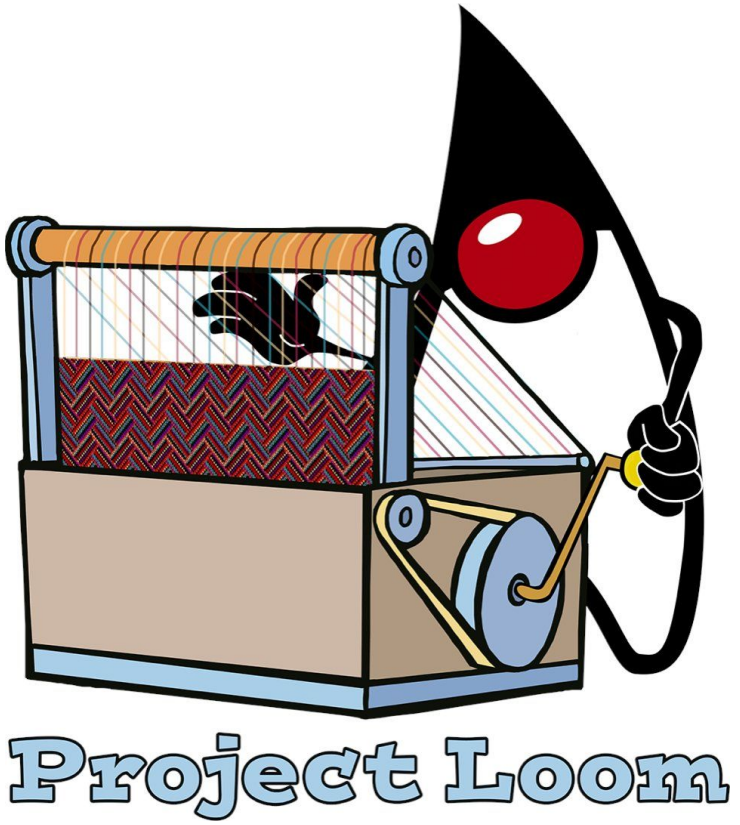
Project Loom e como ele revolucionará a plataforma Java

Eder Ignatowicz
Principal Software Engineer
@ederign

William Siqueira
Senior Software Engineer
@William_Antonio

Red Hat

"... support easy-to-use, high-throughput **lightweight concurrency** and **new programming models** on the Java platform..."

