RUBY 3 CONCURRENCY

#SFRUBY | @GITHUB | @KIGSTER

KONSTANTIN GREDESKOUL

AUGUST 2024

WHO AM I? ERROR

- Life-long software engineer
- Has been doing web tech since 1998
- Ex-CTO of a social network Wanelo
- Rubyist since 2007
- 50 Gems with 140M+ downloads
- Self-taught programmer, with a Mathematics degree

CORE CONCEPTS

KONSTANTIN GREDESKOUL

AUGUST 2024

CONCURRENCY VS PARALLELISM

CONCURRENCY

PARALLEISM

Concurrency is the ability of a system to handle multiple tasks at the same time, or the interleaving of tasks to make progress on several tasks simultaneously. However, these tasks may not necessarily be running at the exact same time.

KONSTANTIN GREDESKOUL

AUGUST 2024

CONCURRENCY VS PARALLELISM

CONCURRENCY

PARALLEISM

Concurrency is our PERCEPTION of multi-tasking...

even when it's not true parallelism, it can still be useful when some tasks are waiting on io: network sockets & file io

KONSTANTIN GREDESKOUL

AUGUST 2024

CONCURRENCY VS PARALLELISM

CONCURRENCY

PARALLEISM

Parallelism is the simultaneous execution of multiple tasks at exactly the same time. This requires multiple processing units (like multiple CPU cores), where each core handles a separate task.

Parallelism is about doing lots of things at once, with tasks truly running in parallel. This is possible on multi-core processors where different tasks can be executed simultaneously on different cores.

KONSTANTIN GREDESKOUL

AUGUST 2024

BASIC TERMINOLOGY

KONSTANTIN GREDESKOUL

AUGUST 2024

CORE CONCEPTS

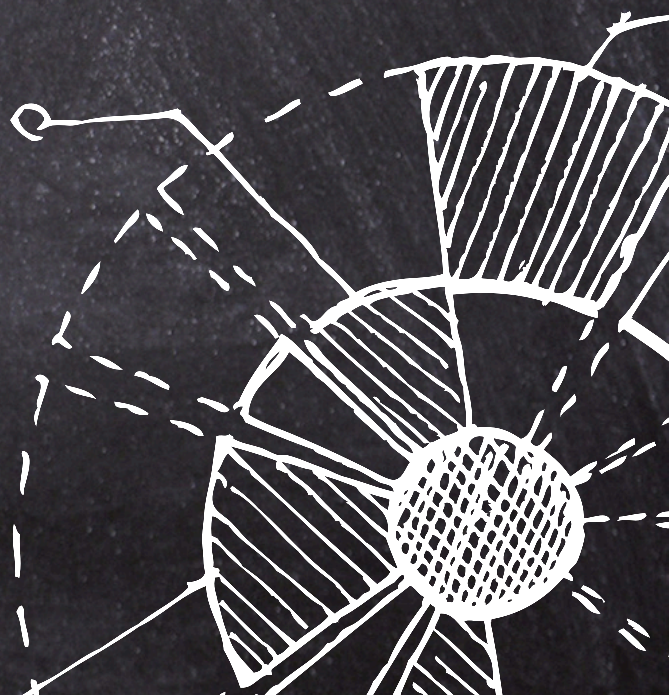
UNIX PROCESS

A UNIX process is an instance of a running program.
It consists of: program's code, data, a set of resources required (files opened, connections)
A process is started, starts running, can be waiting / paused, then exited, or killed by the Kernel.

THREAD

A thread within a program is the smallest unit of execution that can flow independently of other threads. It represents a single sequence of instructions or a "path of execution" within a program. Threads can share memory (although they must protect writes to shared memory using locks called Mutexes).

Threads make efficient use of the system's resources, especially in scenarios that involve waiting for I/O operations.



CORE CONCEPTS

GREEN THREADS

A green thread is a type of thread that is managed by a runtime library or virtual machine (VM) instead of natively by the operating system's kernel. These threads are called "green" because they are not true native threads; instead, they are implemented in user space, meaning that their scheduling and management are handled by the application's runtime environment.

Early JVM had green threads until they replaced them with native threads in 1.3.

NATIVE THREADS

A native thread in a UNIX operating system is a lightweight unit of execution that is managed directly by the operating system's kernel. Native threads, often referred to as kernel threads, allow a program to perform multiple tasks concurrently within a single process. This is why they are often called “Lightweight Processes” or LWPs.

