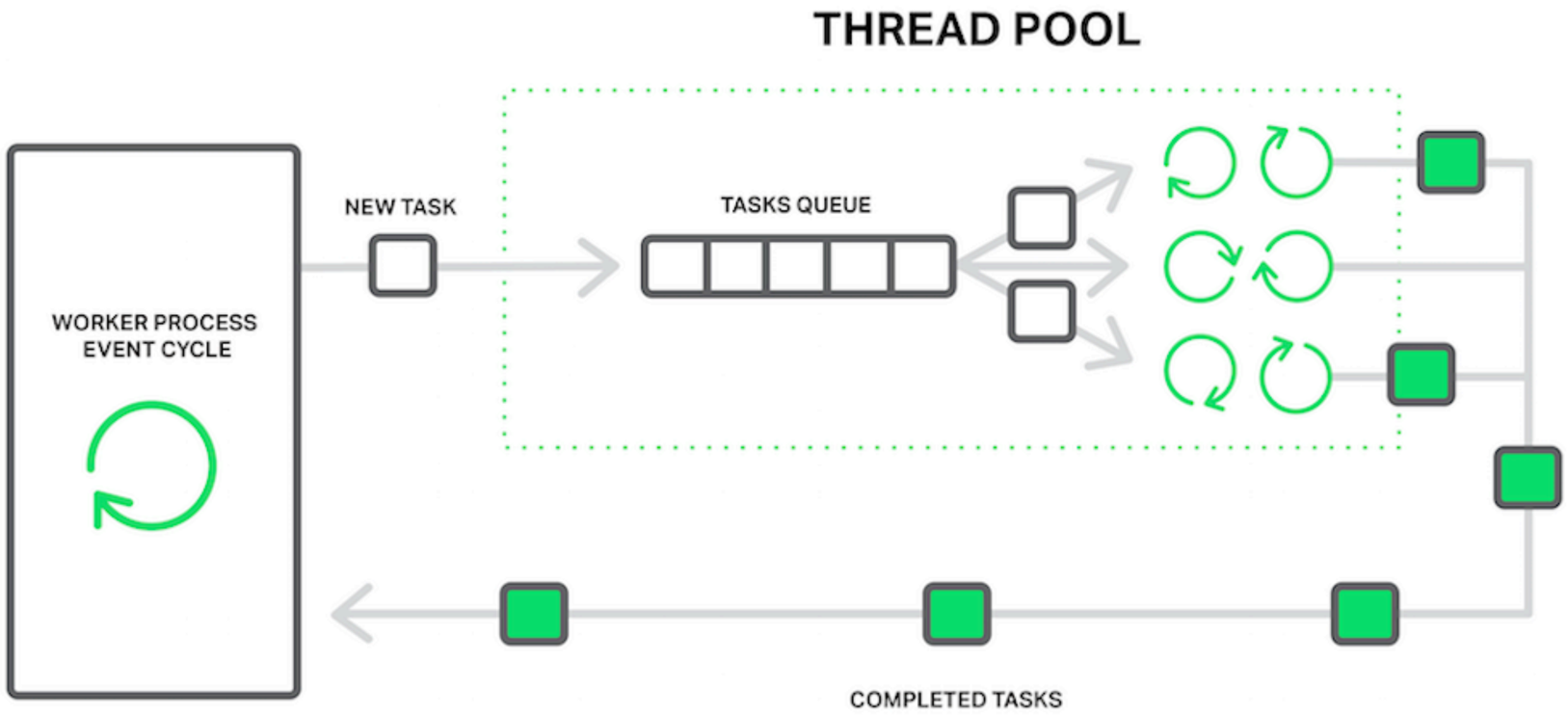
- A ConditionVariable can be used to signal one (or many) threads when some event happens, or some state changes, whereas mutexes are a means of synchronizing access to resources.
- Condition variables provide an inter-thread control flow mechanism.
- For instance, if one thread should sleep until it receives some work to do, another thread can pass it some work, then signal it with a condition variable to keep it from having to constantly check for new input.

```ruby
require 'thread'

class BlockingQueue
  def initialize
    @storage = Array.new
    @mutex = Mutex.new
    @condvar = ConditionVariable.new
  end

  def push(item)
    @mutex.synchronize do
      @storage.push(item)
      @condvar.signal
    end
  end

  def pop
    @mutex.synchronize do
      while @storage.empty?
        @condvar.wait(@mutex)
      end

      @storage.shift
    end
  end
end
```

