Runtime Model of Ruby, JavaScript, Erlang,

and other popular languages

April, 2017 Kraków

Sergii Boiko

Full Stack Engineer

railsware

Why Runtime Model?

Simulating Fireworks

Why not use Erlang processes for simulating particles?

Key Behaviors: CPU-bound and IO-bound

CPU-bound: How fast can we calculate something?

IO-bound: How many "simultaneous" interactions with the outer world can we handle?

Main Questions to Runtime Model

- How efficient are CPU-bound tasks?
  - does runtime support parallelism?
- How efficient are IO-bound tasks?
  - what is concurrency model?
- How efficient is Memory management?
  - does runtime use GC and what kind of GC?

CPU-bound

CPU-bound tasks

CPU-bound - time to complete a task is determined by the speed of the CPU

Examples:

- compiling assets
- building Ruby during installation
- resizing images
- creating ActiveRecord objects after obtaining response from database

CPU-bound: key efficiency factors

1. Bare performance
2. Parallelism
3. GC or non-GC

CPU-bound tasks: Bare Performance

The closer to bare metal - the faster

The champions: statically typed languages compiled to native code

- C/C++
- Rust
- Swift
- Go

CPU-bound tasks: Bare Performance

The runner-ups: statically typed languages with JIT

- Java, Scala
- C#, F#

CPU-bound tasks: Bare Performance

Not that bad: dynamic languages with JIT

Raw estimation: best is about 50% of statically typed languages performance

- Clojure
- JavaScript V8
- JRuby Truffle

CPU-bound tasks: Bare Performance

The also-runs: dynamic languages without JIT

- Erlang / Elixir
- Python
- Ruby MRI

CPU-bound tasks: Parallelism

Parallelism - simultaneous execution of computations

Boils down to using all available cores of CPU

CPU-bound tasks: Parallelism

Parallel:

- C/C++
- Rust
- Go
- JVM (Java, Scala, Clojure, JRuby)
- .NET (C#, F#)
- Haskell
- Erlang / Elixir

Non-Parallel (GIL):

- Ruby MRI
- Python

Non-Parallel (Event Loop):

- JavaScript (Node.JS)
- Ruby (EventMachine)
- Python (Twisted)

CPU-bound tasks: non-GC vs GC

Raw estimation: ~10% performance penalty when using GC

Non-GC:

- C/C++
- Rust
- Swift

GC:

- Go
- Java / Scala
- C# / F#
- JavaScript
- Ruby
- Erlang / Elixir
- Python

CPU-bound tasks

Best combo:

- Statically-typed, compiled to native code
- Non-GC
- Parallel

Garbage Collector

Garbage Collector

Main contributions to Runtime Model:

1. Expect about ~10% of performance penalty
2. Leads to GC "pauses" in execution
3. There is a maximum heap size, which can be handled efficiently

Garbage Collector Types

Reference-Counting:

- Python
- Perl 5

Tracing:

- JVM
- .NET
- Go
- Ruby
- JavaScript/Node.JS
- Haskell
- Erlang / Elixir
- …

Garbage Collector: Tracing

Generational GC dominates

Generational GC:

- JVM
- .NET
- Ruby
- JavaScript/Node.JS
- Erlang / Elixir
- Haskell

Concurrent, Tri-color, mark-sweep:

