livingsocial

don't fear the threads

simplify your life with JRuby

david.copeland@livingsocial.com
@davetron5000
www.naildrivin5.com
www.awesomecommandlineapps.com

```
{
  :me => {
    :tech_lead =>
    “LivingSocial”
  }
}
```

First Assignment

Write a

Write a Command Line App

The Pragmatic Programmers

Build Awesome Command-Line Applications in Ruby

Control Your Computer, Simplify Your Life

David Bryant Copeland

Edited by John Osborn

The Facets of Ruby Series

Write a Command Line App

Write a Command Line App

To charge credit cards faster

NO NEW HARDWARE

THREADS!


D 406
SP. WISTARIA
100 YARDS
91.4 METRES
D.303
PLUM
SYKO
100 YARDS
D.389

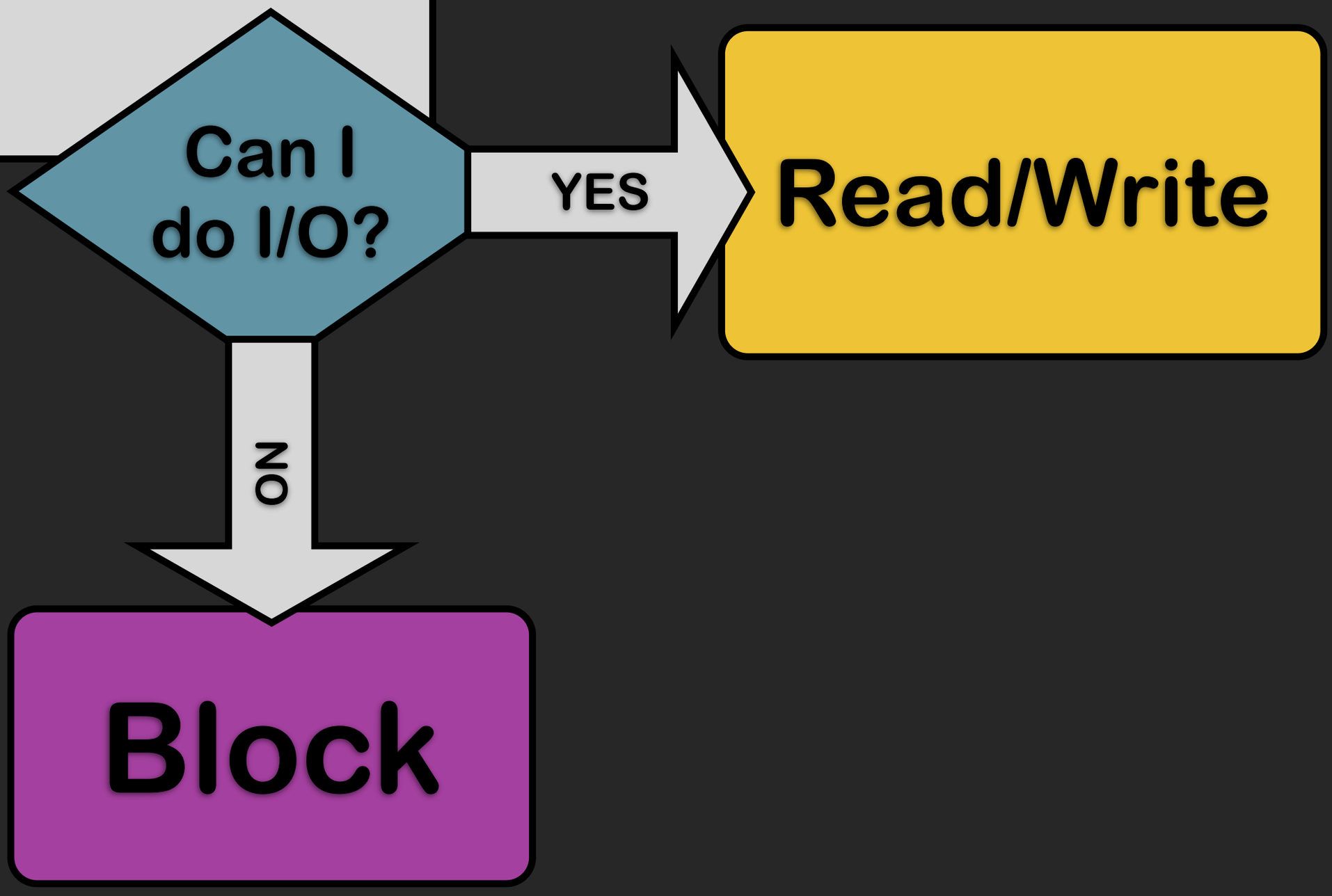
What problem are we solving?

Do some I/O

Compute Stuff

Do some I/O

Do some I/O



Do some I/O

Block

Block

Block

Let someone else work

Block

Maximize Resources

Will our code be

Will our code be easy to write?

Will our code be easy to understand?

easy to write?

Will our code be

easy to test?
easy to understand?
easy to write?

