Fibers Are the Right Solution

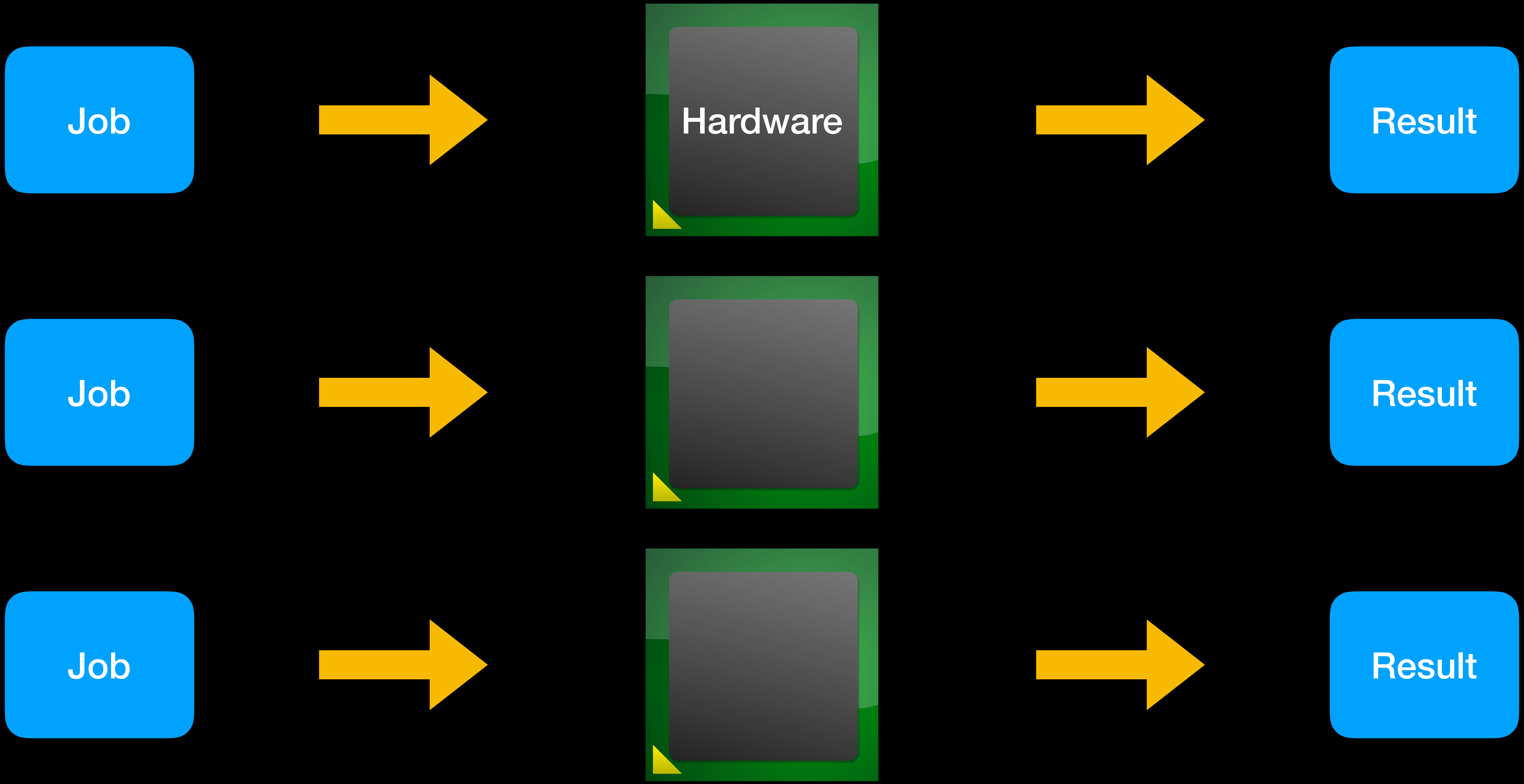
Samuel Williams
samuel@ruby-lang.org   www.codeotaku.com
@ioquatix

Fibers Are the Right Solution

Samuel Williams
samuel@ruby-lang.org www.codeotaku.com
@ioquatix

What is Scalability?

スケーラビリティとは何か?



Job

Hardware

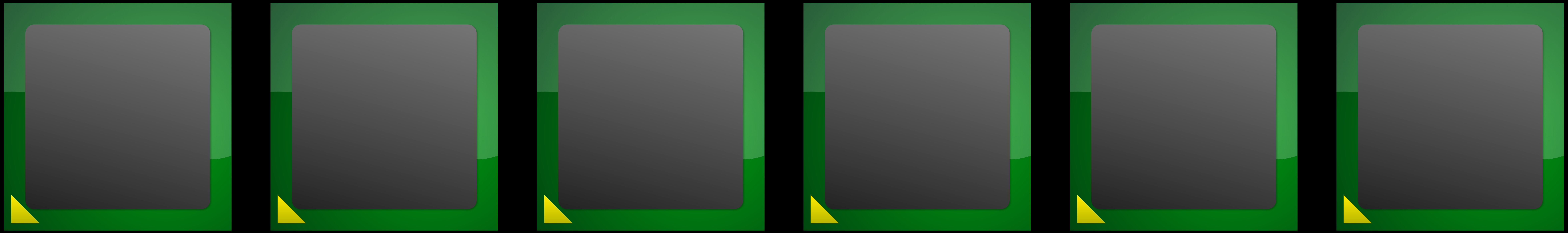
Result

Job

Result

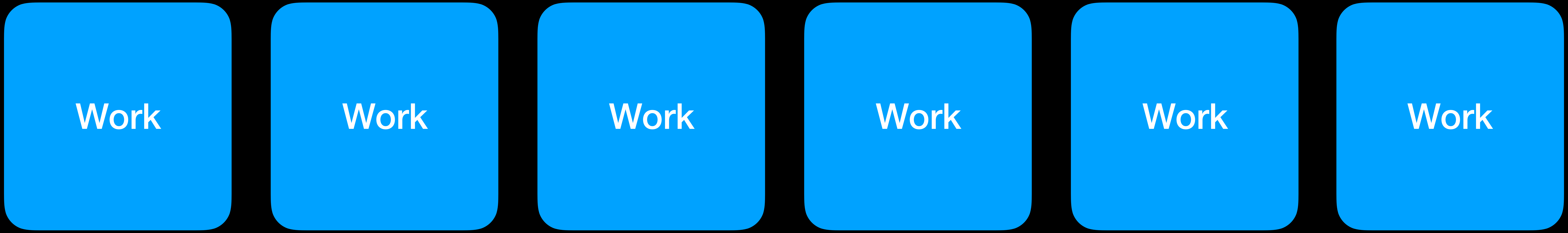
Job

Result



Proportional Improvement

比例的なパフォーマンス改善



Work

Work

Work

Work

Work

Work

Why is scalability important?

なぜスケーラビリティは大切なのか?

Photo from NASA.

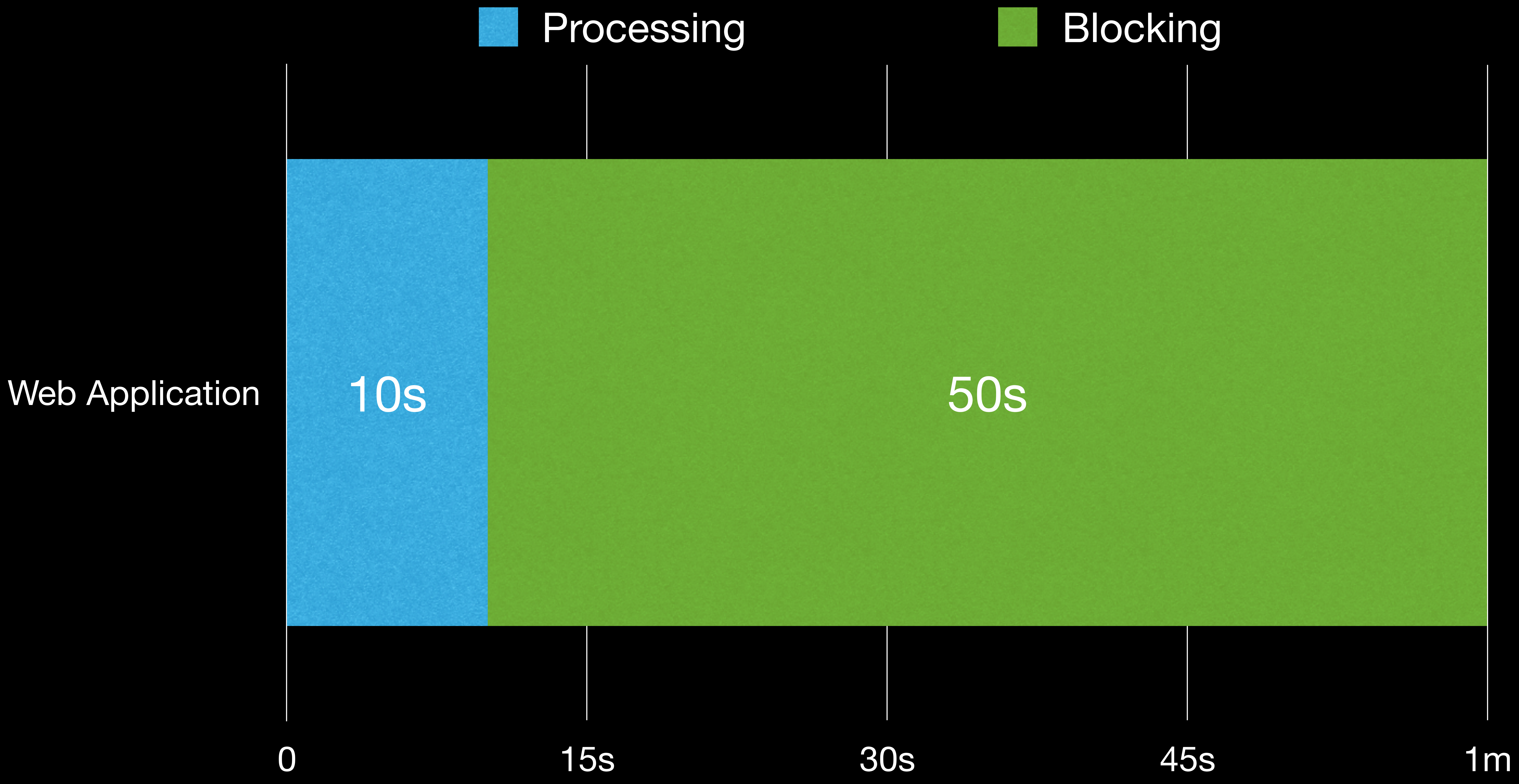
Is Ruby scalable?

Rubyはスケーラブルか?

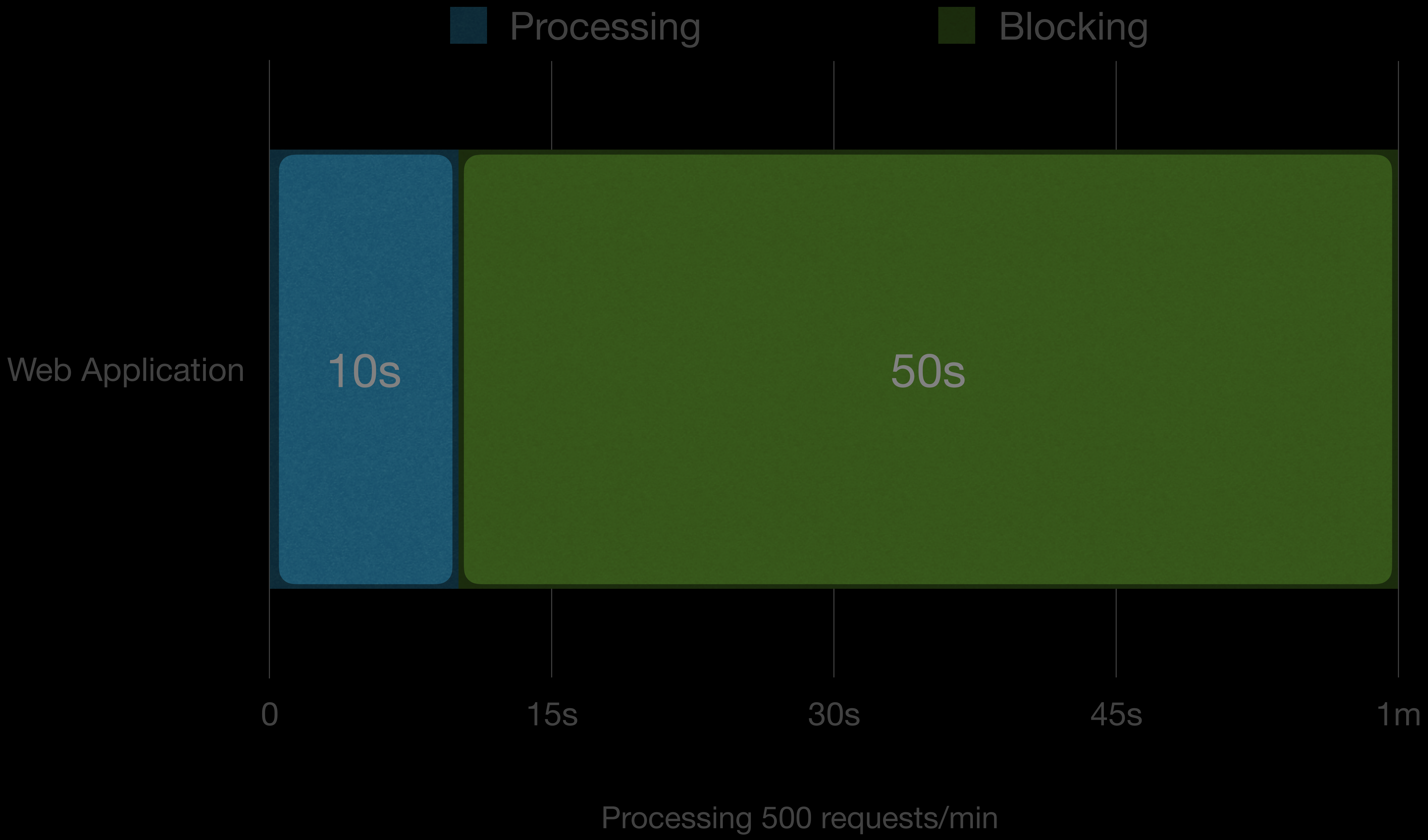
Over a million web sites globally.