- Go

GC at a different angle

1. GC in a shared-heap runtime
2. GC in a multi-heap runtime

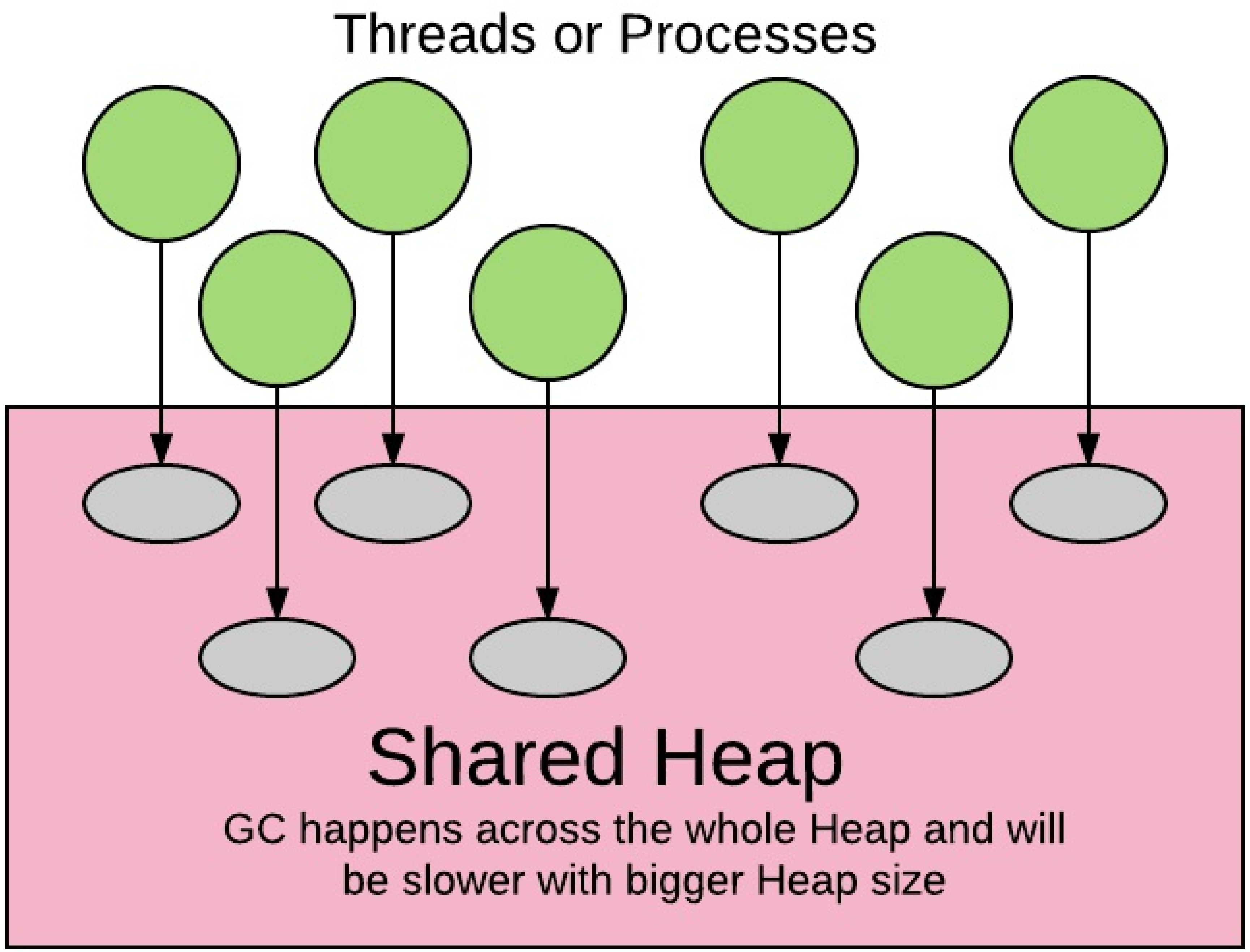
GC in a shared-heap runtime

Mostly all popular runtimes use shared heap

Examples:

- JVM
- .NET
- Go
- Ruby
- JavaScript/Node.JS
- Python
- Haskell
- ...

GC in a shared-heap runtime



GC in a shared-heap runtime

Main issue: GC should be done across one chunk of memory and complexity of GC grows linearly (or worse) with a heap size

Outcomes:

- Performance degradation on big heap size
- Increased delays in runtime execution due to GC