3.2 CONCURRENCY PRIMITIVES PROVIDED BY GEMS

THREAD POOL

- **Most well-known concurrency primitive is a thread pool: a data structure that maintains either fixed, or capped number of threads that can perform work by reading a thread-safe Queue:**



THREAD POOLS — RUBY-THREAD

- Gem ruby-thread provides an easy extension to the built-in Thread Class:

```
require 'thread/pool'

pool = Thread.pool(4)

10.times {
  pool.process {
    sleep 2

    puts 'lol'
  }
}

pool.shutdown
```

THREAD POOLS — CONCURRENT-RUBY

- Gem concurrent-ruby provides a massive list of concurrent and thread-safe primitives, and is likely the most definitive concurrency ruby library today.

```ruby
# create a pool with fixed 5 threads
pool = Concurrent::FixedThreadPool.new(5)
pool.post do
   # perform some parallel work
end

# As with all thread pools, execution resumes
# immediately here in the caller thread

# this pool is smarter — it can resize itself
# based on the queue size
pool = Concurrent::ThreadPoolExecutor.new(
   min_threads: 5,
   max_threads: 5,
   max_queue: 100,
   fallback_policy: :caller_runs
)
```

CONCURRENT-RUBY

- **This gem contains everything you'll ever need to correctly use concurrency in ruby.**
- **Alas, it also contains plenty of things you not likely to need, ever.**

General-purpose Concurrency Abstractions

- Async: A mixin module that provides simple asynchronous behavior to a class. Loosely based on Erlang's gen_server.
- ScheduledTask: Like a Future scheduled for a specific future time.
- TimerTask: A Thread that periodically wakes up to perform work at regular intervals.
- Promises: Unified implementation of futures and promises which combines features of previous `Future`, `Promise`, `IVar`, `Event`, `dataflow`, `Delay`, and (partially) `TimerTask` into a single framework. It extensively uses the new synchronization layer to make all the features **non-blocking** and **lock-free**, with the exception of obviously blocking operations like `#wait`, `#value`. It also offers better performance.

Thread-safe Value Objects, Structures, and Collections

Collection classes that were originally part of the (deprecated) `thread_safe` gem:

- Array A thread-safe subclass of Ruby's standard Array.
- Hash A thread-safe subclass of Ruby's standard Hash.
- Set A thread-safe subclass of Ruby's standard Set.
- Map A hash-like object that should have much better performance characteristics, especially under high concurrency, than `Concurrent::Hash`.
- Tuple A fixed size array with volatile (synchronized, thread safe) getters/setters.

Value objects inspired by other languages:

- Maybe A thread-safe, immutable object representing an optional value, based on Haskell Data.Maybe.

Structure classes derived from Ruby's Struct:

- ImmutableStruct Immutable struct where values are set at construction and cannot be changed later.
- MutableStruct Synchronized, mutable struct where values can be safely changed at any time.
- SettableStruct Synchronized, write-once struct where values can be set at most once, either at construction or any time thereafter.

Thread-safe variables:

- Agent: A way to manage shared, mutable, *asynchronous*, independent state. Based on Clojure's Agent.
- Atom: A way to manage shared, mutable, *synchronous*, independent state. Based on Clojure's Atom.
- AtomicBoolean A boolean value that can be updated atomically.
- AtomicFixnum A numeric value that can be updated atomically.
- AtomicReference An object reference that may be updated atomically.
- Exchanger A synchronization point at which threads can pair and swap elements within pairs. Based on Java's Exchanger.