Rubyが使われた世界中のWebサイトについて

Is this scalable?



Processing 500 requests/min


Processing
Blocking
Web Application
10s
50s
0
15s
30s
45s
1m
Processing 500 requests/min

Postgres

S3

Redis

WebSocket

SMTP

Blocking

MySQL

ブロッキング

Disk

DNS

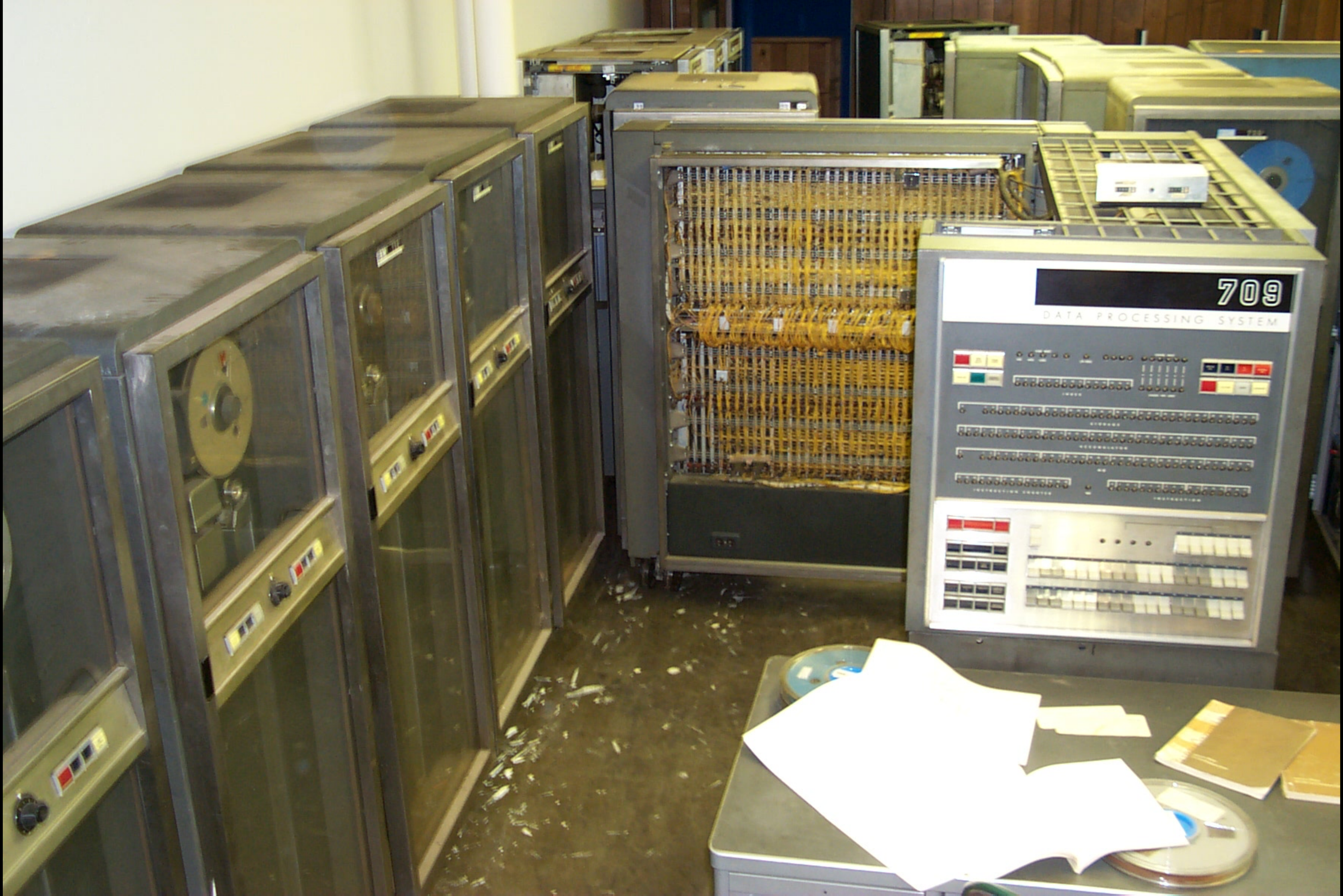
HTTP

How do we maximise hardware utilisation?

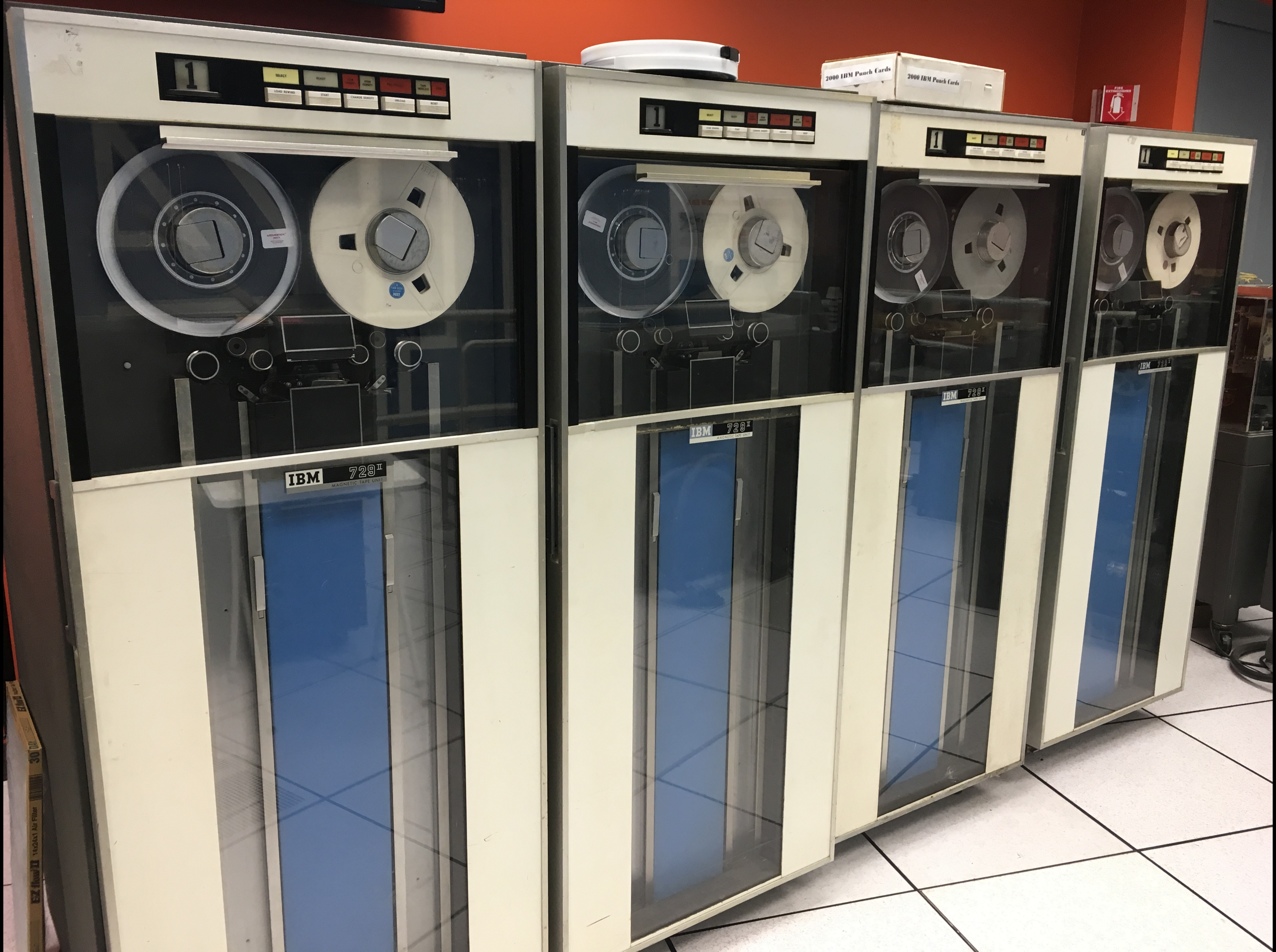
どのようにしてハードウェア最大限活用できるか?