Are we maximizing our resources?

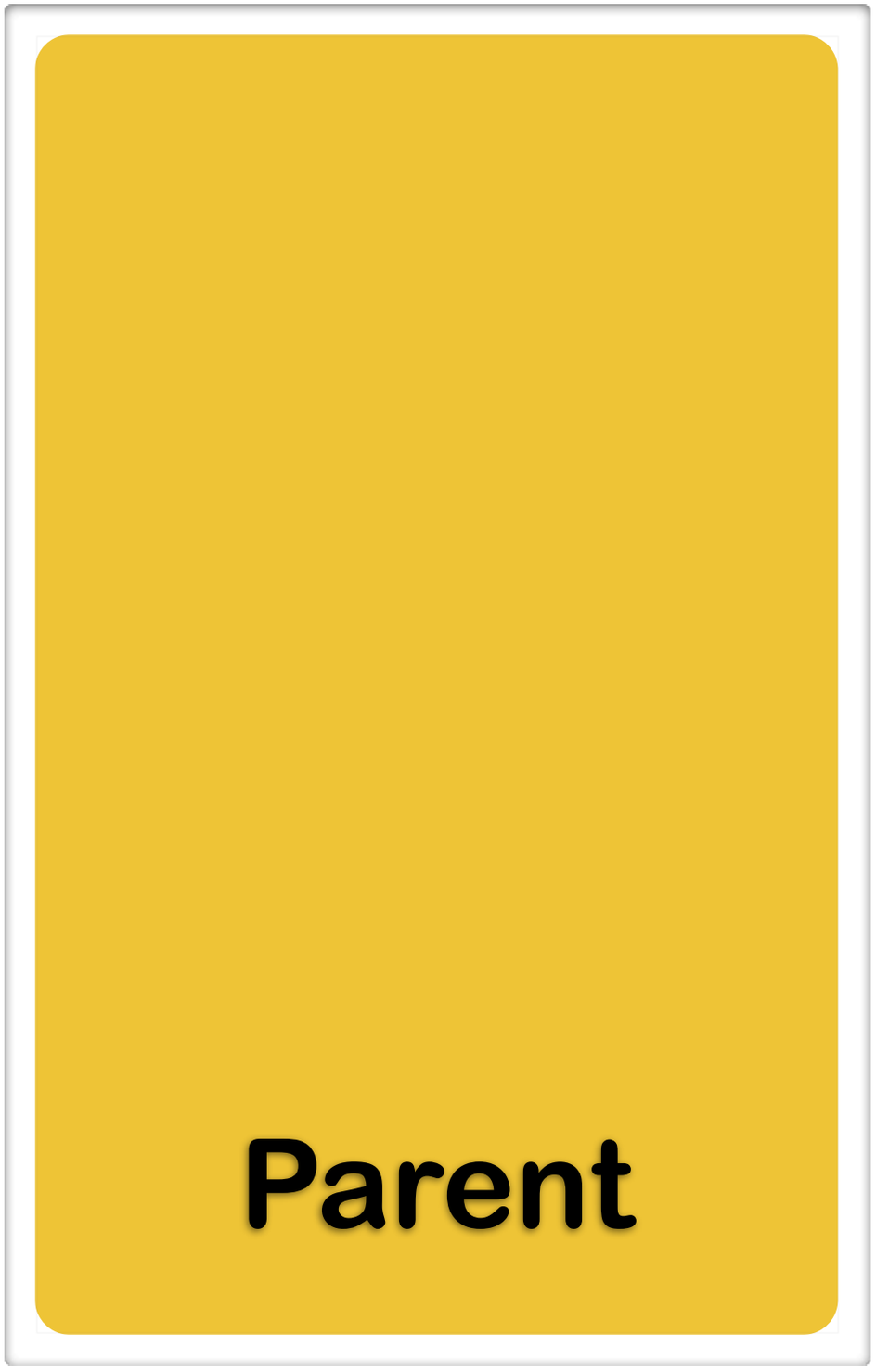
Run lots of processes

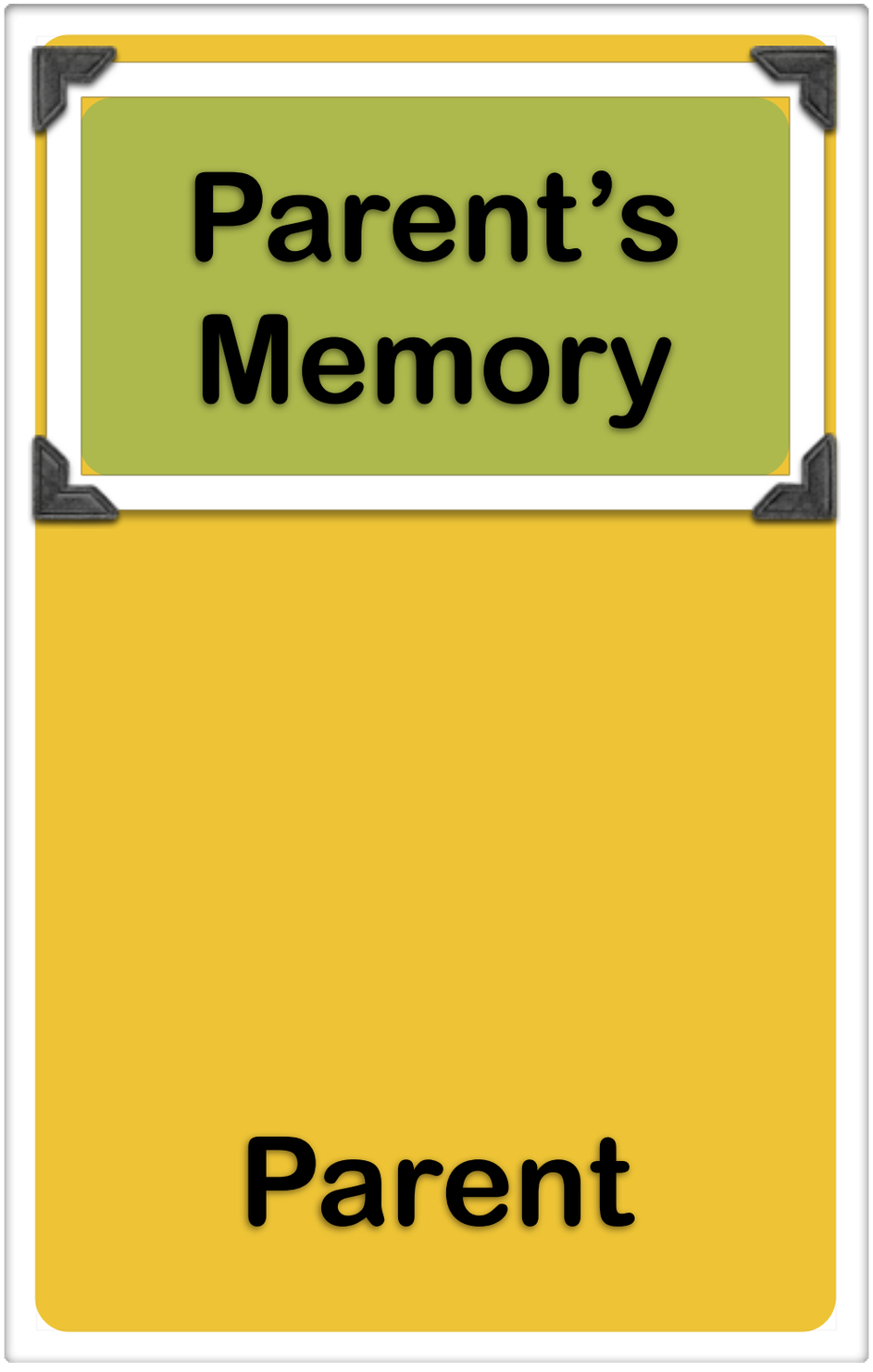
```
fork {
    # your code
}
```

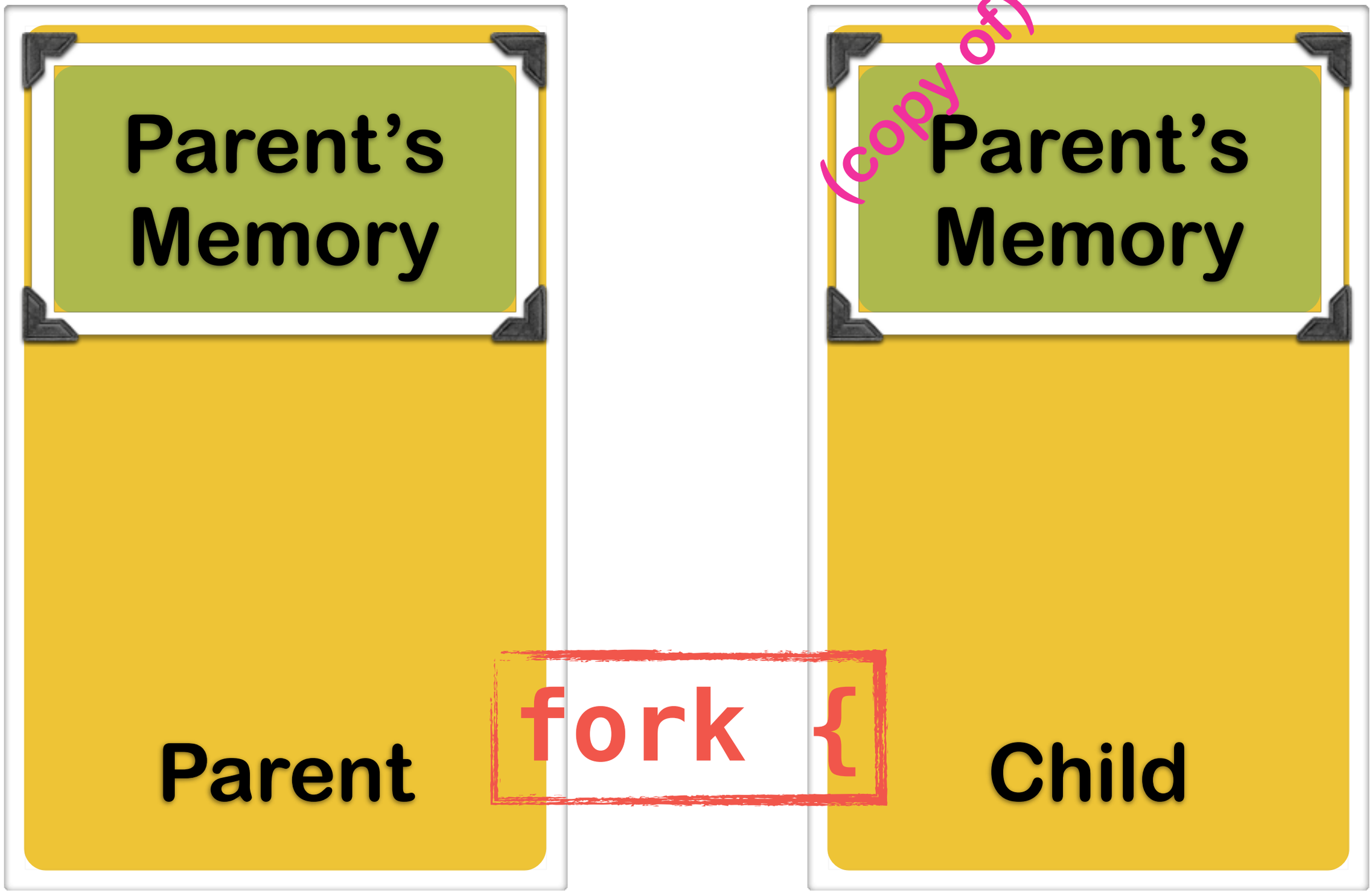
easy to
understand

doesn't

maximize resources


Parent


Parent's Memory
Parent


Parent's Memory
(copy of)
Parent's Memory
fork {
Parent
Child

Event-based I/O

```ruby
require 'em-http'
include EM::HttpRequest

EM.run {

  # some of your code that does IO
  some_callback { |results_of_io|
    # the rest of your code that
    # uses the results
  }
}
```

```ruby
require 'em-http'
include EM::HttpRequest

EM.run {

  # some of your code that does IO
  some_callback { |results_of_io|
    # the rest of your code that
    # uses the results
  }
}
```

Need to do I/O again?

```ruby
require 'em-http'
include EM::HttpRequest

EM.run {

  # some of your code that does SOME of your I/O
  some_callback { |results_of_io|
    # use the results
    # now some more code that does some MORE I/O
    some_other_callback { |results_of_more_io|
      # use the results of THIS I/O