RUBY GEM — PARALLEL

- Gem parallel provides an easy to way to start additional ruby processes and distribute work among them.
  It supports both a thread pool, and a process pool, as shown in the example:

```ruby
Parallel.each(User.all, in_threads: 8) do |user|
  ActiveRecord::Base.connection_pool.with_connection do
    user.update_attribute(:some_attribute, some_value)
  end
end

# maybe helps: reconnect once inside every fork
Parallel.each(User.all, in_processes: 8) do |user|
  @reconnected ||= User.connection.reconnect! || true
  user.update_attribute(:some_attribute, some_value)
end
```

SINGLE & MULTI-CORE CONCURRENCY IN RUBY — NOTABLE GEMS

- **concurrent-ruby:** thread-safe primitives such as Array, Hash,..
- **sucker-punch**: single-process async jobs
- **eventmachine**: event-based concurrency with fibers
- **ruby-thread**: extensions to Thread class such as a Pool, etc.
- **parallel**: both multi-process and multi-threaded jobs
- **sidekiq**: multu-threaded multi-process job processor
- **celluloid**: actor model sidekiq is based on, **unmaintained.**
- **atomic**: atomic primitives, **deprecated in favor of ruby-concurrency**

RUBY GEMS OFFERING MULTI-PROCESS CONCURRENCY

- sidekiq
  — perhaps the easiest way to utilize cores (but requires Enterprise for multi-process)
- sidekiq-pool
  — an open source gem that manages a set of Sidekiq workers
- childprocess
  — great gem that spawns processes on the background
- unicorn, puma, and thin
  — all support multi-worker configuration
  - But only Puma & Thin support ruby multi-threading.

HOW MANY THREADS?
HOW MANY PROCESSES?

IN THIS SECTION WE DISCUSS HOW ONE WOULD FIGURE OUT IDEAL SETTINGS FOR THE NUMBER OF RUBY PROCESSES, AND THE NUMBER OF RUBY THREADS TO CONFIGURE EACH PROCESS WITH.

IT'S NOT A TRIVIAL EXCERCISE, BUT IF DONE RIGHT, ENSURES FULL UTILIZATION OF THE VIRTUALIZED HARDWARE.

IN OTHER WORDS — IT SAVES $$.

Your typical CFO.

DETERMINING THE NUMBER OF PROCESSES TO RUN

- This part is easy — if you are able to start multiple Ruby processes (eg, if you are running puma, unicorn or sidekiq), start as many ruby processes as you have CPU cores available to you on your virtual instance.