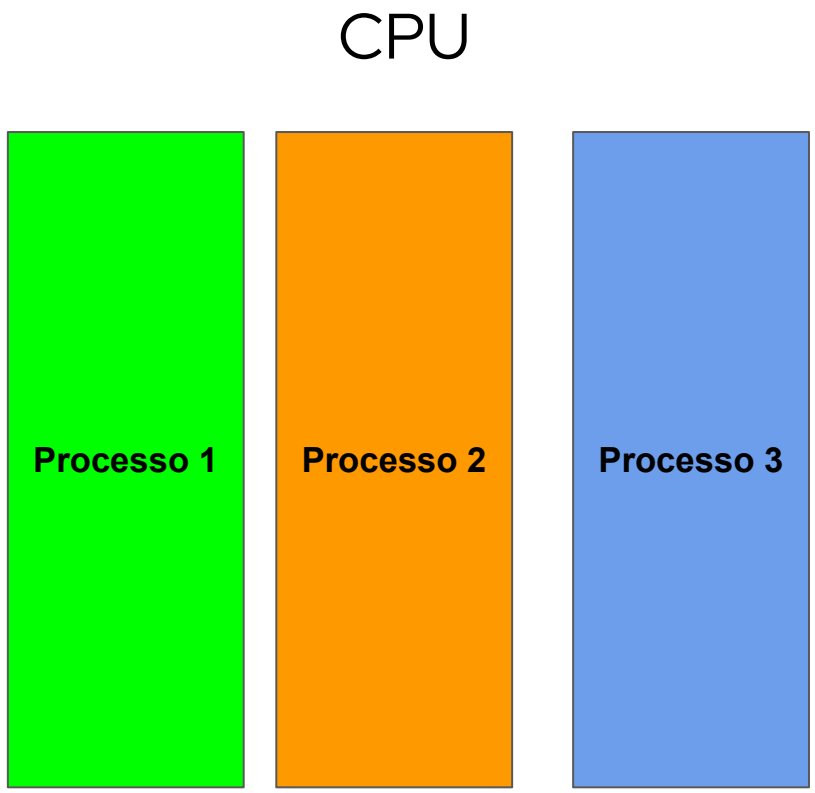


https://wiki.openjdk.java.net/display/loom/Main

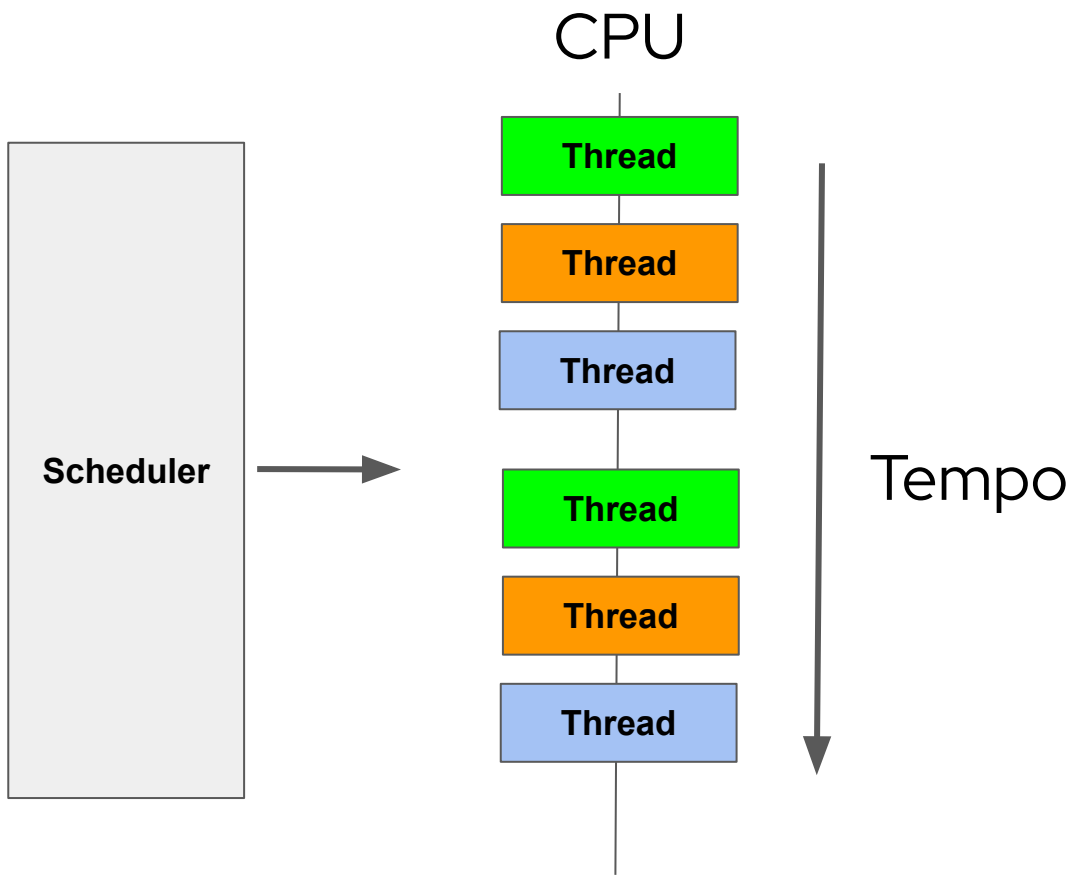
"Introdução" a 180km/h a programação assíncrona em Java!

Red Hat

Como parece que é



Como é:



Threads

Recurso do sistema operacional

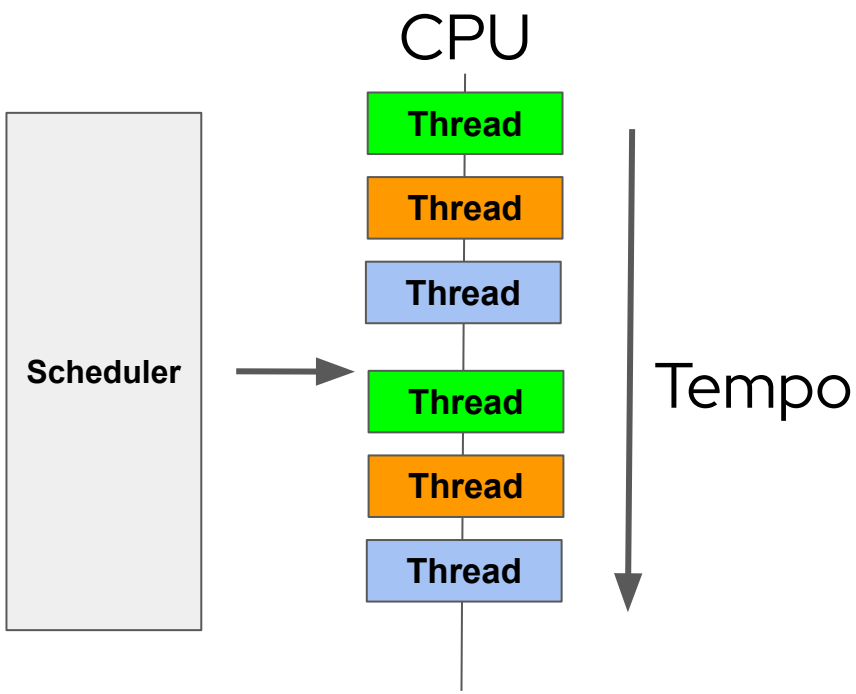
- Possibilita 1 processo ter diversos fluxos de execução em paralelo
- Conjunto de instruções que um programa que podem ser gerenciadas por um scheduler
- 1 processo => N threads executando de forma concorrente e compartilhando recursos
- Recurso limitado dependendo da arquitetura do processador