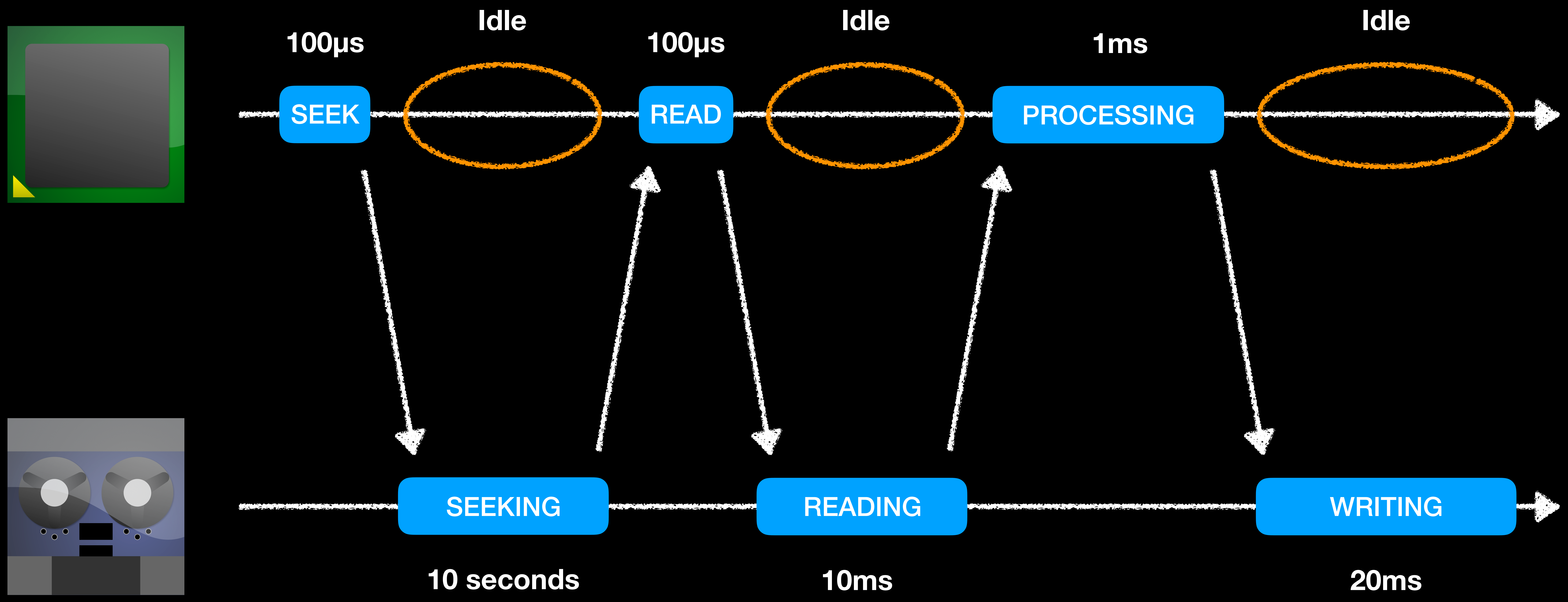
Late 1950s

1950年代後半

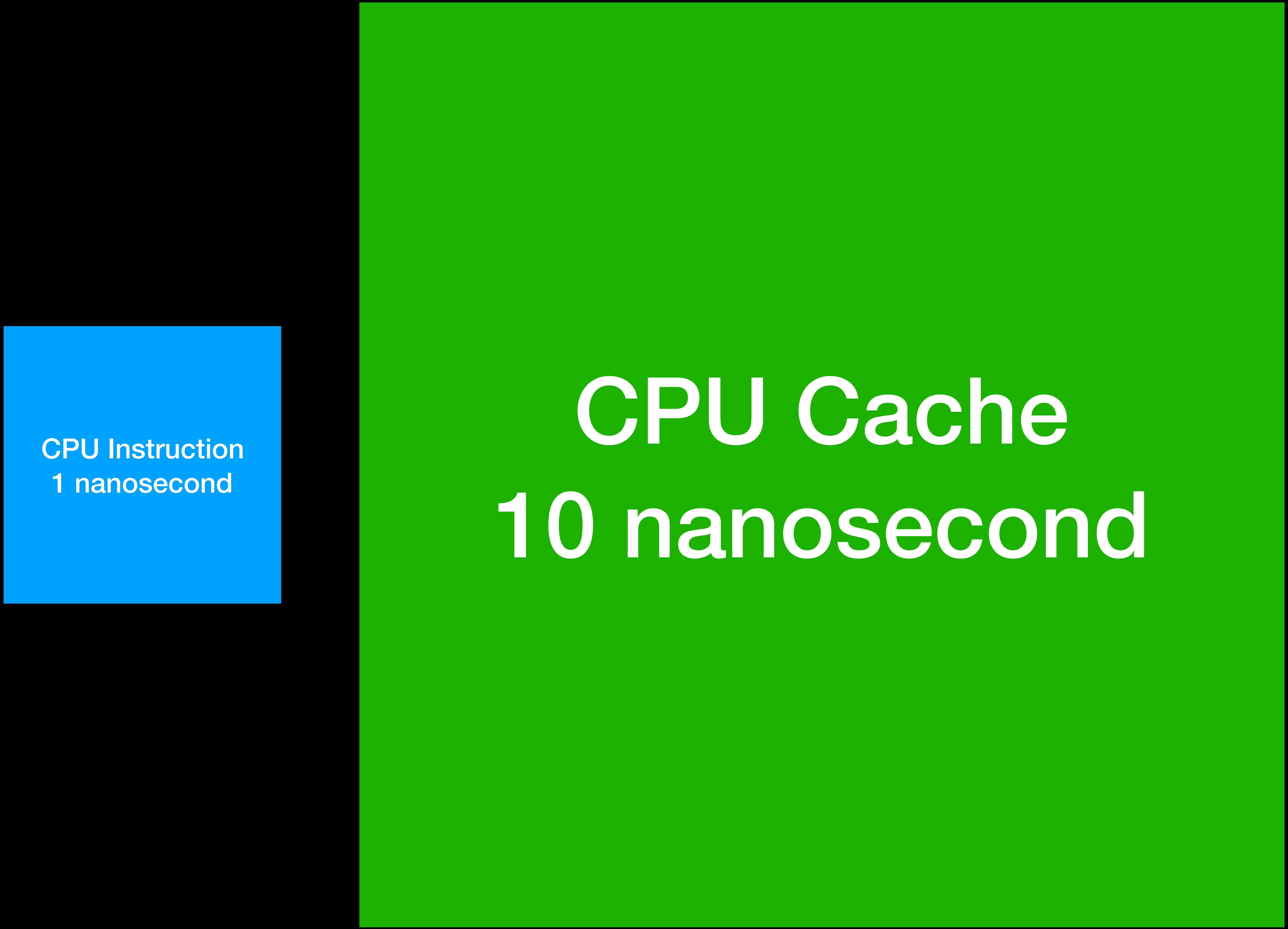


709
DATA PROCESSING SYSTEM


2000 IBM Punch Cards
1
SELECT
READY
LOW DENSITY
HIGH DENSITY
TAPE INDICATE
LOAD REWIND
START
CHANGE DENSITY
UNLOAD
RESET
IBM 729 II
MAGNETIC TAPE UNIT

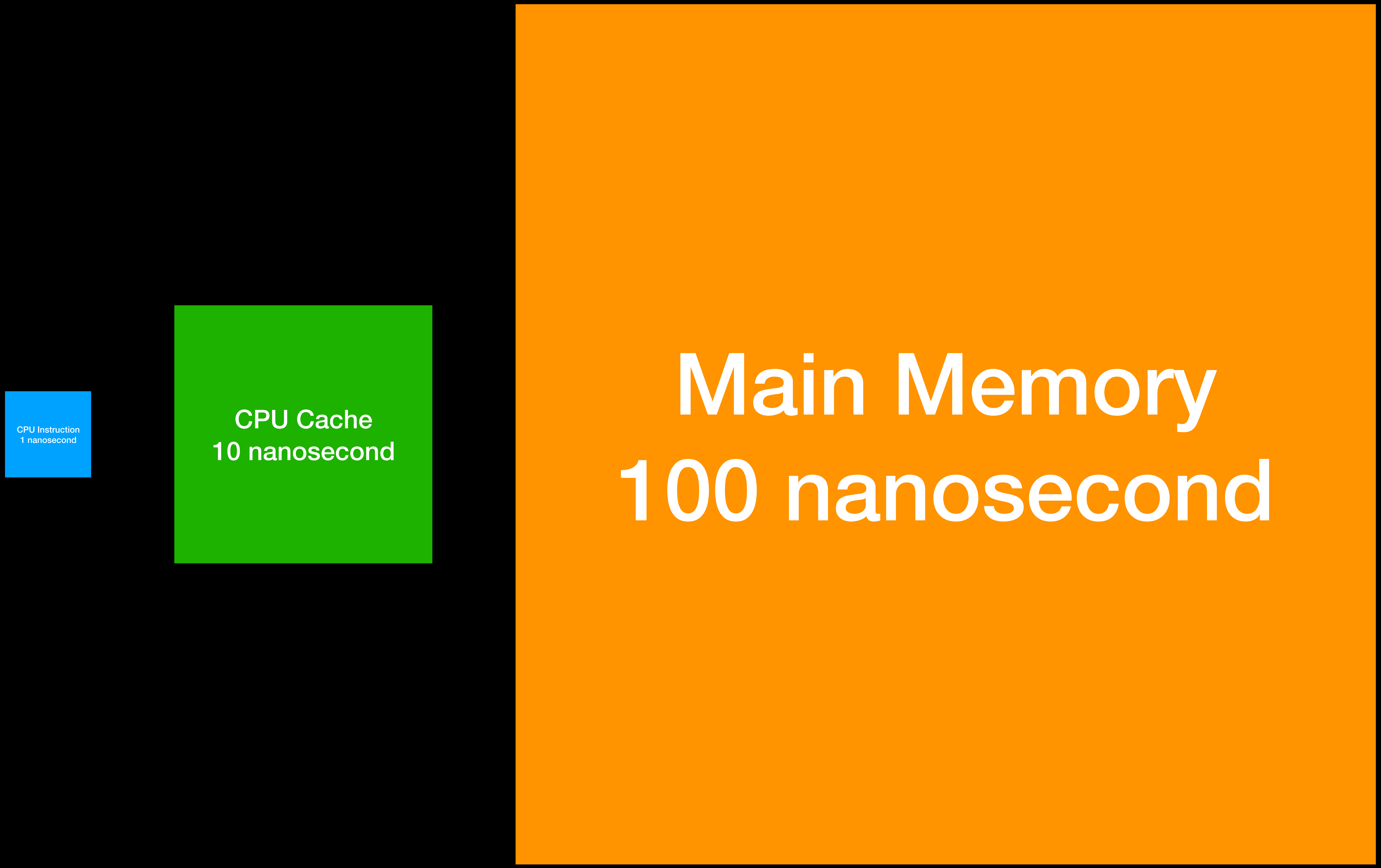

Idle
Idle
Idle
100µs
100µs
1ms
SEEK
READ
PROCESSING
SEEKING
READING
WRITING
10 seconds
10ms
20ms