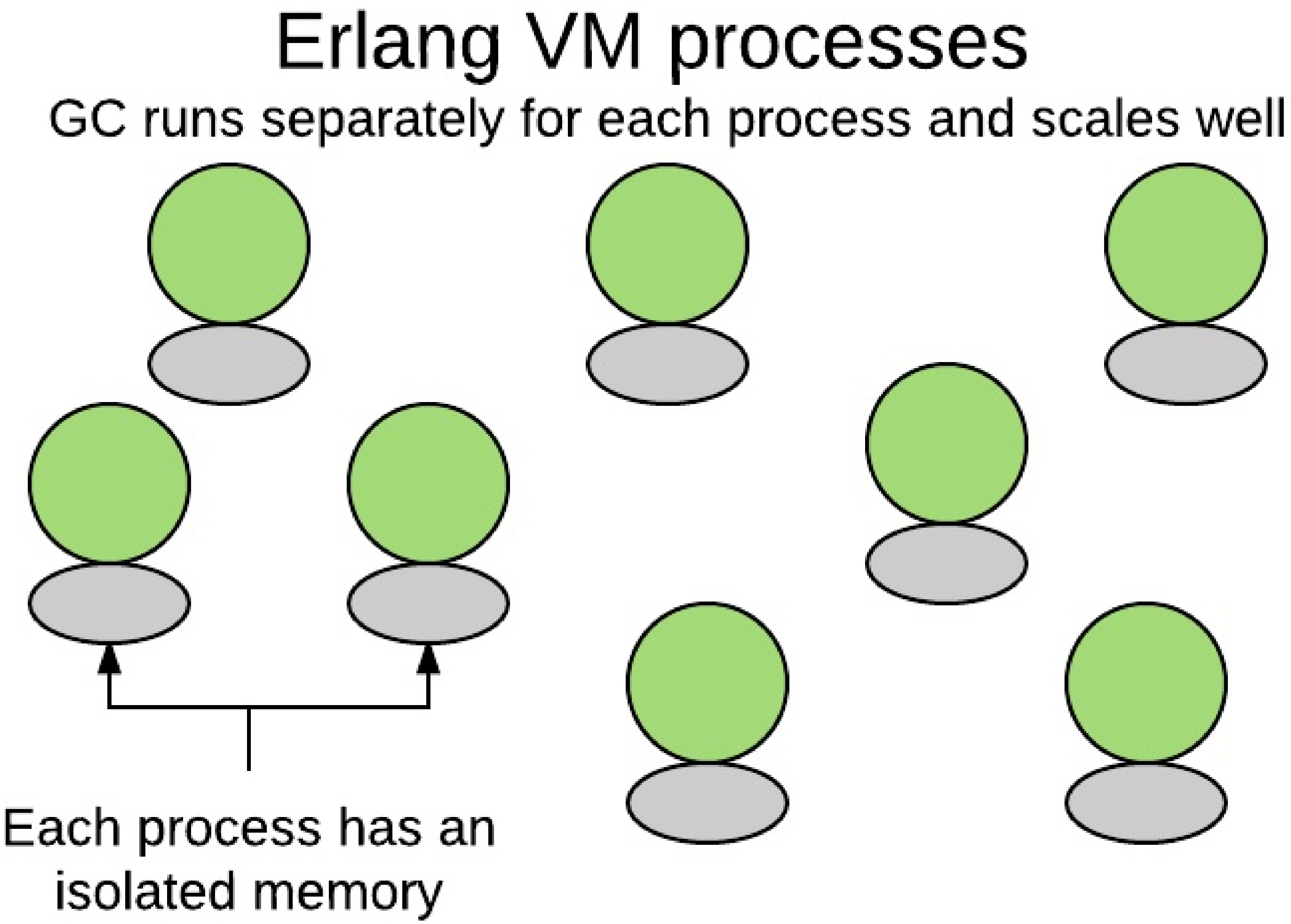
GC in a multi-heap runtime

Example:

- Erlang VM

Main trick: each process has its own isolated heap, which shares nothing with others

GC in a multi-heap runtime



GC in a multi-heap runtime

Main outcomes of Erlang VM memory layout:

1. GC can be done on a much smaller area and scales well
2. GC can be totally avoided by finishing process before GC kicks in (web-request is finished)
3. Erlang VM can use 128Gb and support 2 million active connections

IO-bound

IO-bound

IO-bound: how many "simultaneous" interactions with the outer world can we handle?