    }
  }
}
```

```ruby
require 'em-http'
include EM::HttpRequest

EM.run {

  # some of your code that does SOME of your I/O
  some_callback { |results_of_io|
    # use the results
    # now some more code that does some MORE I/O
    some_other_callback { |results_of_more_io|
      # use the results of THIS I/O

      # SERIOUSLY, more I/O?!??!@  How much do you need?
      yet_more_callbacks { |moar_resultz|
        # How many levels deep are we now?!
      }
    }
  }
}
```

```ruby
require 'em-http'
include EM::HttpRequest

EM.run {

  # some of your code that does SOME of your I/O
  some_callback { |results_of_io|
    # use the results
    # now some more code that does some MORE I/O
    some_other_callback { |results_of_more_io|
      # use the results of THIS I/O

      # SERIOUSLY, more I/O?!??!@  How much do you need?
      yet_more_callbacks { |moar_resultz|
        # How many levels deep are we now?!
      }
    }
  }
}
```

NOT easy to understand

kinda

maximizes resources

Entire call chain must be evented

Not true parallelism

Not true parallelism

unless you run lots of processes

What's parallelism?

Concurrency

Parallelism

Parallelism

Code running at the same time as other code

Parallelism

Impossible with only one CPU

Code running at the same time as other code

Event-based I/O

NOT easy to understand

kinda maximizes resources

Threads

```
Thread.new {
    # your code
}
```

```
fork {
    # your code
}
```

```
Thread.new {
    # your code
}
```

Need to do I/O?

Need to do I/O?

Not a problem

Need to do I/O THREE TIMES?

Need to do I/O THREE TIMES?

Not a problem

What if I need to calculate π to the 2,345,123rd decimal place?

NOT A PROBLEM

Threads achieve true parallelism

easy to

understand

maximizes

resources

Threads don't work in Ruby

Threads don't work in Ruby...right?

MRI (aka C Ruby)

MRI (aka C Ruby)

Thread is an OS thread
(except in 1.8)

MRI (aka C Ruby)

Thread is an OS thread
(except in 1.8)

GIL :(

MRI (aka C Ruby)

Thread is an OS thread
(except in 1.8)

GIL :(

I/O will cause a context switch

Rubinius

Rubinius

Thread is an OS thread

Rubinius

Thread is an OS thread

No GIL!