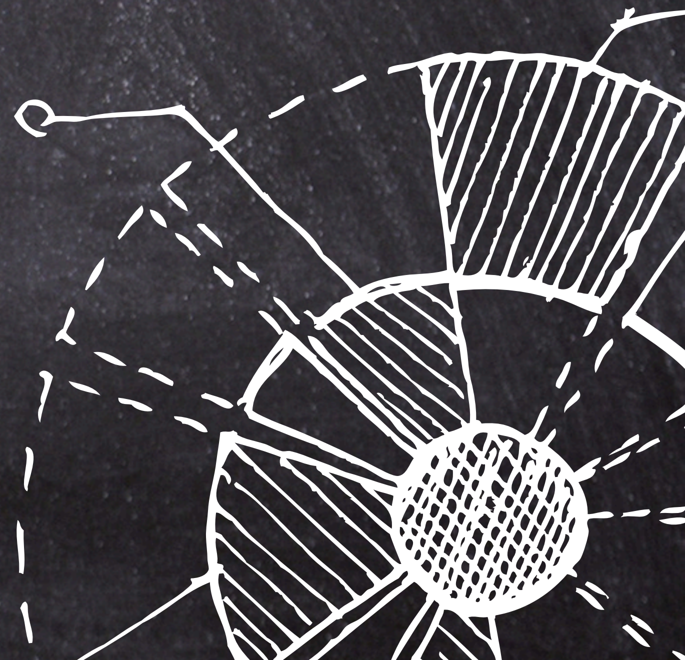
MULTI-THREADED

SINGLE PROCESS, MULTIPLE THREADS

Within any UNIX process there could be any number of native kernel threads (depending on what language was used to write the executable command)

A single Ruby 2 process always ran on a single logical CPU core and could only truly perform one CPU-bound task at any given time, regardless of the number of threads.

But if a Thread had to wait on IO (network, or file IO), then other threads could take over the CPU core, thus better utilizing the available CPU resources.



MULTI-PROCESS

SERVER MANAGER + WORKERS (VIA FORK)

Puma HTTP Server:
in the Cluster Mode starts one master process, which then forks multiple workers

SIDEKIQ Job Processing Framework (Enterprise Edition):
Provides a sidekiq cluster control process that manages multiple Sidekiq Processes (also provided via sidekiq-pool ruby gem).

MULTI-PROCESS

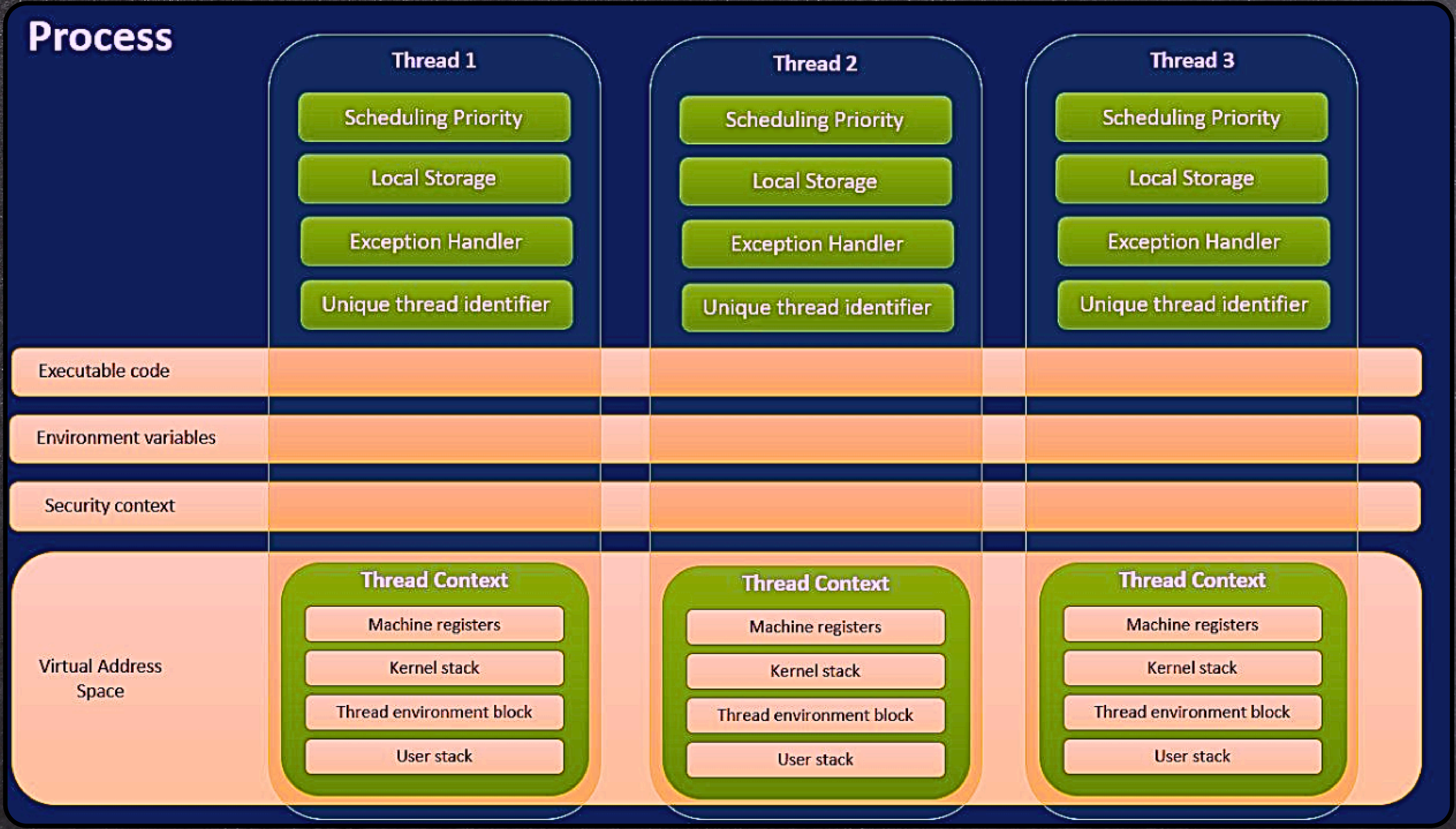
SERVER MANAGER + WORKERS (VIA FORK)