CPU Instruction
1 nanosecond



CPU Instruction
1 nanosecond

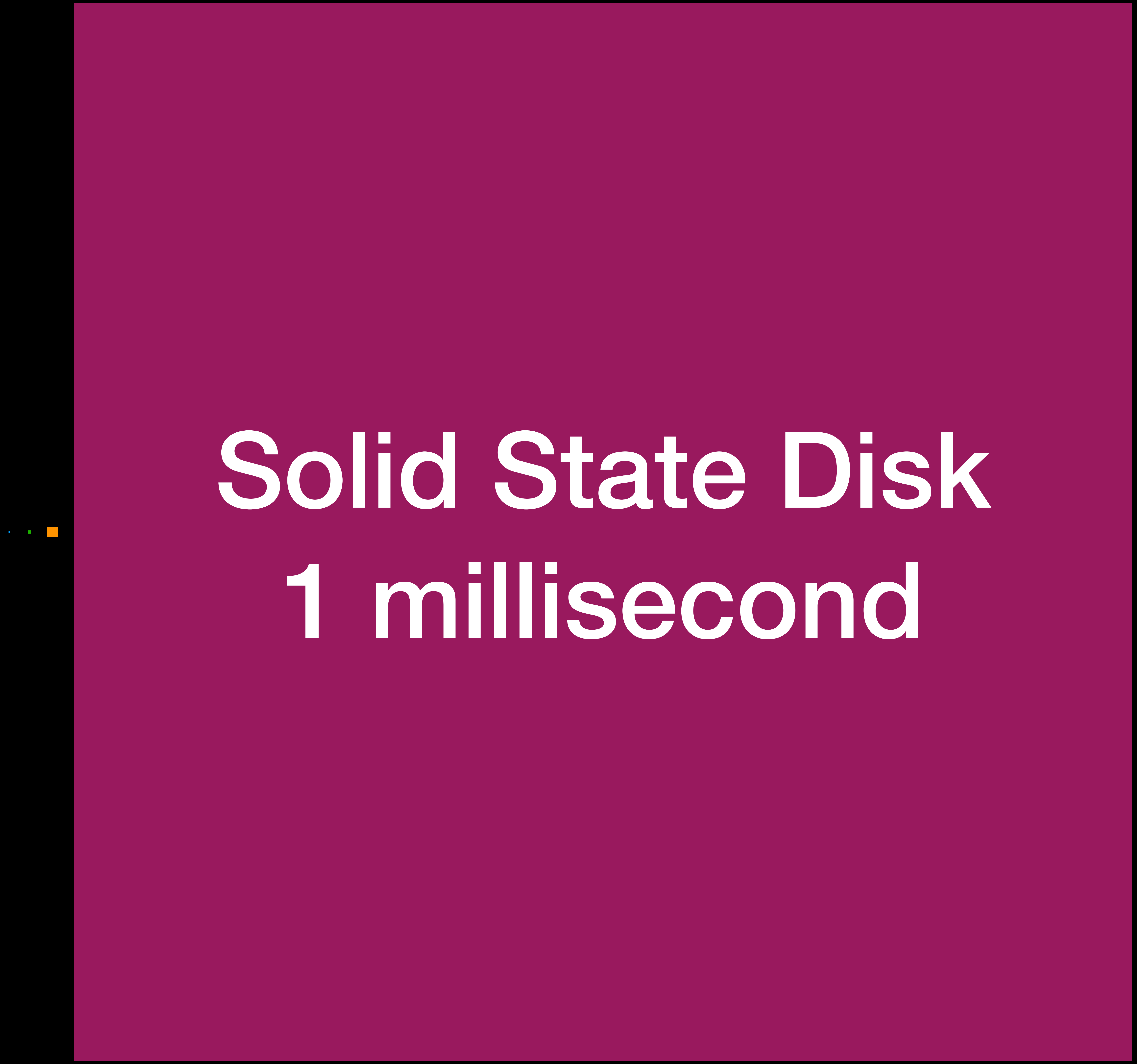
CPU Cache
10 nanosecond



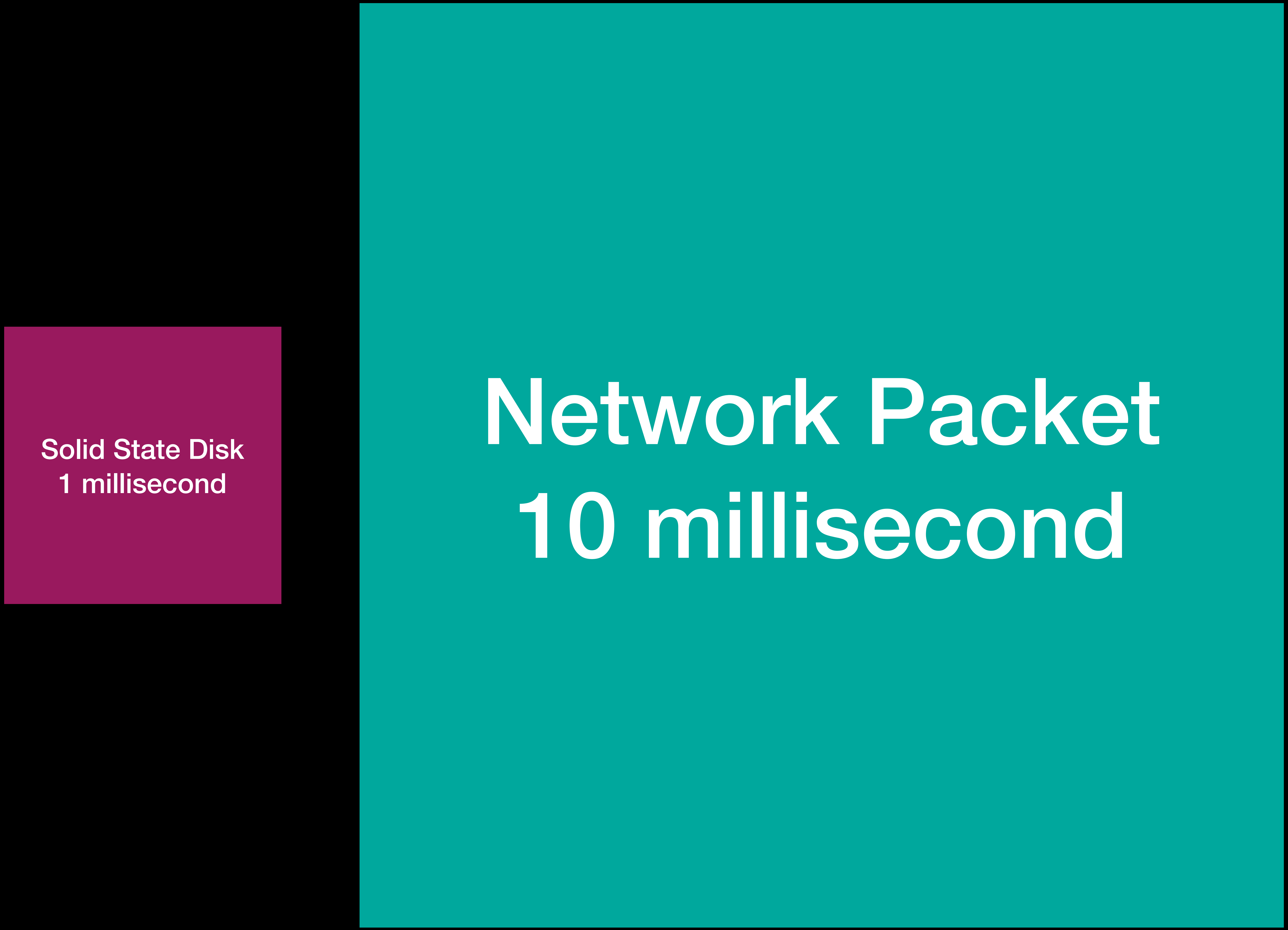
CPU Instruction
1 nanosecond

CPU Cache
10 nanosecond

Main Memory
100 nanosecond



Solid State Disk
1 millisecond

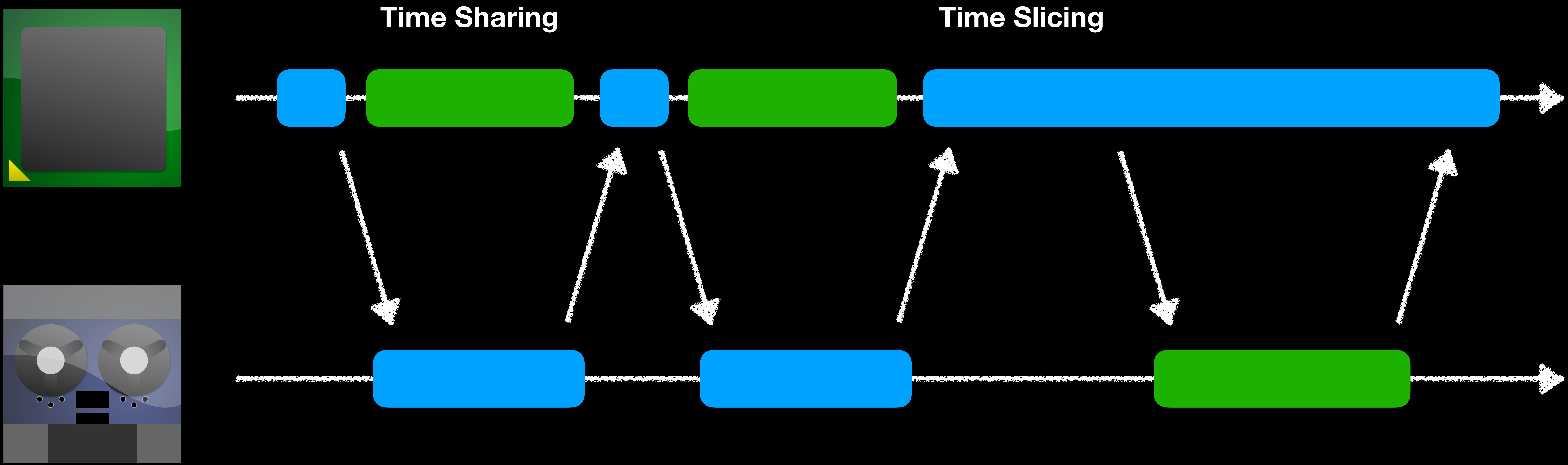


Solid State Disk
1 millisecond

Network Packet
10 millisecond

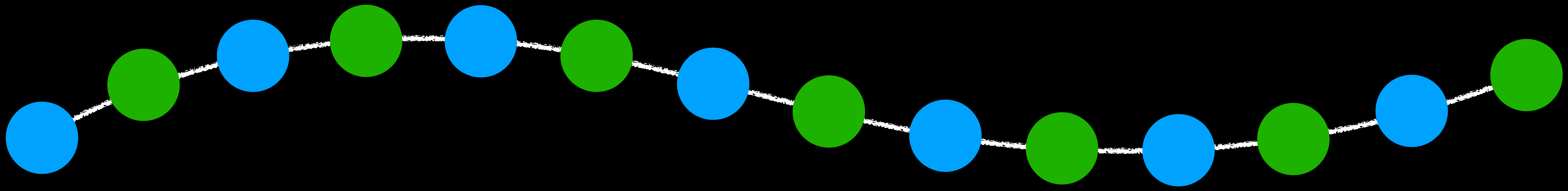
Can we avoid being idle?

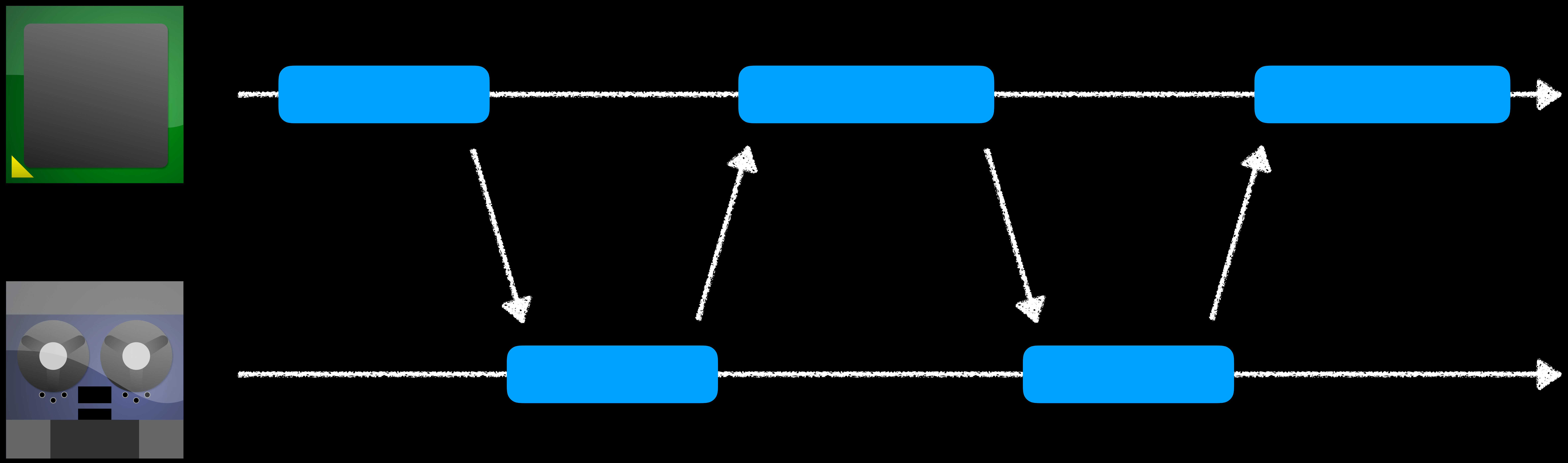
アイドル状態を避けることは可能か?

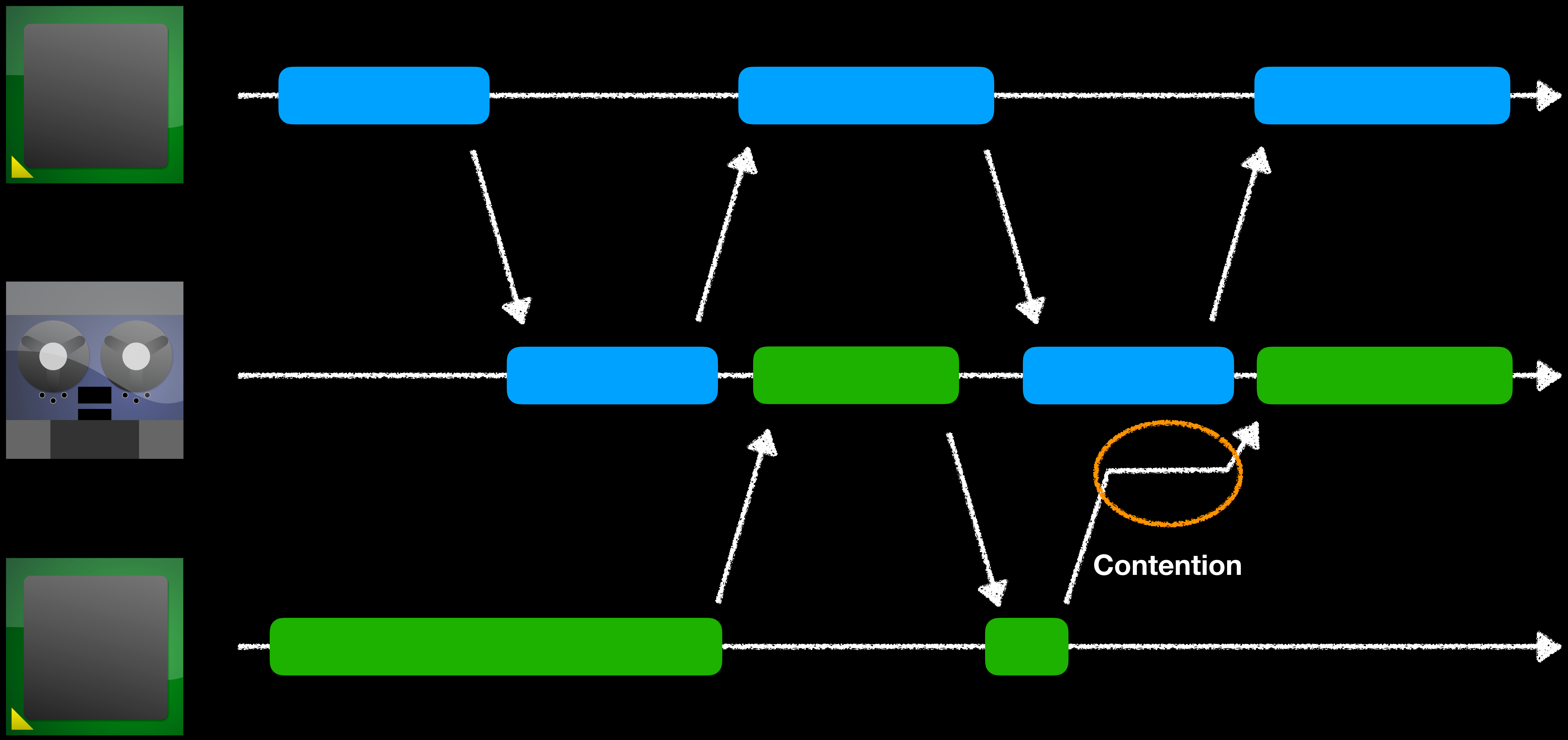

Time Sharing
Time Slicing

並行性

Concurrency

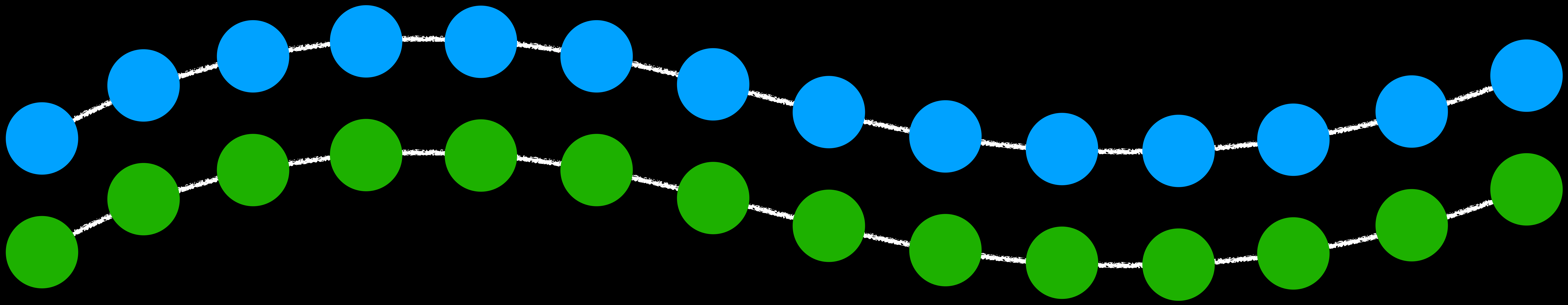





Contention

並列性

Parallelism



How does this apply to Ruby?