Examples:

- IRB/Pry input/output
- Reading file content
- Handling web request
- Handling websocket connection
- Performing database query
- Calling remote service
- Reading data from Redis
- Sending Email

IO-bound: Blocking vs Non-Blocking IO

Synchronous or Blocking: waits for the other side to be ready for IO-interaction

Asynchronous or Non-Blocking: handles other IO-interactions until the other side is ready to interact

IO-bound: Synchronous or Blocking

Pros:

- Easy to develop - sequential code

Cons:

- Working thread is dedicated to only one IO-interaction

IO-bound: Asynchronous or Non-Blocking

Pros:

- High performance - ability to handle large number of connections

Cons:

- Harder to write code
- Callback / Promise hell

IO-bound: Main Concurrency Models

1. Blocking IO + OS Threads
2. Event Loop or Reactor pattern
3. Green Threads

IO-bound: Blocking IO + Threads

In such combination runtime blocks on any IO operation, and OS handles switch to another thread.

Pros:

- Easy to write logic per thread - everything is sequential
- Quite performant - max limit ~ 5000 concurrent threads
- Memory efficient - everything is shared

Cons:

- Shared state is a big issue for mutable languages
- Requires usage of different thread synchronization primitives
- High requirements to quality of third-party libraries

IO-bound: Blocking IO + Threads

Managed Runtimes:

- JVM
- .NET
- Ruby MRI (despite having GIL)
- Python (despite having GIL)

IO-bound: Blocking IO + Processes

- Unicorn - Ruby web-server
- Postgres for handling client connections
- Apache (one of the modes)

Pros:

- Isolated memory - no risks of simultaneous writes to the same memory
- Sequential, "blocking" code

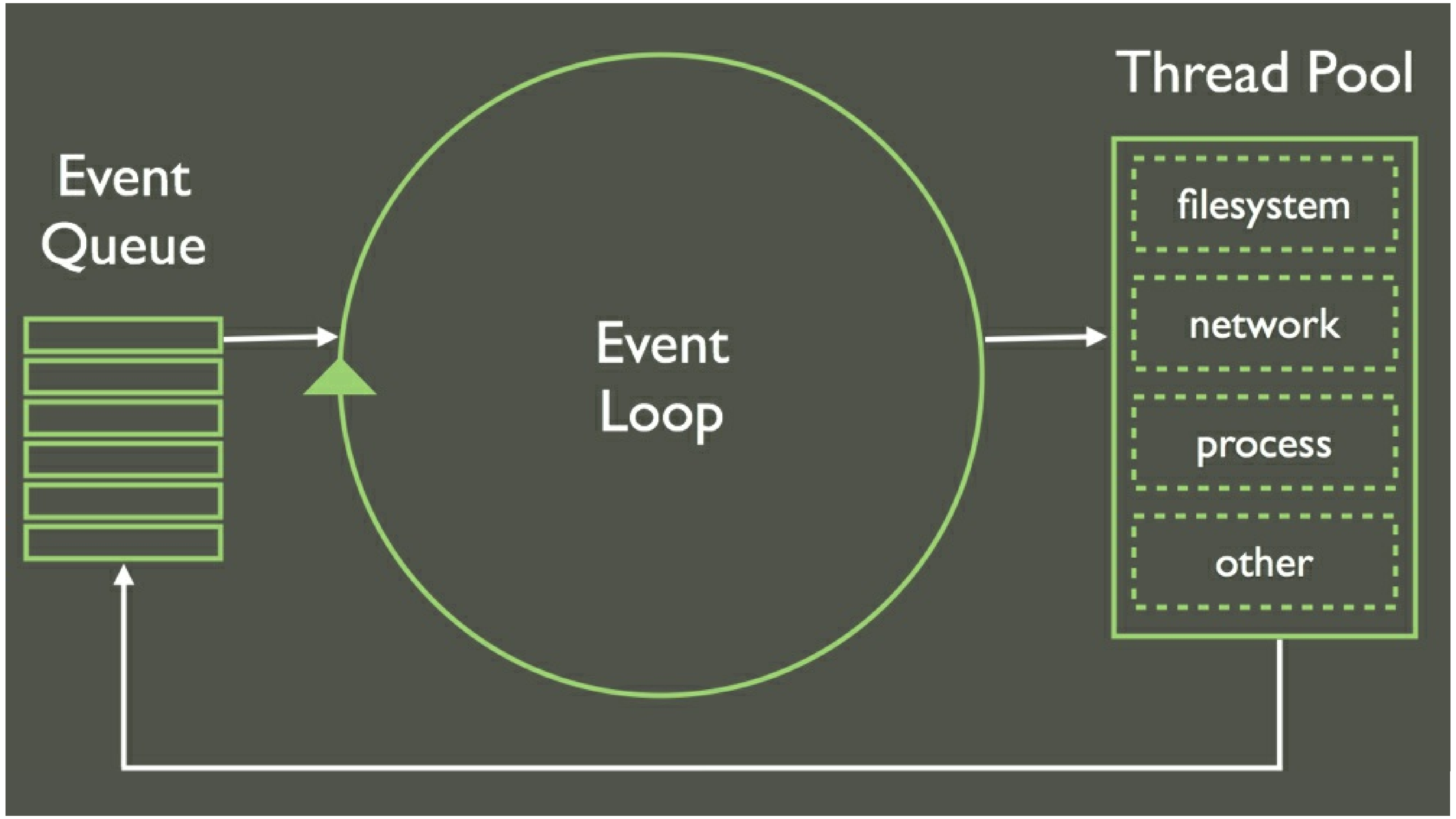
Cons:

- Higher memory consumption compared to threads
- Lower performance compared to asynchronous mode

IO-bound: Asynchronous: Event Loop or Reactor pattern

- Node.JS
- Ruby + EventMachine
- Python + Twisted

IO-bound: Asynchronous: Event Loop



IO-bound: Asynchronous: Event Loop or Reactor pattern

Pros:

- Memory-efficient - shared memory
- Memory-safe - no race conditions, because only one "callback" is performed till it finishes
- High-performant - potentially can handle millions of connections

Cons:

- Single-threaded - only one CPU core is used
- Callback / Promise hell, but can be avoided with coroutines or async/await

IO-bound: Synchronous Asynchronity: Green Threads

Instead of using OS threads, runtime has its own scheduler and manages threads without OS.

- API looks like synchronous
- But under the hood everything runs asynchronously

Green Threads Benefits compared to OS Threads:

- Smaller memory usage per thread
- Cheaper context-switch

IO-bound: Green Threads: Failure Stories

- Java 1.1
- Ruby 1.8

Main issues:

- Using mutexes as a main concurrency primitive
- Not efficient implementation

Both switched to OS Threads

IO-bound: Green Threads: Success Stories

- Erlang VM
- Golang VM
- GHC Haskell

Main difference - simpler set of primitives for handling concurrency without mutexes and synchronization:

- Actors and message-passing in Erlang
- Gorotines and channels in Go
- MVar and STM in Haskell

IO-bound: Green Threads of Go

Pros:

- Sequential "blocking" code
- Memory-efficient - one virtual machine
- Non-copying memory exchange between goroutines through channels
- Great IO-performance

Cons:

- Go still has a possibility to mutate a global state

IO-bound: Green Threads of Erlang

Pros:

- Sequential "blocking" code
- Memory-efficient - one virtual machine
- Great IO-performance: can handle about ~2_000_000 connections

Cons:

- Copies data when exchanging messages between processes

Ruby MRI Runtime

Ruby MRI Runtime: CPU-bound

CPU-bound:

- dynamic
- no JIT
- non-parallel

CPU-bound performance is poor

MRI-team is going to add JIT, still not clear when it will happen

Ruby MRI Runtime: GC

GC:

- one big heap - delays in GC
- generational mark&sweep - good

Current GC is quite good

MRI-team does not have any further plans to improve GC speed

Ruby MRI Runtime: IO-bound

IO-bound:

- blocking
- single-threaded
- multi-process (Unicorn)