Despite "copy on write" semantics of recent fork(), frequently forking sub-processes is relatively slow and expensive.

That said, sometimes it makes sense, eg PostgreSQL forks a new process for each connection, although it typically maintains a pool of idle connections ready to work.

This is also because PostgreSQL developers actively use UNIX shared memory segment called shared buffers.

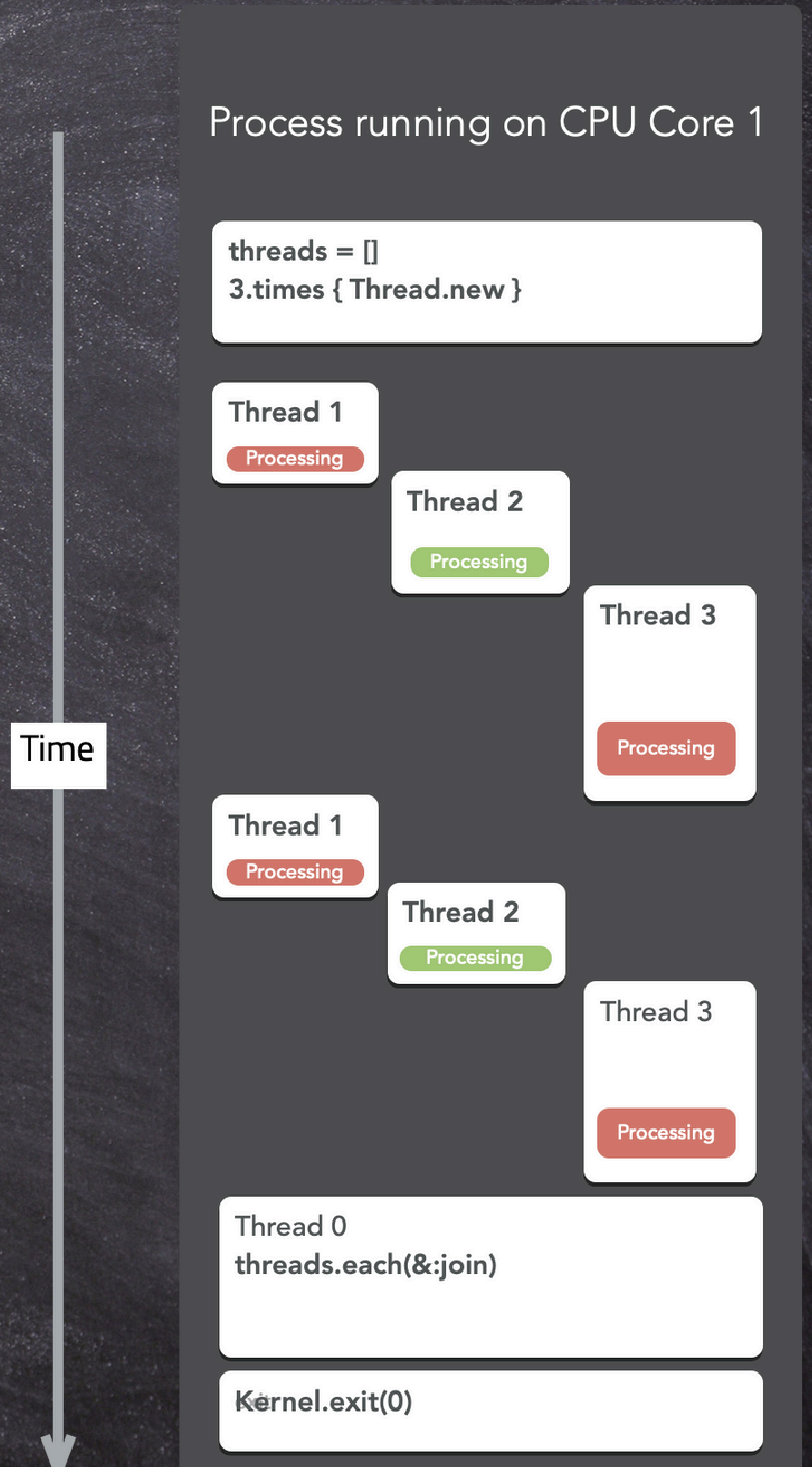
MEMORY ALLOCATION IN THREADS



CONTEXT SWITCHING

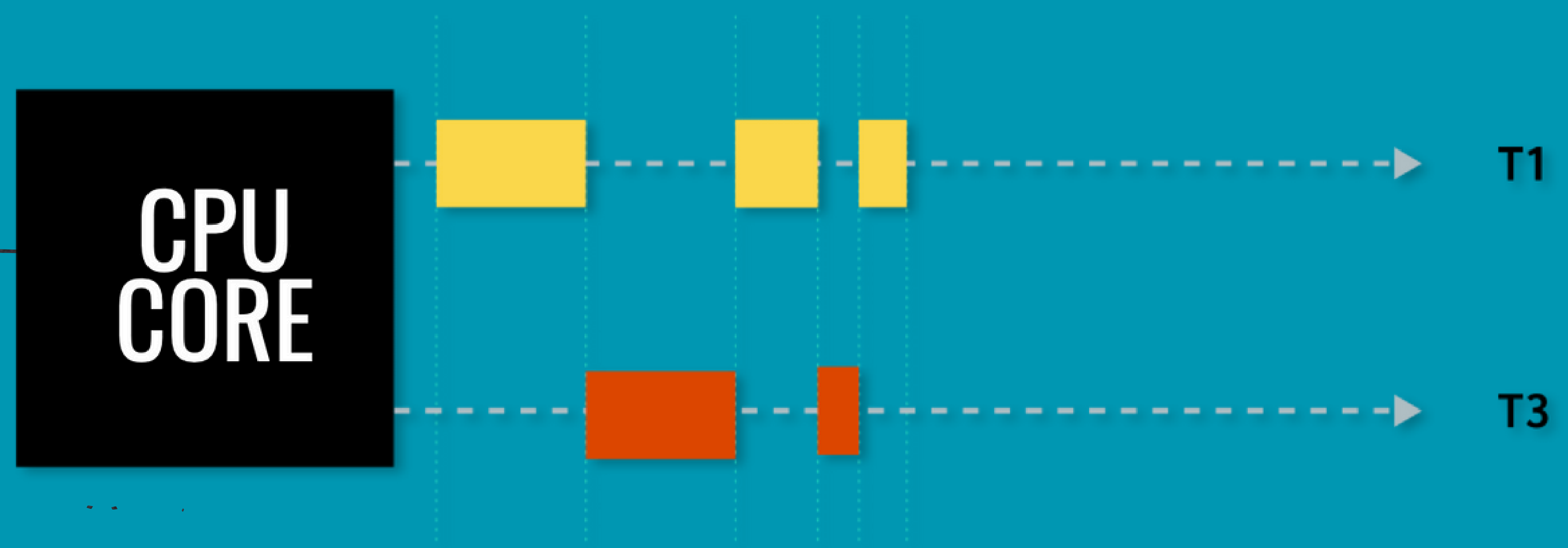
Context switching in a UNIX operating system refers to the process of saving the state of a currently running process or thread and restoring the state of another process or thread so that the CPU can execute it.

This allows the operating system to manage multiple processes and threads efficiently, even on a single CPU core, by rapidly switching between them.



KONSTANTIN GREDESKOUL

CPU CONTEXT SWITCHING



PRE-RUBY 3

SUMMARY

- Thread.new { } starts a new native thread in Ruby, but it's bound to a single CPU core.
  - Scheduling is performed by the OS Kernel, with some feedback from the programmer via Thread.pass method.

- Fiber.new { } starts a new fiber within a Thread, allowing the programmer to control which Fiber gets the CPU at any given time.