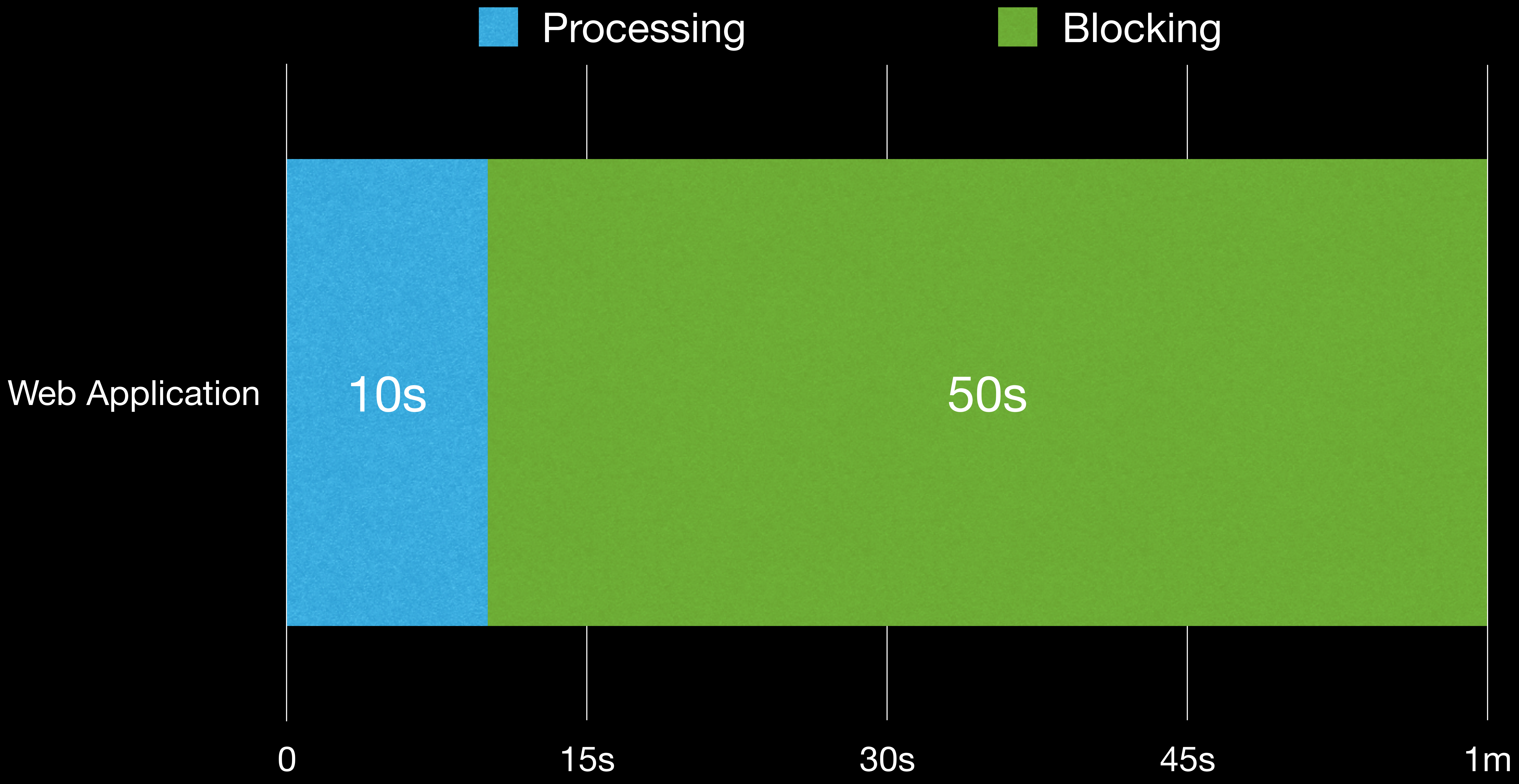
どのようにしてRubyに応用するか?

Is Ruby fast enough?

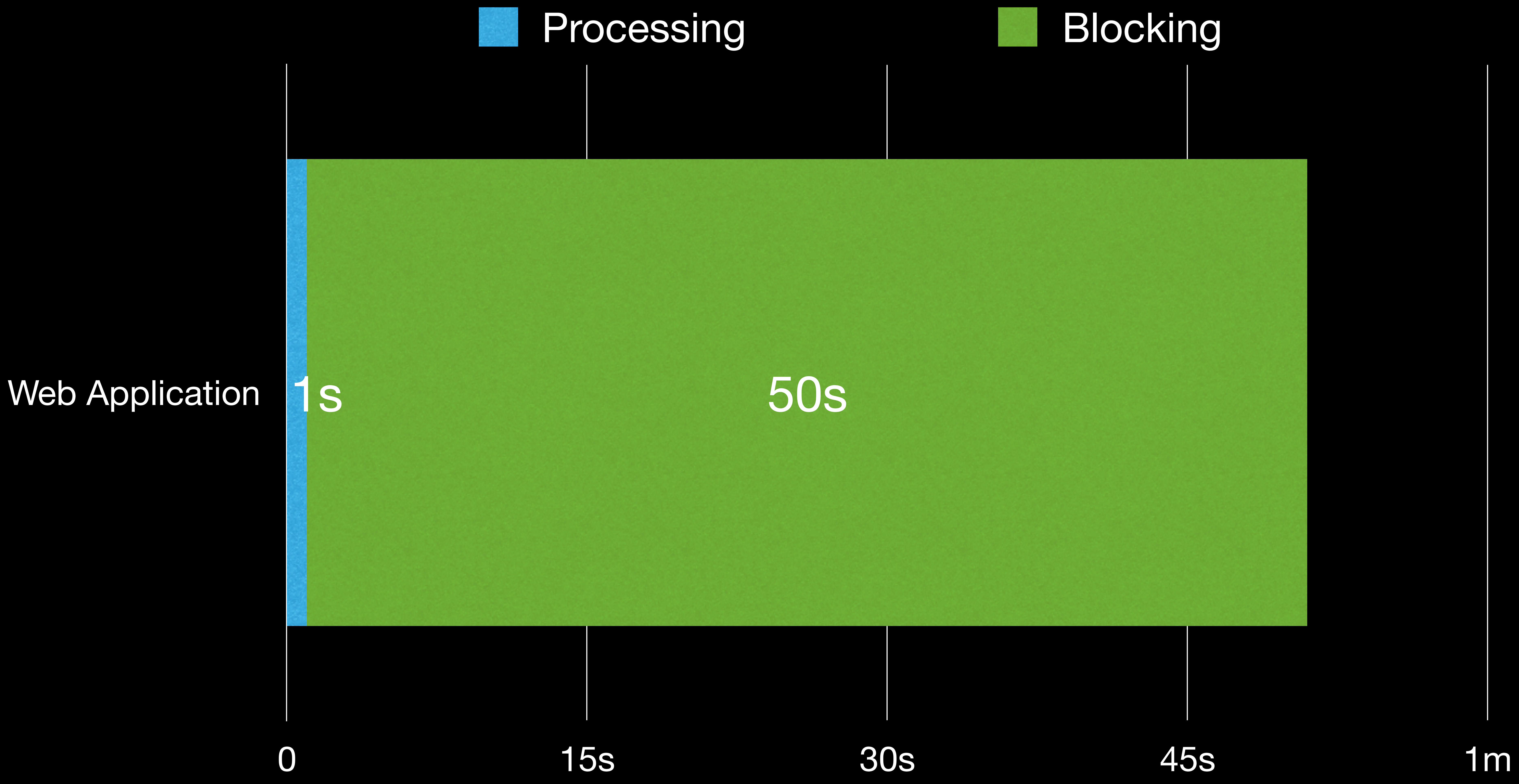
Rubyは十分早いか?

Let’s make Ruby 10x faster.

Rubyを10倍早くしよう

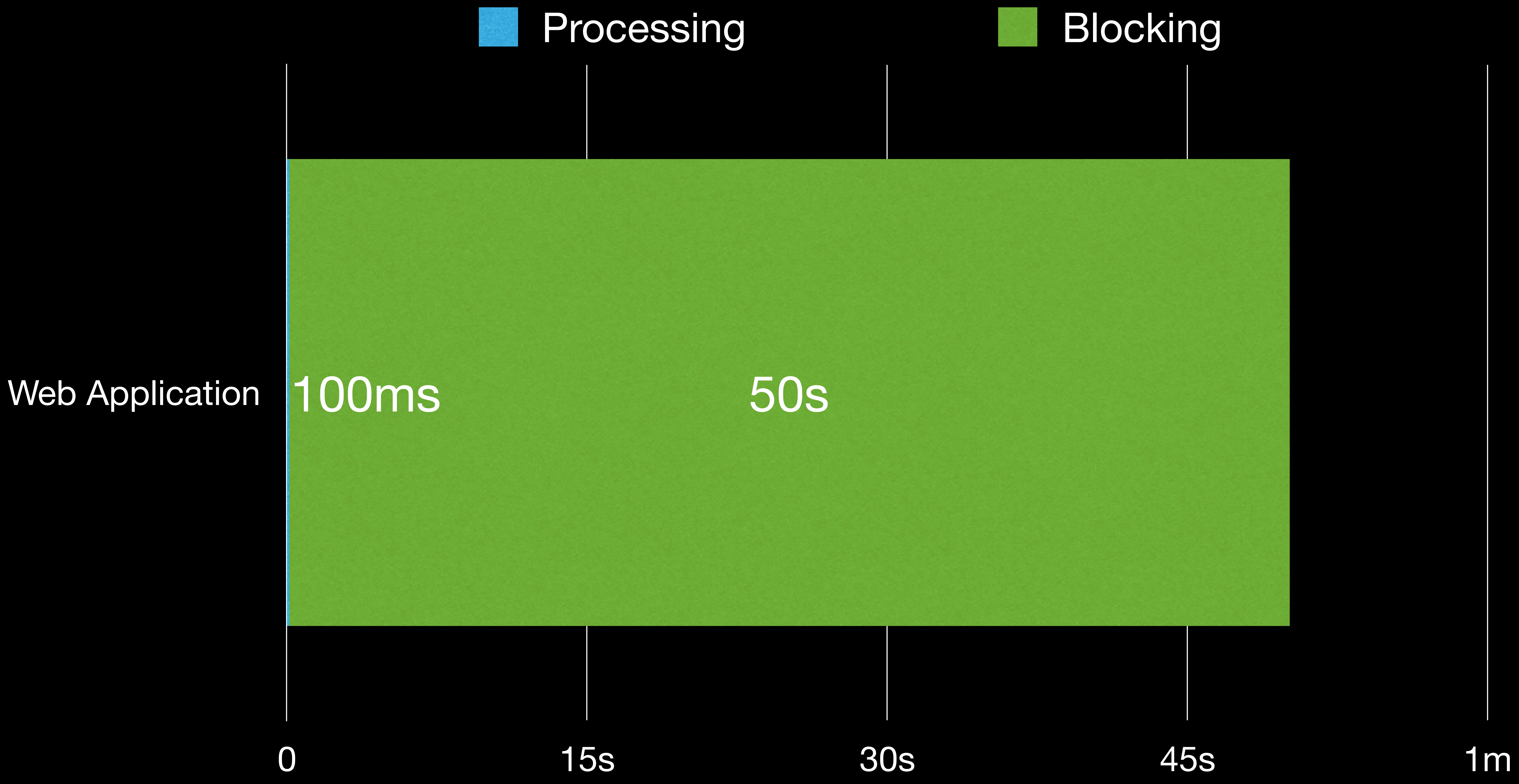


Processing
Blocking
Web Application
10s
50s
0
15s
30s
45s
1m


Processing
Blocking
Web Application
1s
50s
0
15s
30s
45s
1m

Let’s make Ruby 100x faster.

Rubyを100倍早くしよう


Processing
Blocking
Web Application
100ms
50s
0
15s
30s
45s
1m

How do we handle more requests?

どのようにしてもより多くのリクエストを処理するか?

Can we use multiple processes?

複数のプロセスを利用することはできるか?