Rubinius

Thread is an OS thread

No GIL!

True parallelism

JRuby

JRuby

Thread is a JVM Thread (which is an OS thread)

JRuby

Thread is a JVM Thread (which is an OS thread)

Context switch for variety of reasons

JRuby

Thread is a JVM Thread (which is an OS thread)

Context switch for variety of reasons

True parallelism

JRuby

BATTLE TESTED

So, you've decided to use Threads...

You need to know four things

Start & Manage

Start & Manage

(Dealing with) Shared State

Start & Manage

(Dealing with) Shared State

Using Third Party Libraries

Start & Manage

(Dealing with) Shared State

Using Third Party Libraries

Context Switching

Start & Manage Threads

Chaos

```ruby
Thread.new {
    # your code
}
Thread.new {
    # moar code
}
# etc
```

Hand-Roll

```ruby
threads = []
threads << Thread.new {
  # your code
}
threads << Thread.new {
  # moar code
}
# etc
threads.each(&:join)
# All threads have completed
exit 0
```

java.util.concurrent

```
import java.util.concurrent # JRuby only

service =
    Executors.new_fixed_thread_pool(100)

service.execute {
  # your code
}
service.execute {
  # some other code
}
```

Method Summary

static Callable<Object>	**callable**(PrivilegedAction<?> action) Returns a Callable object that, when cal
static Callable<Object>	**callable**(PrivilegedExceptionAction<?> Returns a Callable object that, when cal
static Callable<Object>	**callable**(Runnable task) Returns a Callable object that, when cal
static <T> Callable<T>	**callable**(Runnable task, T result) Returns a Callable object that, when cal
static ThreadFactory	**defaultThreadFactory**() Returns a default thread factory used to c
static ExecutorService	**newCachedThreadPool**() Creates a thread pool that creates new thr
static ExecutorService	**newCachedThreadPool**(ThreadFactory thre Creates a thread pool that creates new thr ThreadFactory to create new threads when neec
static ExecutorService	**newFixedThreadPool**(int nThreads) Creates a thread pool that reuses a fixed n
static ExecutorService	**newFixedThreadPool**(int nThreads, Threa Creates a thread pool that reuses a fixed n when needed.

Method Summary

static Callable<Object>	**callable**(PrivilegedAction<?> action) Returns a Callable object that, when cal
static Callable<Object>	**callable**(PrivilegedExceptionAction<?> Returns a Callable object that, when cal
static Callable<Object>	**callable**(Runnable task) Returns a Callable object that, when cal
static <T> Callable<T>	**callable**(Runnable task, T result) Returns a Callable object that, when cal
static ThreadFactory	**defaultThreadFactory**() Returns a default thread factory used to c
static ExecutorService	**newCachedThreadPool**() Creates a thread pool that creates new thr
static ExecutorService	**newCachedThreadPool**(ThreadFactory thre Creates a thread pool that creates new thr ThreadFactory to create new threads when need
static ExecutorService	**newFixedThreadPool**(int nThreads) Creates a thread pool that reuses a fixed n
static ExecutorService	**newFixedThreadPool**(int nThreads, Threa Creates a thread pool that reuses a fixed n when needed.

Method Summary

static Callable<Object>	**callable**(PrivilegedAction<?> action) Returns a Callable object that, when ca
static Callable<Object>	**callable**(PrivilegedExceptionAction<?> Returns a Callable object that, when ca
static Callable<Object>	**callable**(Runnable task) Returns a Callable object that, when ca
static <T> Callable<T>	**callable**(Runnable task, T result) Returns a Callable object that, when ca
static ThreadFactory	**defaultThreadFactory**() Returns a default thread factory used to c
static ExecutorService	**newCachedThreadPool**() Creates a thread pool that creates new thr
static ExecutorService	**newCachedThreadPool**(ThreadFactory thre Creates a thread pool that creates new thr ThreadFactory to create new threads when need
static ExecutorService	**newFixedThreadPool**(int nThreads) Creates a thread pool that reuses a fixed n
static ExecutorService	**newFixedThreadPool**(int nThreads, Threa Creates a thread pool that reuses a fixed n when needed.