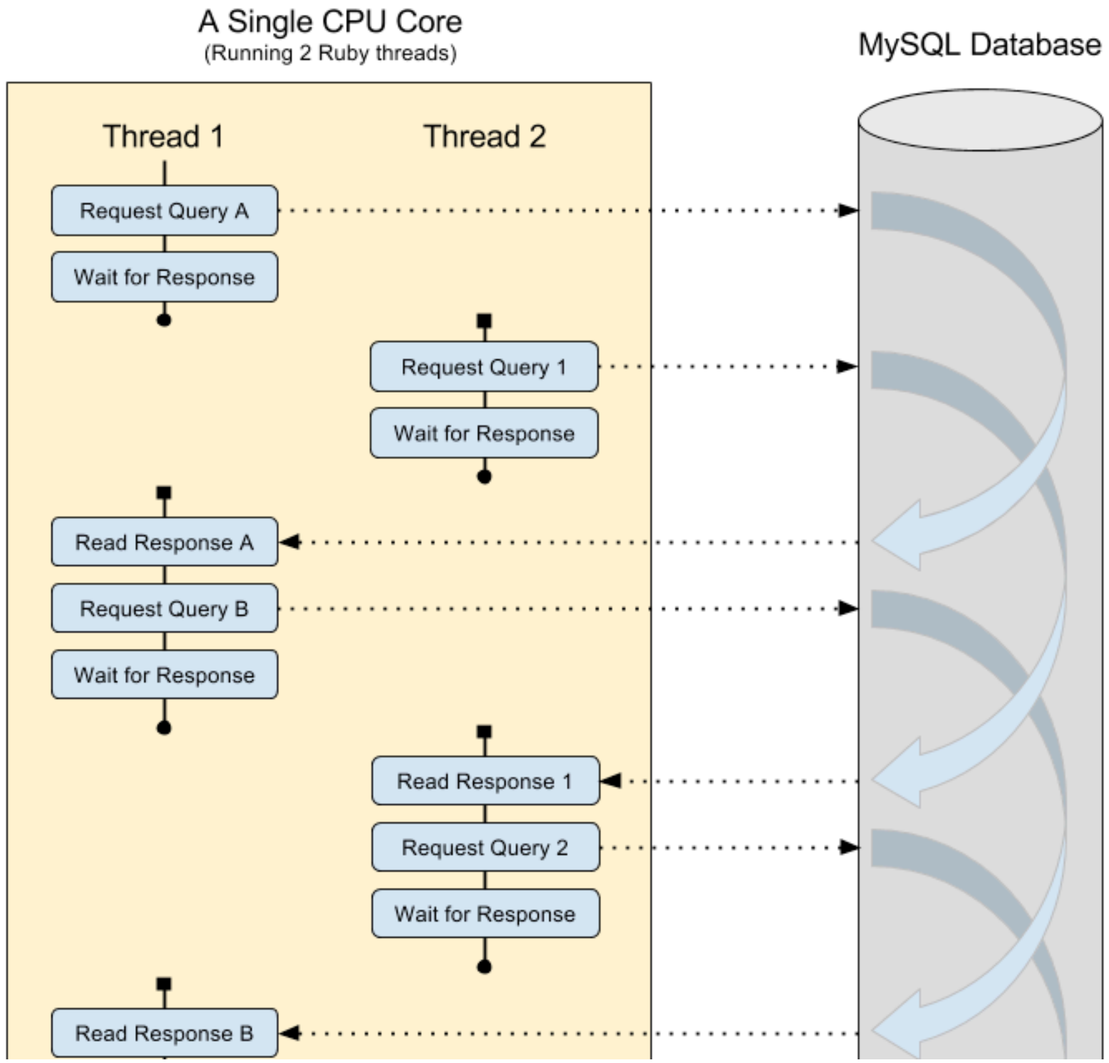
That's it. Have 16 cores? Start 16 ruby processes.

DETERMINING THE NUMBER OF THREADS

- Figuring out number of threads is a bit trickier.
- Sidekiq defaults to 15 threads per process.
  This can be quite a lot.
- The truth is you must monitor the process and observe "context switches" — if they happen a lot, reduce # of threads. Otherwise increase.