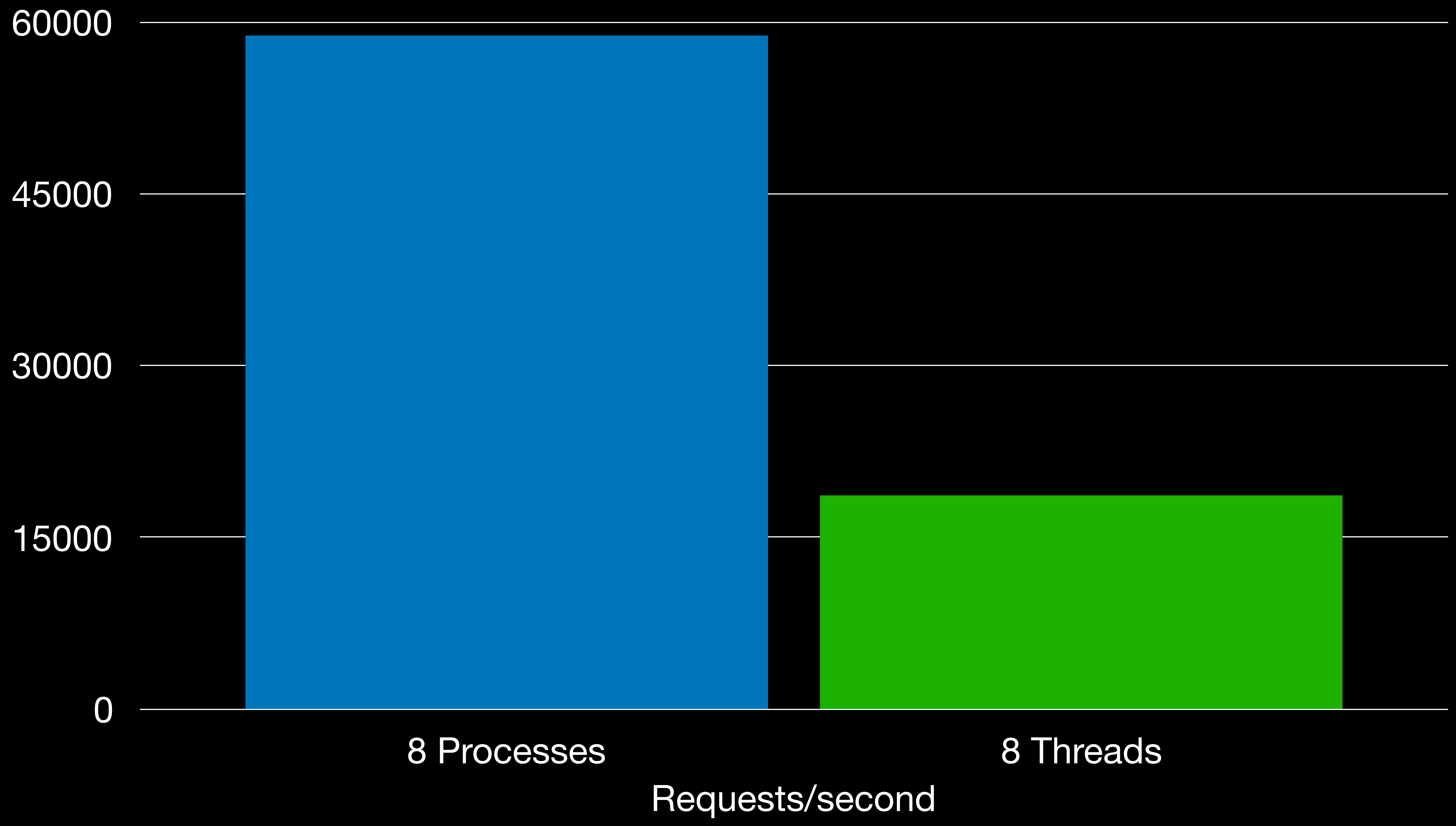
What about threads?

スレッドは?

How bad is the global interpreter lock?

グローバルインタプリタロックはどれほど悪いのか

Falcon “Hello World” Web Server



Are processes and threads sufficient?

プロセスとスレッドは十分か?

How many processes can we create?

いくつのプロセスを作ることができるか?

How many threads can we create?

いくつのスレッドを作ることができるか

100?

1,000?

10,000?

What about long running connections?

ロングランニングコネクションはどうするか?

What about 100,000 connected WebSockets?

100,000のWebSocketsは?

We need to go deeper

Event driven
non-blocking I/O

イベントドリブンのノンブロッキングI/O

```
while message = connection.read
  handle(message)
end
```

```
while message = connection.read
  handle(message)
end
```

```
while message = connection.read
  handle(message)
end
```

```
while true
  ready = IO.select(connections)
  ready.each{|connection| handle(connection.read)}
end
```

How do we handle user logic?

ユーザーロジックはどのようにして処理するか

```ruby
def remote_size(host, port)
  peer = TCPSocket.new(host, port)
  count = 0

  while buffer = peer.read(1024)
    count += buffer.bytesize
  end

  return count
ensure
  peer&.close
end

puts remote_size(HOST, PORT)
```

Sequential is easy.

シーケンシャルな処理は簡単

Callbacks…

```
def remote_size(host, port)
  TCPSocket.new(host, port) do |peer, error|
  end
end

remote_size(HOST, PORT) do |size, error|
  puts size
end
```

```ruby
def remote_size(host, port)
  TCPSocket.new(host, port) do |peer, error|
    if error
      yield nil, error
    else
      count = 0

      peer.read(1024) do |buffer, error|
      end
    end
  end
end
```

```
   yield nil, error
  else
   count = 0

   read_more = lambda do
    peer.read(1024) do |buffer, error|
     if error
      yield nil, error
     elsif buffer
      count += buffer.bytesize

      read_more.call
     else
      yield count, nil
     end
    end
   end

   read_more.call
  end
 end
end
```

```
while buffer = peer.read(1024)
  count += buffer.bytesize
end

return count
```

```
TCPSocket.new(host, port) do |peer, error|
  if error
    yield nil, error
  else
    count = 0

    read_more = lambda do
      peer.read(1024) do |buffer, error|
        if error
          yield nil, error
        elsif buffer
          count += buffer.bytesize

          read_more.call
        else
          yield count, nil
        end
      end
    end

    read_more.call
  end
end
```

```ruby
  yield nil, error
else
  count = 0

  read_more = lambda do
    peer.read(1024) do |buffer, error|
      if error
        peer.close
        yield nil, error
      elsif buffer
        count += buffer.bytesize

        read_more.call
      else
        peer.close
        yield count, nil
      end
    end
  end

  read_more.call
end
```

Callback Hell

```
if error
  yield nil, error
else
  count = 0

  read_more = lambda do
    peer.read(1024) do |buffer, error|
      if error
        peer.close
        yield nil, error
      elsif buffer
        count += buffer.bytesize

        read_more.call
      else
        peer.close
        yield count, nil
      end
    end
  end

  read_more.call
end
```

Async/Await…

```ruby
async def remote_size(host, port)
  peer = await TCPSocket.new(host, port)
  count = 0

  while buffer = await peer.read(1024)
    count += buffer.bytesize
  end

  return count
ensure
  peer&.close
end

async lambda do
  puts await remote_size(HOST, PORT)
end.call
```

```
async lambda do
  puts(await remote_size(HOST, PORT))
end.call
```

```
async lambda do
  await puts(await remote_size(HOST, PORT))
end.call
```

Async/Await Hell

```
async def remote_size(host, port)
  peer = await TCPSocket.new(host, port)
  count = 0

  while buffer = await peer.read(1024)
    count += buffer.bytesize
  end

  return count
ensure
  peer&.close
end

async lambda do
  await puts(await remote_size(HOST, PORT))
end.call
```

Can we do better?

改善できるか

Should we rewrite existing code?

既存のコードを書き直すべきか?

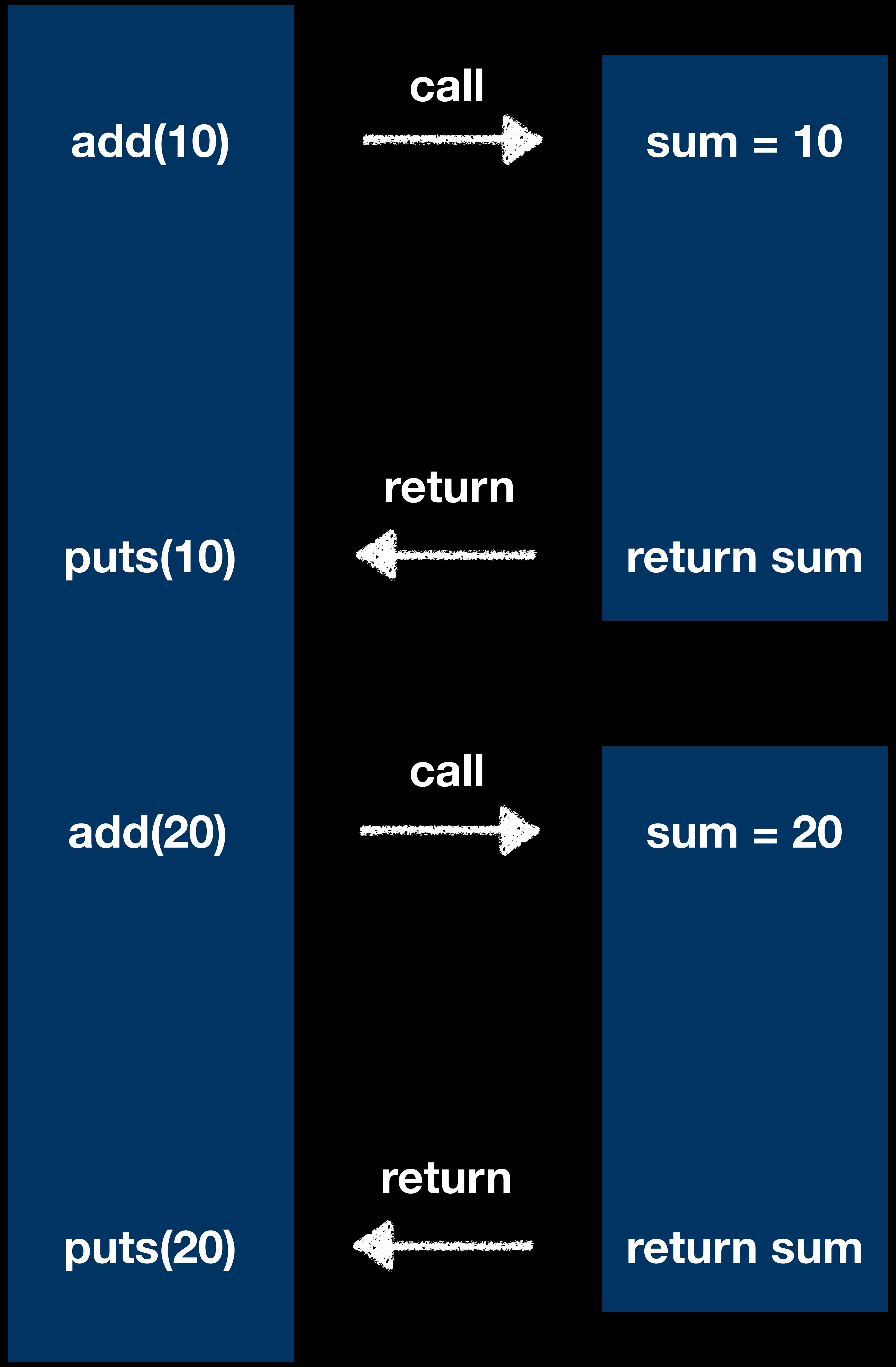
What about using fibers?

ファイバーを使うのはどうか

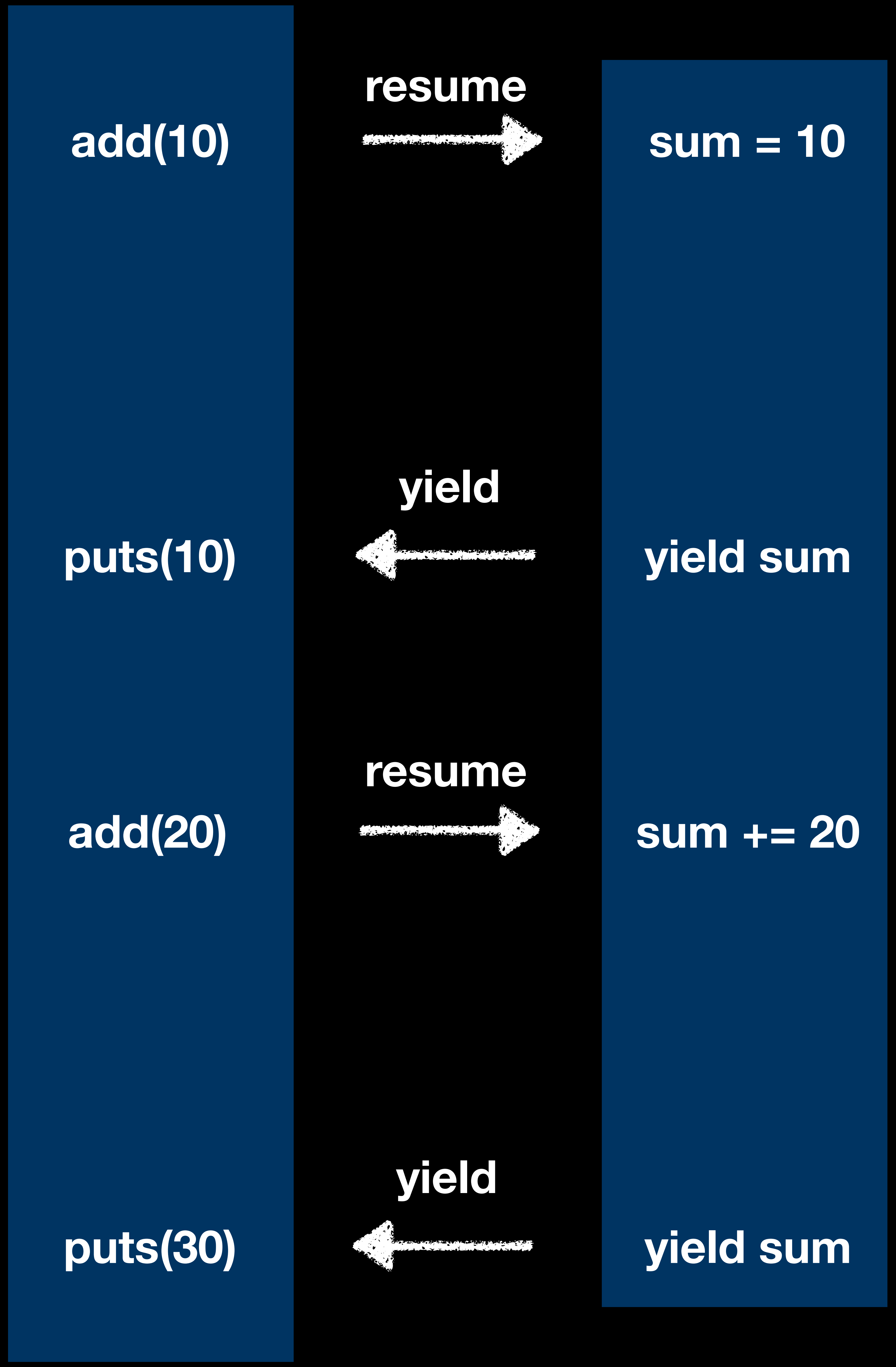
What are fibers?