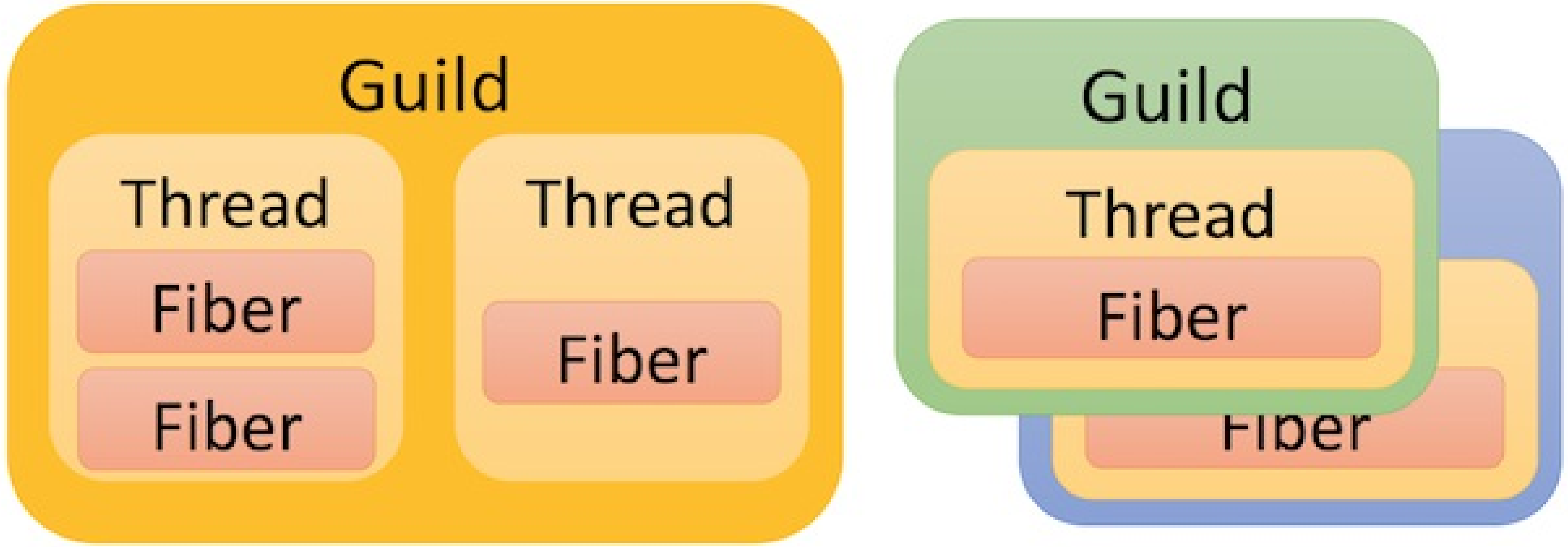
IO-bound stuff is not efficient compared to Node.JS or Erlang - slower 5-10x

Why Ruby doesn't use Threads and Java does?

- Java was built and promoted from the start as a concurrency-focused language
- Every library was meant to work in multi-threaded environment
- Ruby was never built with multi-threading in mind
- Main risk is to run into library which unsafely changes global state

Ruby MRI Runtime: New Hope - Guilds

Guild is a set of Threads and Fibers which can't directly access memory of another Guild



Ruby MRI Runtime: New Hope - Guilds

Pros:

- Memory-efficient - non-mutable stuff is shared (code, freezed objects)
- Memory-safe - different Guilds can't simultaneously mutate same object
- Good enough performance for web-requests
- Parallel - Guilds don't have GIL

Ruby MRI Runtime: New Hope - Guilds

Cons:

- Still can't handle big amount of connections - Guilds are more expensive than Green Threads or Event Loop
- Compared to Event Loop or Green Treads:
    - Memory usage is higher
    - Context switch is slower

Ruby MRI Runtime: New Hope - Guilds

MRI team is keen to implement them, but there are a lot of small issues with memory sharing, which should be addressed

Estimated performance gain: ~3-5x

Other factors contributing to Runtime Model

- Data Structures: mutable or immutable, O(?)
- GC implementation details
- Heap and Stack usage
- Memory Model
- CPU Architecture
- Underlying OS system calls(pthreads, select, epoll/kqueue, etc)
- ...

Follow us on twitter @railsware

railsware