- To take advantage of multi-CORE hardware, we (ruby developers) had to:
  - Run Puma with many workers (multi-process concurrency)
  - Run many single-CPU core Docker containers (multi-container concurrency)
  - Run Puma and Sidekiq with # of threads > 1 to utilize time spent in disk and network IO

- It worked (to saturate multiple cores), but it used a ton of memory.

MOORE'S LAW?

Since the raw clock-speed of CPUs tapered off, manufacturers went wide "horizontally" instead of "vertically": constantly increasing the number of CPU Cores that can execute concurrently.

That forced software industry to adapt to multi-core systems and to develop new languages (such as Go & Rust) that are fully capable of ususing multiple cores without the complexity penalty required by the previous generation of concurrent constructs: such as the "pthread" C++ library.

JRuby could always use native threads and saturate all CPU cores.
MRI Ruby couldn't do that until now.

1 Ruby 3 Came out with Ractor

2 So even a Ruby Programer can has many CPU cores

3 We can finally saturate all CPU cores with one Ruby 3 process. Yay!

RUBY 3 FTW!!!

KONSTANTIN GREDESKOUL

AUGUST 2024

RACTOR

KONSTANTIN GREDESKOUL

AUGUST 2024

WHO THE HELL IS THIS TRACTOR?

Ractors are Ruby's answer to true parallelism. Each Ractor has its own separate memory and can run on a different CPU core, bypassing the GIL. Communication between Ractors is done via message-passing, ensuring isolation.

In that sense Ractors are like Services that send and receive messages, but live inside a single Ruby Process.

LIKE A THREAD, BUT CAN'T SHARE ANYTHING

Ractors are suitable for CPU-bound tasks that require parallel execution across multiple cores while maintaining memory isolation.

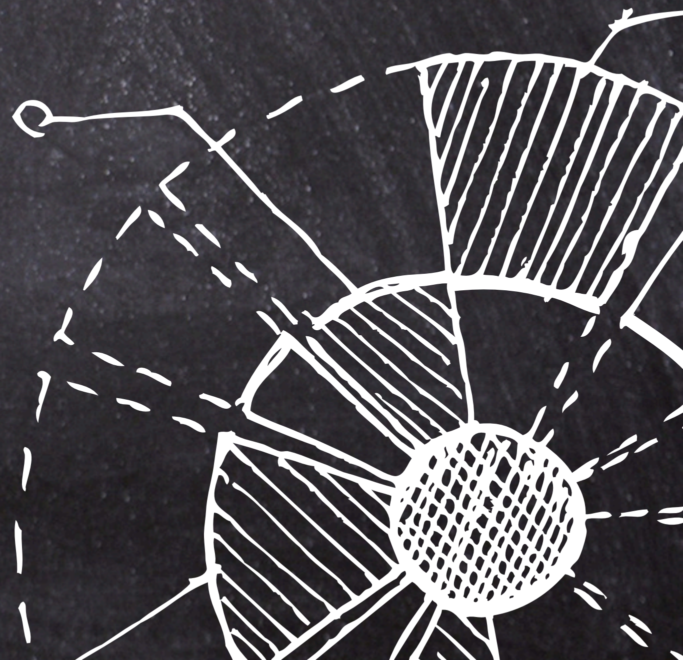
Ruby 3 process with Ractors can finally SATURATE ALL CPU CORES.

Finally I can mine my Crypto using a single MRI Ruby Process....

WHO THE HELL IS THIS TRACTOR?

ACTUALLY RACTORS CAN SHARE SOME DATA: AS SHAREABLE OBJECTS

- Unlike Processes, Ractors CAN share some memory: the can share so-called “Shareable Objects”.
- Frozen constants are shareable.
- Class and Module objects are shareable so the Class/Module definitons are shared between ractors.
- Ractor objects themselves are also shareable objects.



EXAMPLES

SHARED METHODS

```ruby
require 'benchmark'
require 'async'
require 'digest'

ENV['RUBYOPT'] = 'W0'

def factorial(n)
  n == 0 ? 1 : n * factorial(n - 1)
end

def digest(word)
  sleep 1
  Digest::SHA256.hexdigest word
end
```

MULTI-PROCESS

```ruby
Benchmark.bmbm(10) do |x|
  x.report('sequential:') do
    4.times do
      1000.times { factorial(1000) }
    end
  end

  x.report('processes:') do
    pids = []
    4.times do
      pids << fork do
        1000.times { factorial(1000) }
      end
    end
    # wait for child procceses to exit
    pids.each { |pid| Process.wait(pid) }
  end
end

#                    user     system      total        real
# sequential:    1.439956   0.005165   1.445121 (  1.445128)
# processes:     0.000758   0.007042   1.683600 (  0.428644)
```