ファイバーとは何か

```ruby
def add(sum)
  return sum
end

puts add(10) # => 10
puts add(20) # => 20
```



```
add = Fiber.new do |sum|
  while true
    sum += Fiber.yield(sum)
  end
end

puts add.resume(10) # => 10
puts add.resume(20) # => 30
```

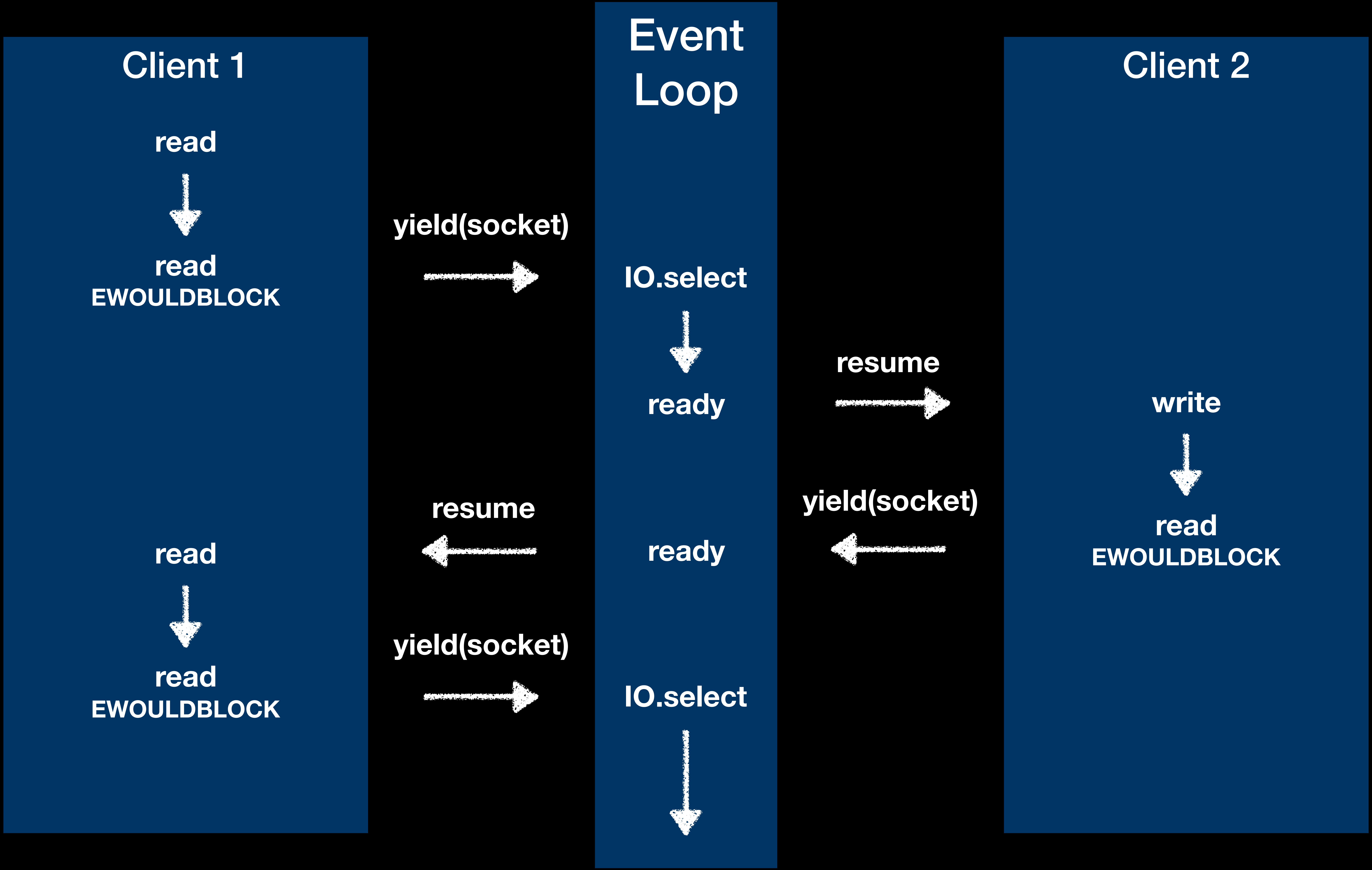


Stack and instruction pointer is not lost.

スタックとインストラクションポインタはなくなってはいない

How do we use fibers for blocking I/O?

どのようにしてファイバーをブロッキングI/Oに使うか



Client 1

read

read
EWOULDBLOCK

read

read
EWOULDBLOCK

yield(socket)

resume

yield(socket)

Event Loop

IO.select

ready

ready

IO.select

resume

yield(socket)

Client 2

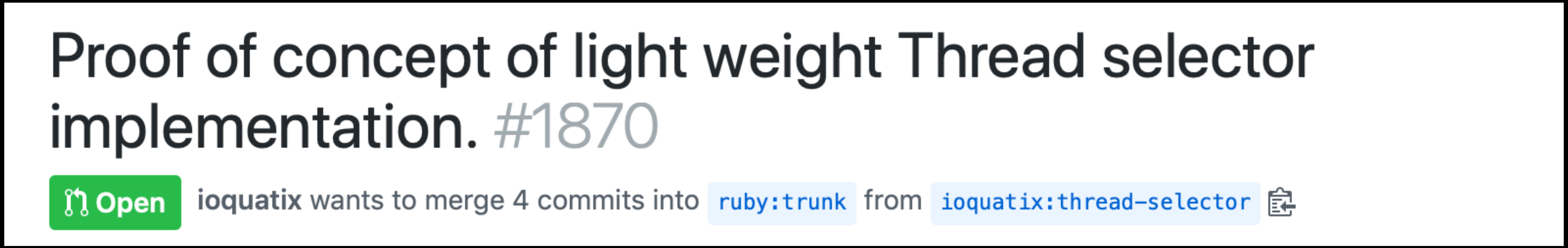
write

read
EWOULDBLOCK

```ruby
def remote_size(*address)
  Async do
    peer = Async::IO::TCPSocket.new(*address)
    count = 0

    while buffer = peer.read(1024)
      count += buffer.bytesize
    end

    return count
  ensure
    peer&.close
  end.wait
end

puts remote_size(HOST, PORT)
```

How do we make existing code scalable?

どのようにして既存のコードをスケーラブルにするか

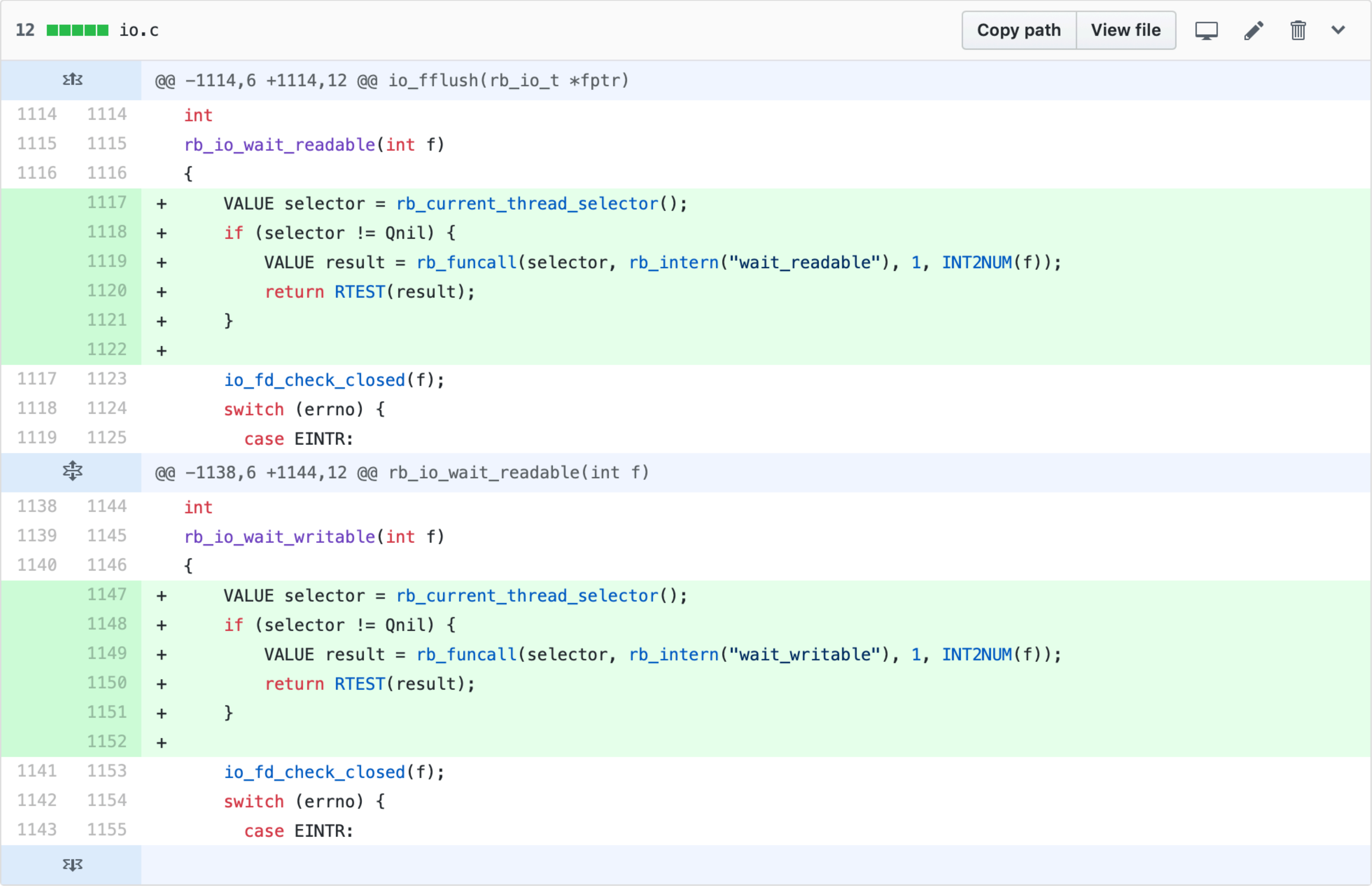


Proof of concept of light weight Thread selector implementation. #1870

Open ioquatix wants to merge 4 commits into `ruby:trunk` from `ioquatix:thread-selector`