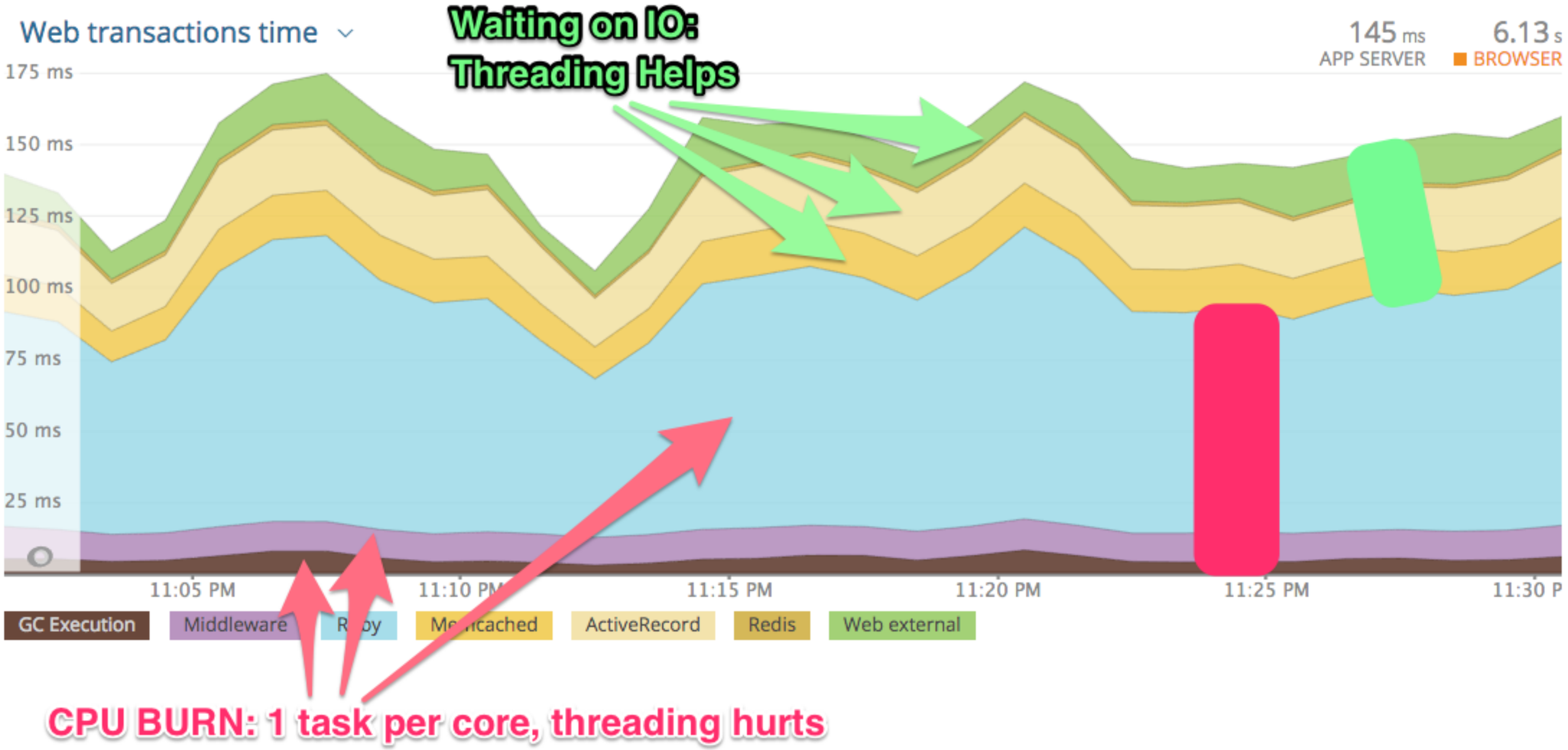
YOU CAN USE NEWRELIC AND DATADOG TO DETERMINE THE OPTIMAL NUMBER OF THREADS IN YOUR APPLICATION...

ZOOMING INTO A MULTI-THREADED RUBY PROCESS



- A ruby process with two threads can take advantage of pauses caused by waiting on IO (i.e. database response, file read, network, etc).

READING NEW RELIC: WAITING ON IO VERSUS CPU BURN



THREAD SAFETY

WHAT MAKES MULTI-THREADED RUBY PROGRAM SAFE?

- Threads should never change variables that are visible to other threads.
  - This includes class instance variables, class globals, and any other variables visible to threads.
  - If you must access it, wrap it in a Mutex, like so:

- Rule of thumb: avoid using Thread.new directly inside of a web request lifecycle...
  - Instead — use Sidekiq + Sidekiq Batch API which allows UI to query the status of the job.