MULTI-THREADED

```ruby
Benchmark.bm do |x|
  x.report('sequential:') do
    4.times do
      1000.times { factorial(1000) }
    end
  end

  x.report('threads:') do
    threads = []
    4.times do
      threads << Thread.new do
        1000.times { factorial(1000) }
      end
    end
    # wait for all thread to finish using join method
    threads.each(&:join)
  end
end

# user      system      total        real
# sequential:   1.441784   0.006109   1.447893 (  1.447912)
# threads:      1.468147   0.008806   1.476953 (  1.476755)
```

MULTI-FIBERES

```ruby
# frozen_string_literal: true

fib2 = nil

fib = Fiber.new do
  puts '1 - fib started'
  fib2&.transfer
  Fiber.yield
  puts '4 - fib resumed'
end

fib2 = Fiber.new do
  puts '2 - control moved to fib2'
  fib.transfer
end

fib.resume
puts '3 - fib paused execution'
fib.resume

# 1 - fib started
# 2 - control moved to fib2
# 3 - fib paused execution
# 4 - fib resumed
```

FIBERS WITH SCHEDULER

```ruby
animals = %w[fox rat bat owl]

Benchmark.bm do |x|
  x.report('fibers without scheduler:') do
    fibers = []
    animals.each do |word|
      fibers << Fiber.new do
        digest(word)
      end
    end
    fibers.each(&:resume)
  end

  x.report('fibers with scheduler:') do
    Thread.new do
      Fiber.set_scheduler(Async::Scheduler.new)
      animals.each do |word|
        Fiber.schedule do
          digest(word)
        end
      end
    end.join
  end
end

# user      system      total         real
# fibers without scheduler:   0.000816   0.000188   0.001004 (  4.003279)
# fibers with scheduler:      0.002658   0.001208   0.003866 (  1.006556)
```