ExecutorService

ExecutorService

ExecutorService

`execute()`

ExecutorService

execute()

shutdown()

ExecutorService

execute()

shutdown()

await_termination()

ExecutorService

execute()

shutdown()

await_termination()

shutdown_now()

```
service = Executors.new_fixed_thread_pool(10)
tcp_erver = TCPServer.new("127.0.0.1",8080)




loop do {
   s = tcp_server.accept
   service.execute {
      s.puts calculate_pi()
      s.close
   }

}
```

```ruby
service = Executors.new_fixed_thread_pool(10)
tcp_server = TCPServer.new("127.0.0.1",8080)

Signal.trap('SIGINT') {
  service.shutdown
}

loop do {
  s = tcp_server.accept
  service.execute {
    s.puts calculate_pi()
    s.close
  }
  break if service.is_shutdown
}
```

```ruby
service = Executors.new_fixed_thread_pool(10)
tcp_server = TCPServer.new("127.0.0.1",8080)

Signal.trap('SIGINT') {
  service.shutdown
}

loop do {
  s = tcp_server.accept
  service.execute {
    s.puts calculate_pi()
    s.close
  }
  break if service.is_shutdown
}
service.await_termination(10,TimeUnit.SECONDS)
service.shutdown_now
```

So much more

So much more

`ScheduledExecutorService`

So much more

`ThreadFactory`

`ScheduledExecutorService`

So much more

Read the javadocs

`ThreadFactory`

`ScheduledExecutorService`

Shared State

```
results = []
Thread.new {

    results << some_result()

}
Thread.new {

    results << other_result()

}
```

```
mutex = Mutex.new

results = []
Thread.new {
   mutex.synchronize {
      results << some_result()
   }
}
Thread.new {
   mutex.synchronize {
      results << other_result()
   }
}
```

Avoid explicit locking/sync

```
results = ConcurrentLinkedQueue.new

Thread.new {
  results.offer(some_result())
}
Thread.new {
  results.offer(some_result())
}
```

```
import java.util.concurrent.atomic

flag = AtomicBoolean.new(true)
Thread.new {
  if flag.get {
    flag.set(false)
  }
}
Thread.new {
  flag.set(true)
}
```

So much more

So much more

`ConcurrentHashMap`

So much more

`AtomicReference`

`ConcurrentHashMap`

So much more

Read the javadocs

`AtomicReference`

`ConcurrentHashMap`

Third Party Libraries

Are they thread safe?