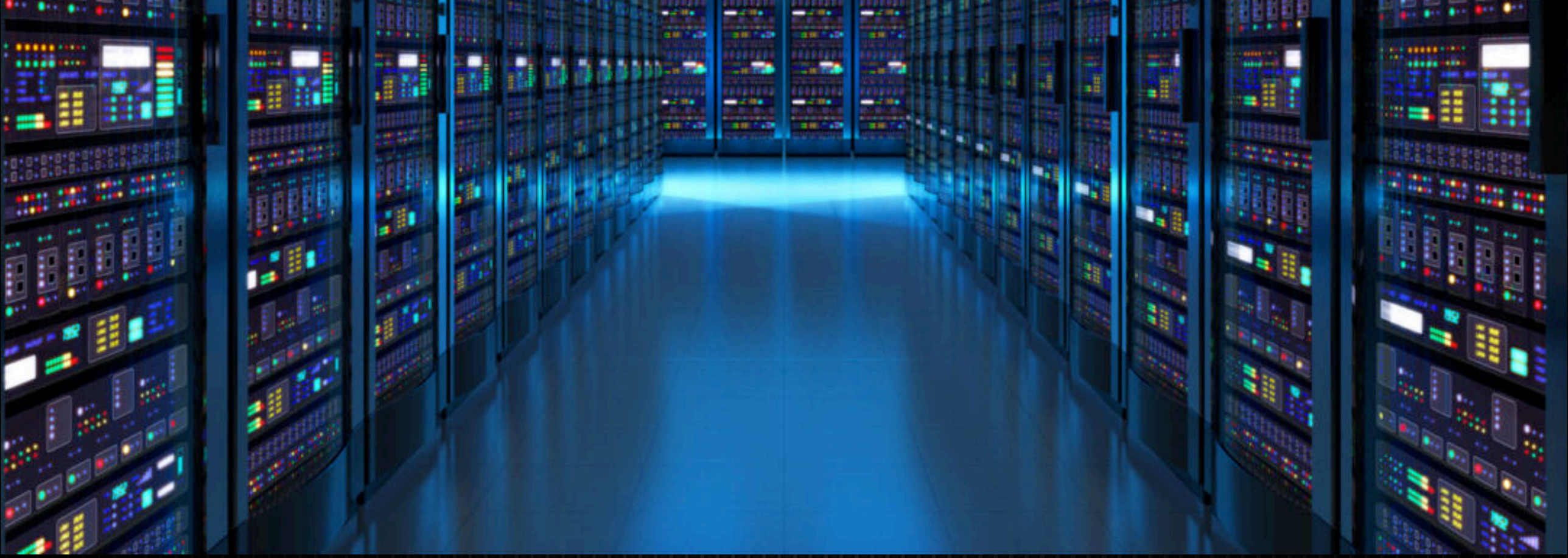
```ruby
require "thread"

class Counter
  def initialize
    @counter = 0
    @mutex = Mutex.new
  end

  def increment
    @mutex.synchronize do
      @counter += 1
    end
  end
end
```

WHAT MAKES A RUBY PROGRAM NOT THREAD SAFE?

- Simply put, when you share mutable state between threads
- None of the core data structures (except for Queue) in Ruby are thread-safe.
- Use Thread.current[] to store thread-specific variables
- Prefer to instantiate classes in each thread (like Rails instantiates Controllers for each web request).
- Use Queue to pass data between Ruby threads
- Avoid accessing and writing to Class Instance variables or any other globals.

ISSUES RESULTING FROM MIS-USE OF CONCURRENCY

- may saturate all CPU
- may underutilize the CPU
- may lock ruby processes and web request processing
- may take production down
- may produce extremely hard to find bugs
- may result in you having to buy beer for your teammates...



THREAD LIGHTLY

ACKNOWLEDGEMENTS AND REFERENCES

- https://pragprog.com/book/jsthreads/working-with-ruby-threads
- https://github.com/meh/ruby-thread
- https://www.toptal.com/ruby/ruby-concurrency-and-parallelism-a-practical-primer
- https://www.slideshare.net/JerryDAntonio/everything-you-know-about-the-gil-is-wrong
- https://github.com/jdantonio/concurrent-ruby-presentation
- https://joearms.github.io/published/2013-04-05-concurrent-and-parallel-programming.html
- https://brianchan.us/2017/05/27/concurrency-vs-parallelism/
- https://www.codebasehq.com/blog/ruby-threads-queue
- https://github.com/dasch/ruby-csp

QUESTIONS?

KONSTANTIN GREDESKOUL

https://github.com/kigster

https://twitter.com/kig

https://kig.re

https://reinvent.one/