RACTORS

```ruby
def factorial(n)
  n == 0 ? 1 : n * factorial(n - 1)
end

Benchmark.bm do |x|
  x.report('sequential:') do
    4.times do
      1000.times { factorial(1000) }
    end
  end

  x.report('ractors:') do
    ractors = []
    4.times do
      ractors << Ractor.new do
        1000.times { factorial(1000) }
      end
    end
    # take response from ractor, so it will actually execute
    ractors.each(&:take)
  end
end

# user      system      total        real
# sequential:   1.431720   0.005095   1.436815 (  1.437175)
# ractors:      2.226264   0.044831   2.271095 (  0.848970)
```

```ruby
require 'etc'

def tarai(x, y, z) =
  x <= y ? y : tarai(tarai(x-1, y, z),
                     tarai(y-1, z, x),
                     tarai(z-1, x, y))


cores = Etc.nprocessors
Benchmark.bm do |x|
  # x.report('seq'){ cores.times{ tarai(14, 7, 0) } }

  x.report('par'){
    cores.times.map do
      Ractor.new { tarai(14, 7, 0) }
    end.each(&:take)
  }
rub… sorbet  91% ln:21 %:1
end

N… ractor.rb[+]                rub… sorbet  91% ln:22 %:1
```

```
✗ kig@macbook-m3-kig ~ ruby examples/ractor.rb
       user     system      total        real
parexamples/ractor.rb:18: warning: Ractor is experimental, and the behavior may change in future versi
ons of Ruby! Also there are many implementation issues.
```

```
  1[||||||||||||||||||||||||||||||||||||||||||          50.3%] Tasks: 958, 3119 thr, 0 kthr; 9 running
  2[||||||||||||||||||||||||||||||||||||||              46.0%] Load average: 2.83 2.94 2.97
  3[||||||||||||||||||||||||||||||||                    38.0%] Uptime: 24 days, 04:39:18
  4[||||||||||||||||||||||||||||                        33.3%] Battery: 100.0% (Running on A/C)
  5[||||||||||||||||||||||||                            28.3%] Uptime: 24 days, 04:39:18
  6[|||||||||||||||||||||||                             27.3%] Time: 15:37:11
  7[|||||||||||||||||||||||||||||                       35.8%]
  8[||||||||||||||||||||||||||||||||||                  40.7%]
  9[|||||||||||||||||||                                 24.5%]
 10[|||||||||||||||||||||||||||||                       34.7%]
 11[||||||||||||||||                                    19.1%]
 12[|||||||                                              7.2%]
 13[||||||||||||||                                      17.3%]
 14[|||                                                  3.3%]
 15[||                                                   1.3%]
 16[||||||||||||||||                                    19.2%]
Mem[||||||||||||||||||||||||||||||||||||||||||||||32.4G/64.0G]
Swp[                                                   0K/0K]

  PID△USER      TIME+   CPU% MEM% NLWP Command
26114 kig      0:03.00   0.0  0.0    9 │        └ ruby examples/ractor.rb

F1Help  F2Setup F3SearchF4FILTERF5List  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

HELPFUL GEMS

gem "async"

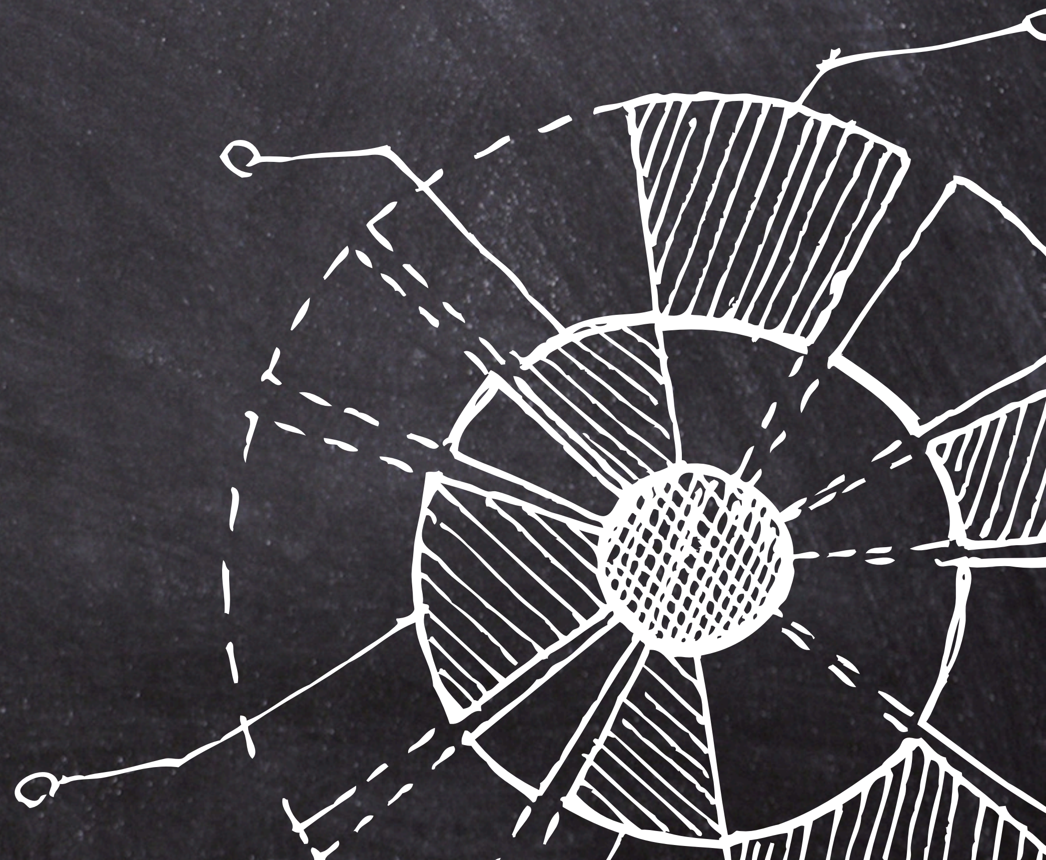
```ruby
require 'async'

reactors = []
reactors << Async::Reactor.new # internally calls Fiber.set_scheduler

# This should run in the above reactor, rather than creating a new one.
Async do
  puts "Hello World"
end
```

HELPFUL GEMS

```ruby
require_relative 'helpers'

array = (0..5000).to_a

Benchmark.bm do |x|
  x.report("in #{NPROC} processes:") do
    # reproducibly fixes things (spec/cases/map_with_ar.rb)
    Parallel.each(array, in_processes: NPROC) do |number|
      factorial(number)
    end
  end

  x.report("in #{NPROC} threads:") do
    # maybe helps: explicitly use connection pool
    Parallel.each(array, in_threads: NPROC) do |number|
      factorial(number)
    end
  end
end