Transparently make all I/O non-blocking

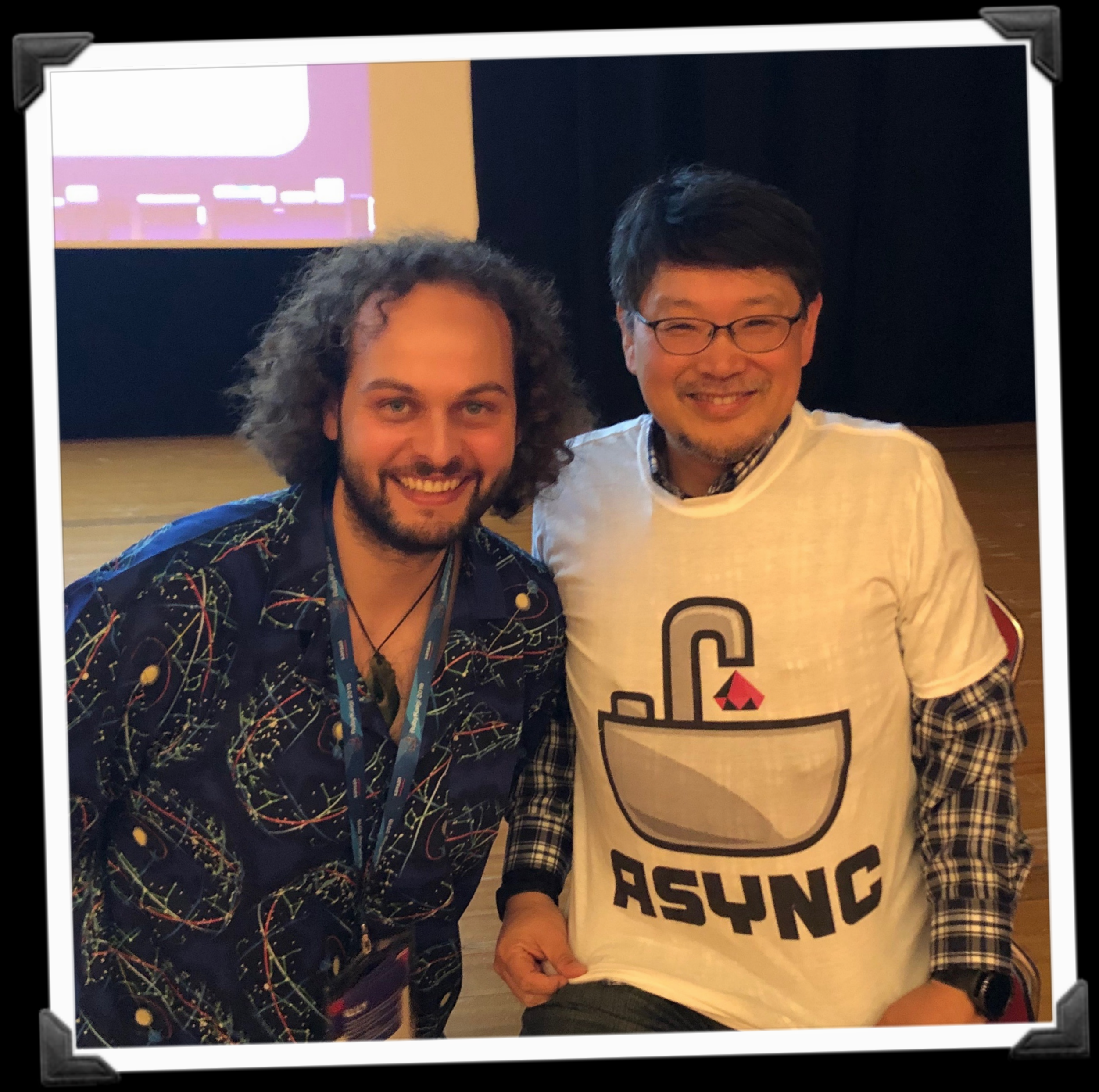
コードを変更することなく全てのI/Oをノンブロッキングにする



```diff
@@ -1114,6 +1114,12 @@ io_fflush(rb_io_t *fptr)
 int
 rb_io_wait_readable(int f)
 {
+  VALUE selector = rb_current_thread_selector();
+  if (selector != Qnil) {
+    VALUE result = rb_funcall(selector, rb_intern("wait_readable"), 1, INT2NUM(f));
+    return RTEST(result);
+  }
+
   io_fd_check_closed(f);
   switch (errno) {
    case EINTR:
@@ -1138,6 +1144,12 @@ rb_io_wait_readable(int f)
 int
 rb_io_wait_writable(int f)
 {
+  VALUE selector = rb_current_thread_selector();
+  if (selector != Qnil) {
+    VALUE result = rb_funcall(selector, rb_intern("wait_writable"), 1, INT2NUM(f));
+    return RTEST(result);
+  }
+
   io_fd_check_closed(f);
   switch (errno) {
    case EINTR:
```

```ruby
    thread = Thread.new do
      selector = Selector.new
      Thread.current.selector = selector

      i, o = IO.pipe
      i.nonblock = true
      o.nonblock = true
      e = i.to_enum(:each_char)

      Fiber.new do
        o.write("Hello World")
        o.close
      end.resume

      Fiber.new do
        while c = (e.next rescue nil)
          message << c
        end
      end.resume

      selector.run
    end
```

```ruby
  def run
    while @readable.any? or @writable.any?
      readable, writable = IO.select(@readable.keys, @writable.keys, [])

      readable.each do |io|
        @readable[io].transfer
      end

      writable.each do |io|
        @writable[io].transfer
      end
    end
  end

  def wait_readable(fd)
    io = IO.for_fd(fd)

    @readable[io] = Fiber.current

    @fiber.transfer

    @readable.delete(io)

    return true
  end
```



https://github.com/socketry/async

Bryan Powell
bryanp

Block or report user

> “Async is the right model because web apps are almost always I/O bound. The Ruby web ecosystem is really lacking in scalability (e.g. WebSockets on Puma). Async unlocks the next tier of scalability in the most Ruby-like way possible.”

Bryan Powell, on migrating from Puma to Falcon.

https://github.com/socketry/falcon

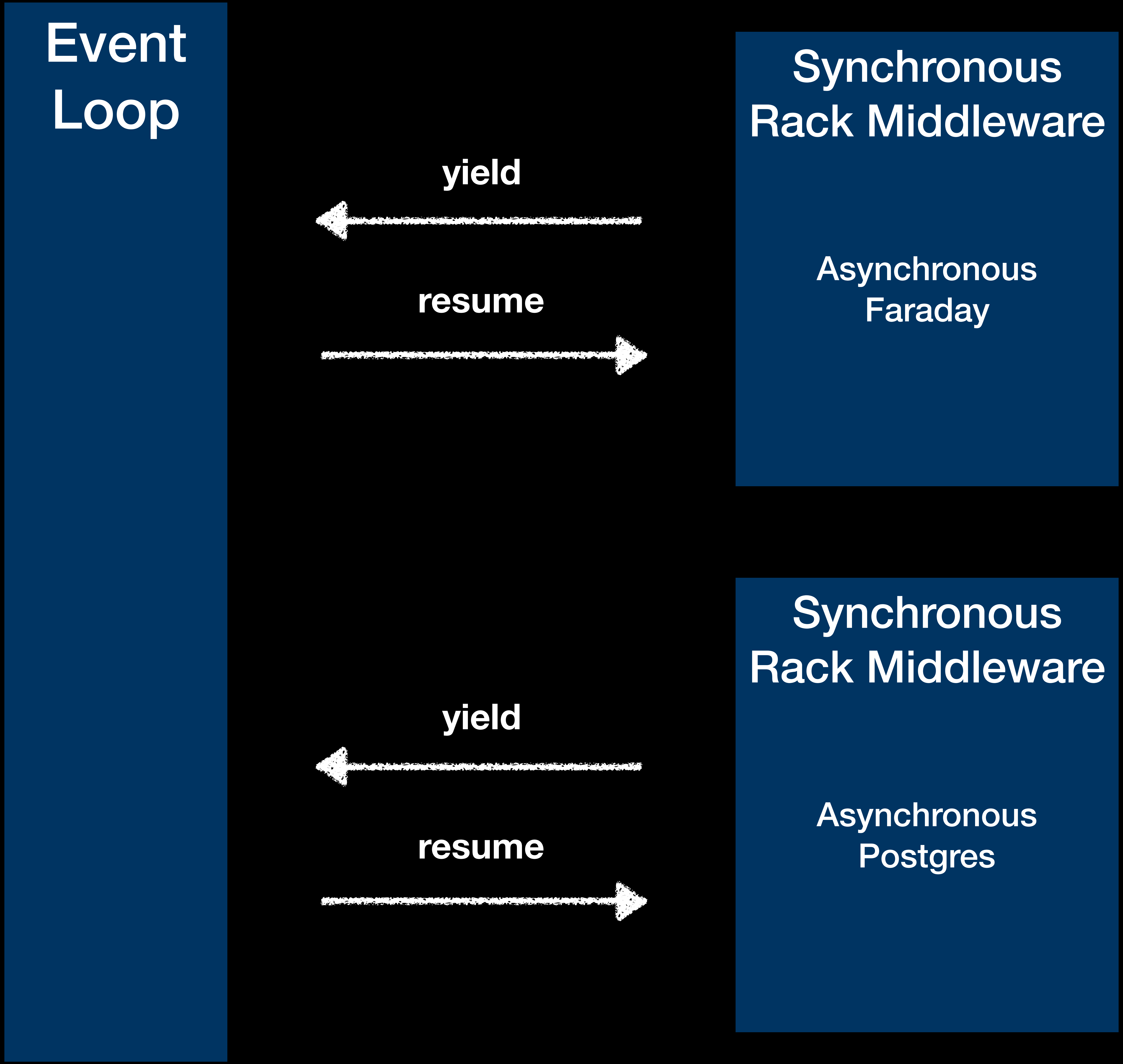
Multi-process

Multi-thread

Multi-fiber

HTTP/1
HTTP/2 & TLS

WebSockets



Event Loop

yield

resume

yield

resume

Synchronous Rack Middleware

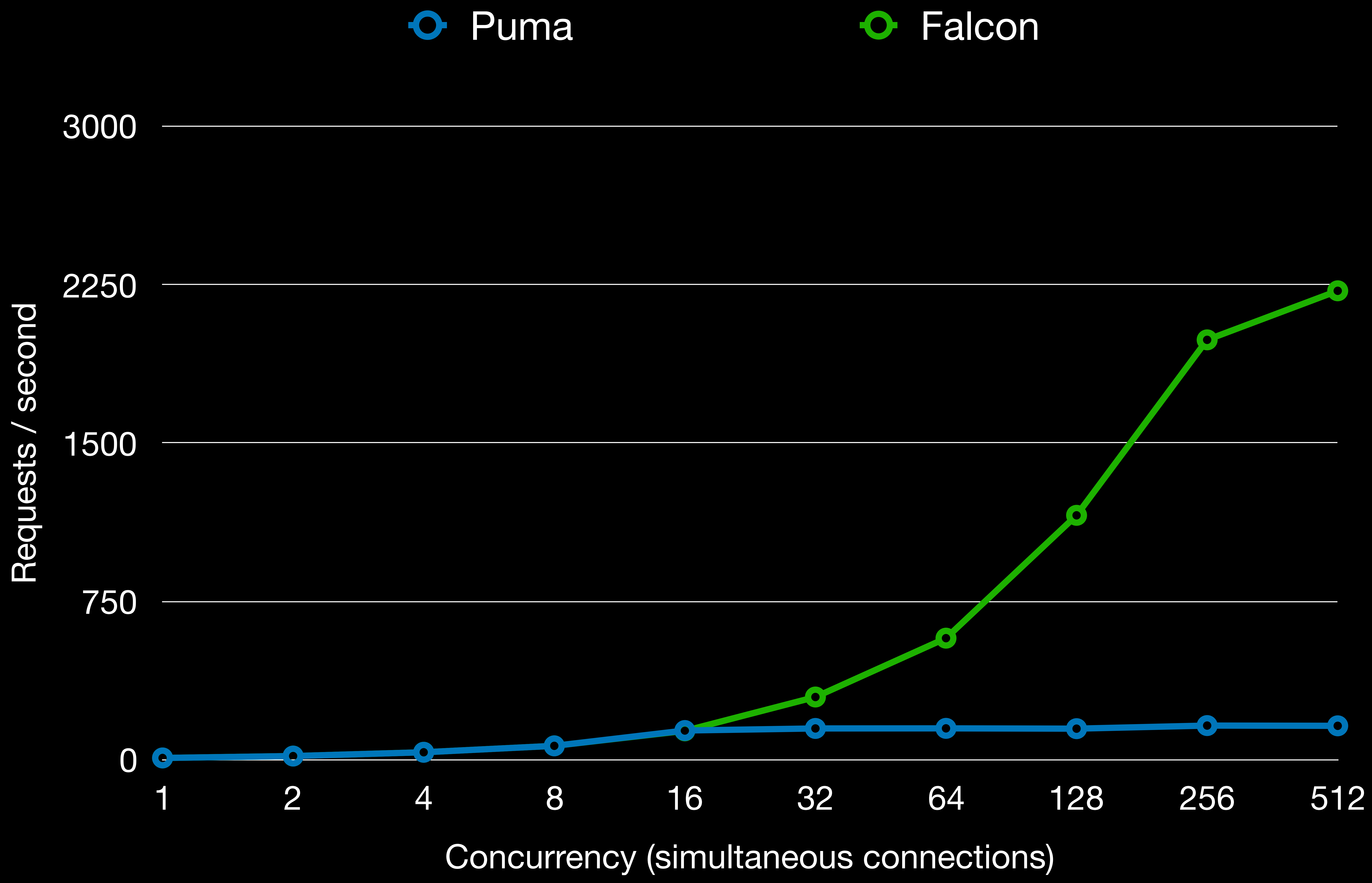
Asynchronous Faraday

Synchronous Rack Middleware

Asynchronous Postgres

How does it perform?

パフォーマンスはどうか


Puma
Falcon
Requests / second
3000
2250
1500
750
0
1
2
4
8
16
32
64
128
256
512
Concurrency (simultaneous connections)

Are fibers the right solution?

ファイバーが正しい選択か

Fibers scale better than threads.

スレッドよりもファイバーの方がスケールする

Fibers are easier than threads.

スレッドよりもファイバーの方が簡単

Fibers improve the scalability of existing code.

ファイバーは既存コードのスケーラビリティを改善する

S3

Postgres

WebSocket

Redis

Fibers Are the Right Solution

MySQL

SMTP

ファイバーが正しい選択

HTTP

DNS

Disk

Fibers Are the Right Solution