Probably

Probably

...but to be sure

variables

Global variables

Class variables

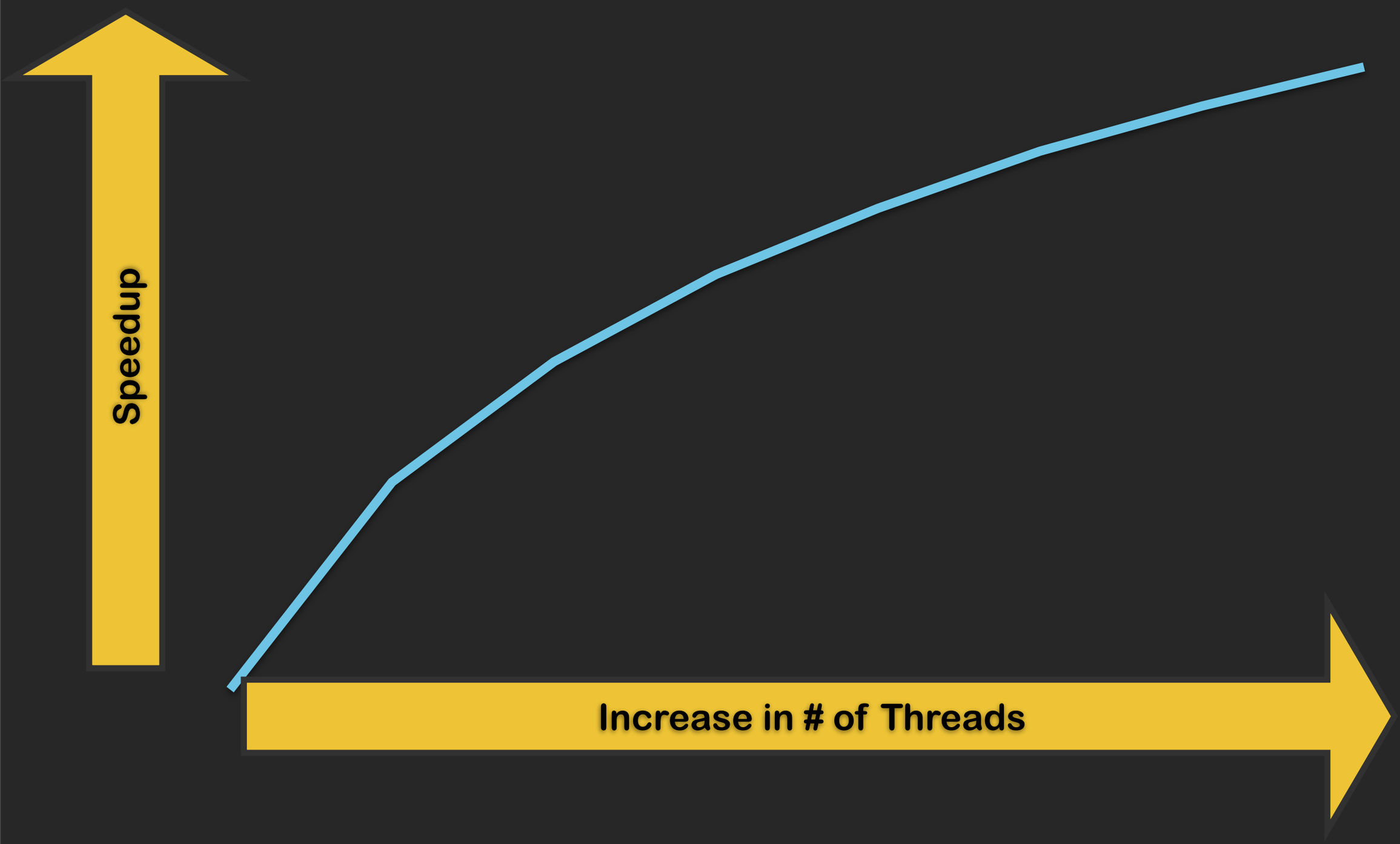
Global

Most are fine

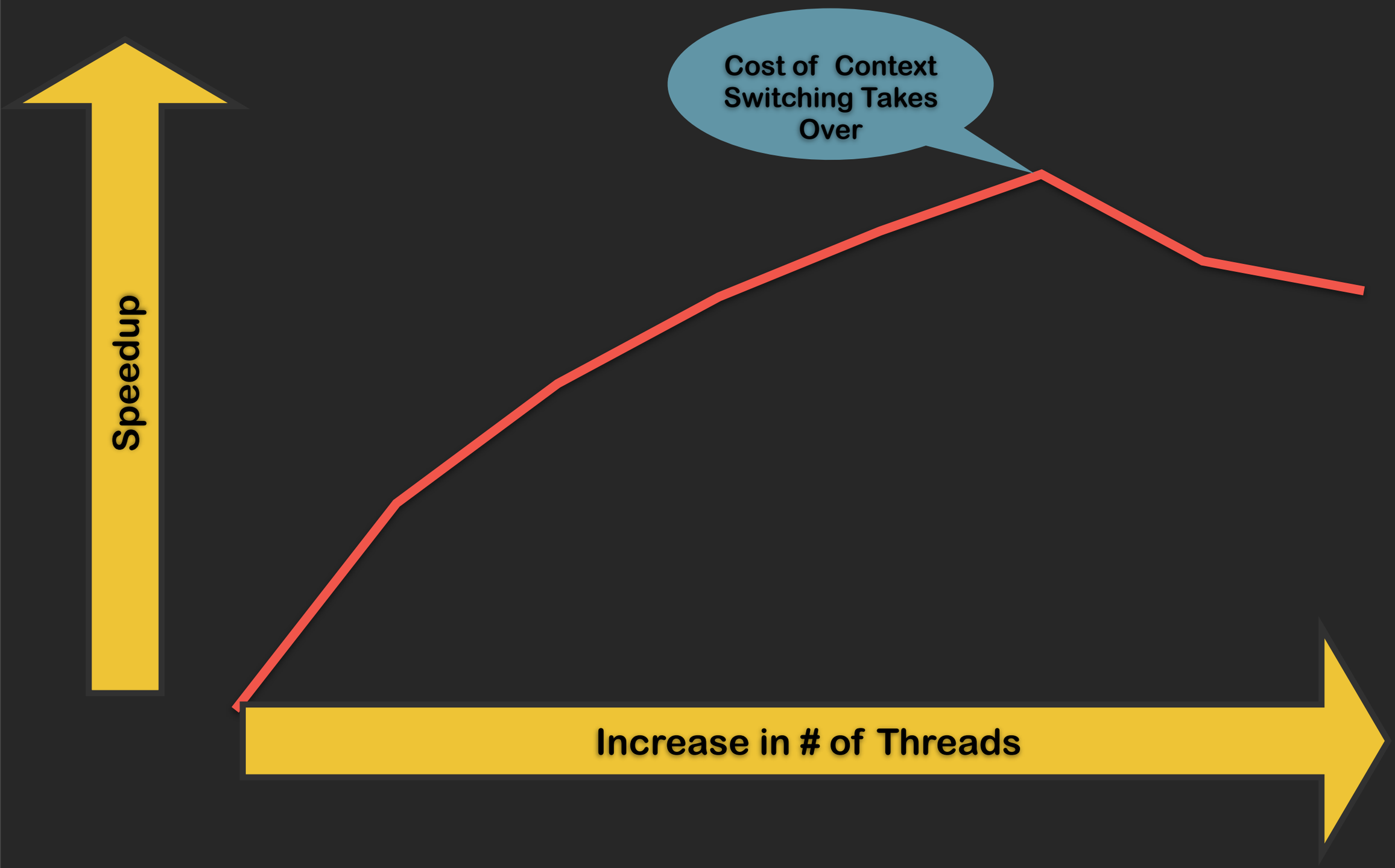
Context Switching

Amdahl's Law

Performance Gains



Performance Gains with Context Switching



Thread Pools

Worker

Worker

Worker

Worker

Worker

Worker

Worker

Worker

Worker

Thread Pool

Thread

Thread

Thread

Worker

Worker

Worker

Worker

Worker

Thread Pool

Thread

Worker

Thread

Worker

Thread

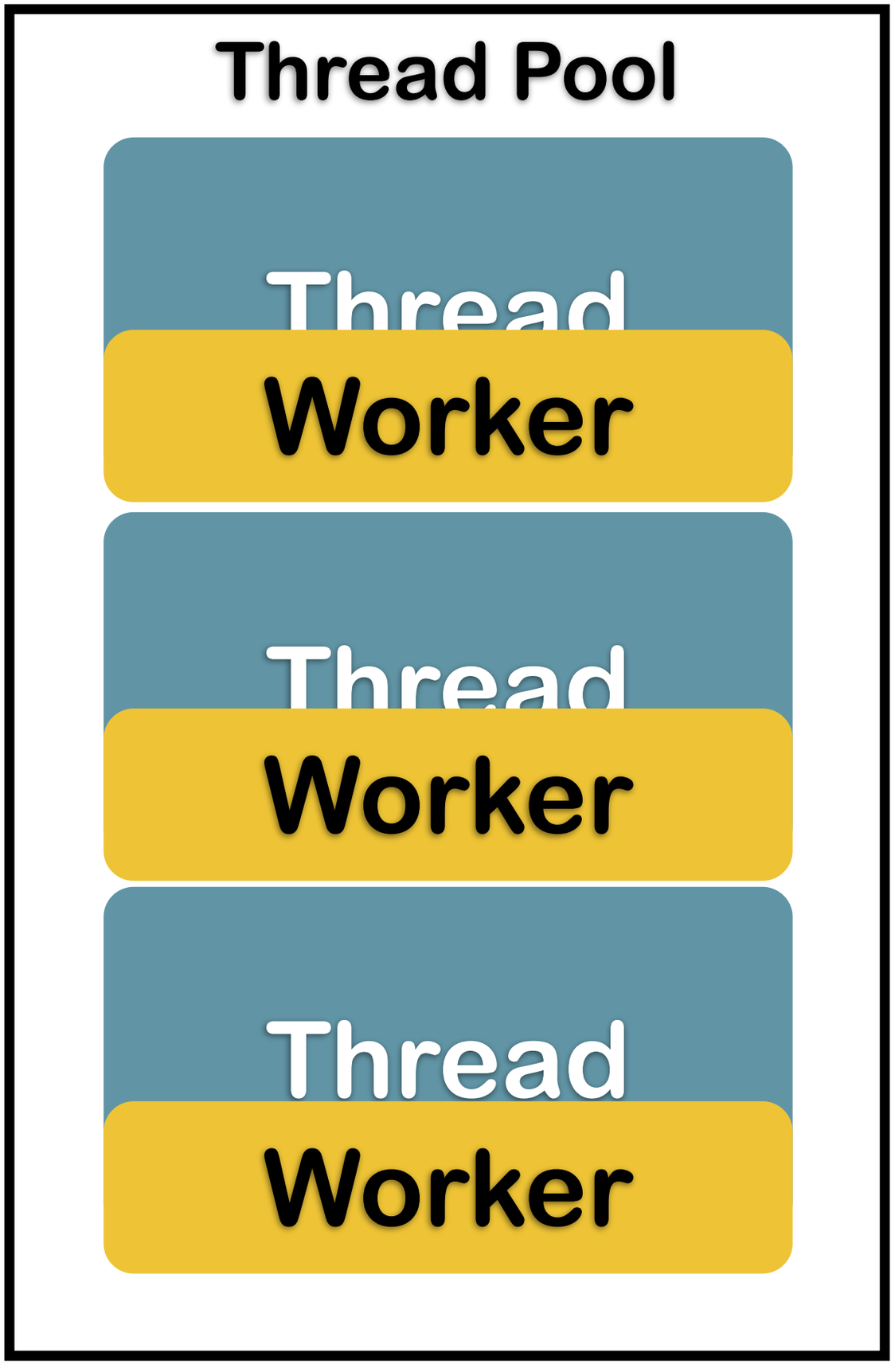
Worker

Worker

Worker

Worker

Worker



```
Thread.new {
    # your code
}
```

```
service.execute {
    # your code
}
```

Not usually a concern

Always use Threads, right?

Thread UNsafe Libraries

Evented

Simple, I/O bound task

Evented

Cannot use JRuby

Evented

Otherwise, Threads will simplify your code and maximize your resources

Semantic layers on top of evented I/O

You're using Threads

You're using Threads (in degenerate form)

Exercises

Echo Server

- Listen on a port
- Respond to each request in a new Thread
- **Extra Credit**: Record stats on requests in a shared data structure

Connection Pool

- Allow N clients to access X shared instances of, say, Redis (where N > X)
- Clients “check out” a connection and get exclusive access
- Clients “check in” when done
- Instances get re-used

THANKS!

david.copeland@livingsocial.com

@davetron5000

www.naildrivin5.com

www.awesomecommandlineapps.com