#                         user     system      total        real
# in 16 processes:   0.066134   0.101054  14.111030 (  1.065917)
# in 16 threads:    10.190970   0.441881  10.632851 ( 10.630783)
```

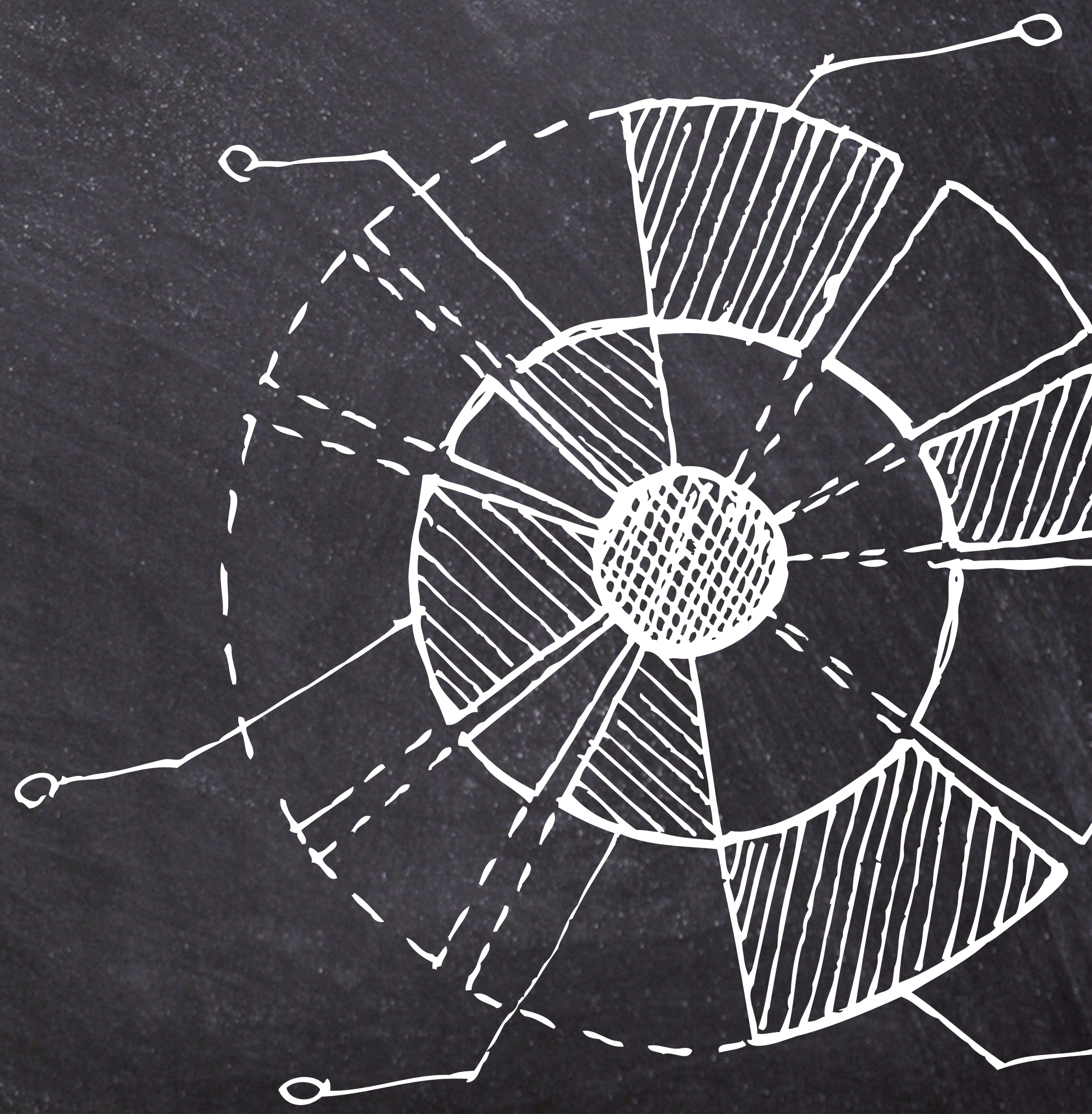
gem ‘parallel’

now with ractor support



CONCLUSION

- Use Processes when you need complete isolation between tasks and want to utilize multiple CPU cores without worrying about shared memory.
- Use Threads when you need lightweight concurrency within a process, especially for IO-bound tasks, but be mindful of the GIL and thread safety.
- Use Fibers for highly cooperative multitasking or when you need to manage non-blocking IO operations efficiently without parallelism.
- Use Ractors when you need to run CPU-bound tasks in parallel across multiple cores within a single process, ensuring safe concurrency without shared state issues.



REFERENCES

- Concrrency: Threads, Fibers & Ractors
- Introduction to Ractors in Ruby
- My Adventure with Async Ruby
- Ruby 3.3 Documentation: Ractor Class

QUESTIONS?

THANK YOU!

THE END

https://github.com/kigster – open source

https://twitter.com/kig – random thoughts

https://kig.re/ – blog

https://reinvent.one/ – consulting

https://youtube.com/@kigster – videos