Pros

- Possibilita "concorrência" (mesmo com 1 core)
- Abstração para a programadora

Cons

- Pesada e carrega muita informação de estado
- Criação e Context switching custoso
- **Recurso escasso e compartilhado do sistema operacional**



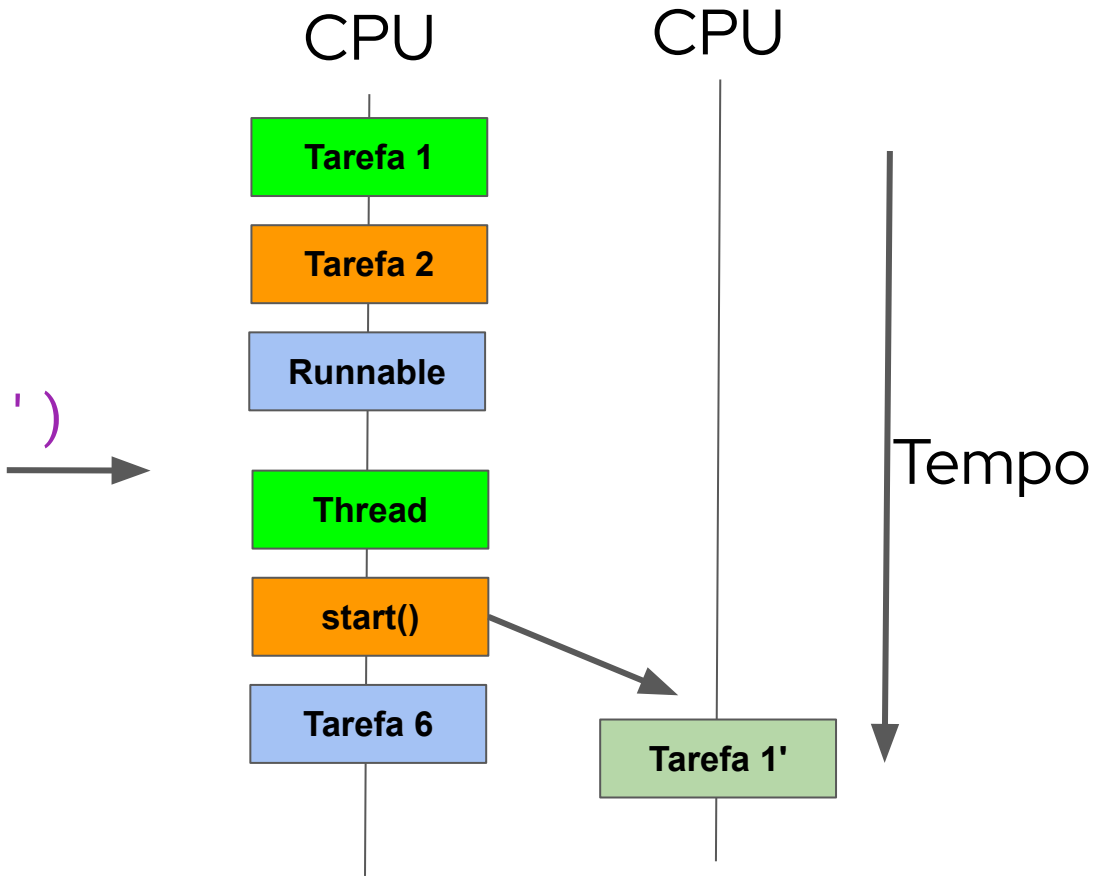
Red Hat

Em Java "tudo" é thread

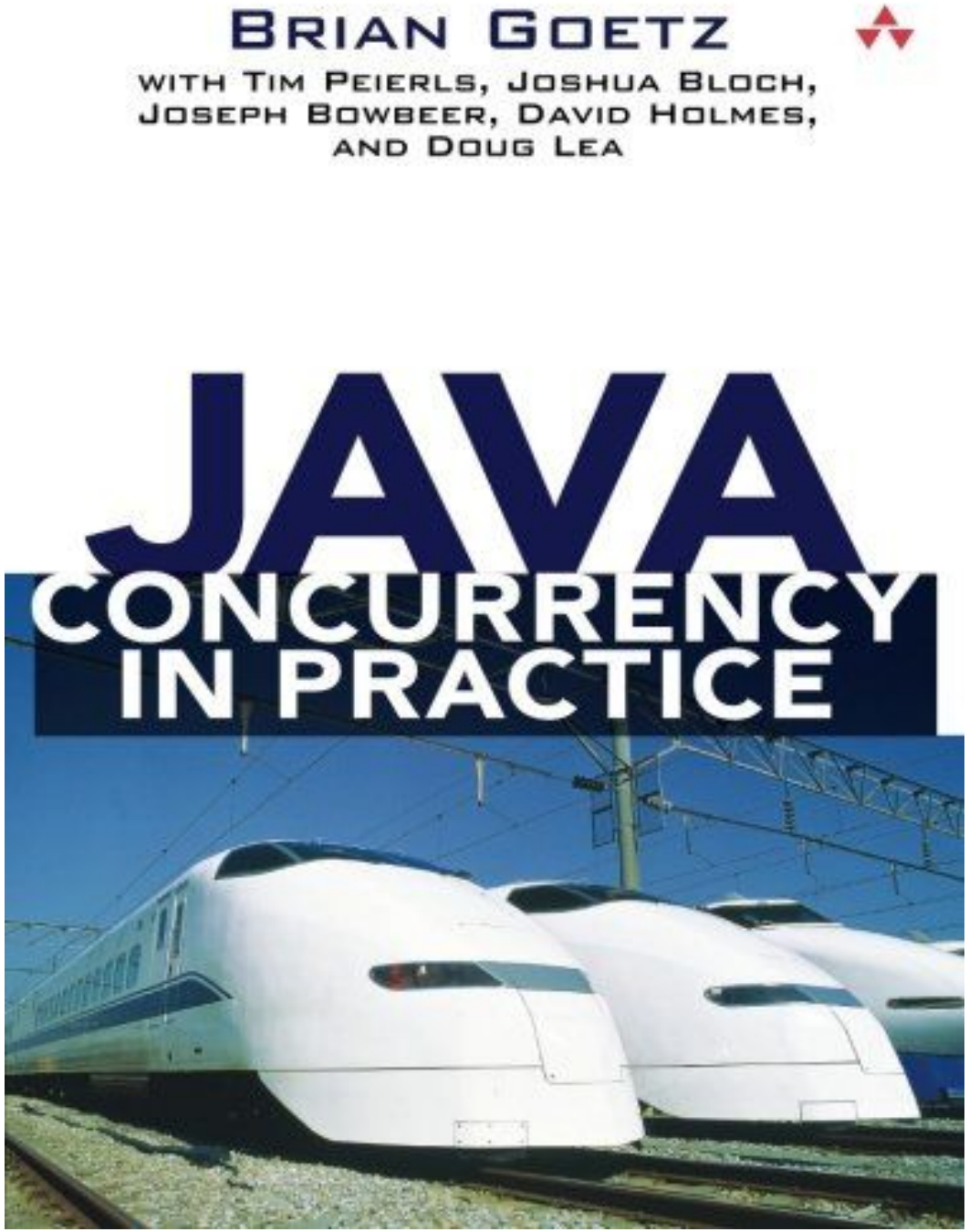
- Unidade básica de concorrência é Thread (java.lang.Thread)
- Exceptions, Thread Locals, Debugger, Profiler

- Thread são mapeadas 1:1 para kernel threads (threads do sistema operacional)
  - Caras para criação
  - Caras para troca de contexto implementadas pelo kernel (switch)
  - Número limitado (tamanho da stack)
  - Scheduling igual para todas

```
//Tarefa 1
//Tarefa 2
Runnable runnable = () -> {
    //o que eu quero executar (tarefa 1')
};

Thread t = new Thread(runnable);
t.start();

//Tarefa 6
```



```
for (int i = 0; i < parameter; i++) {
    Runnable run = () -> {
        //task bem longa e complexa
    };
    Thread th = new Thread(runnable);
    th.start();
}
```


BRIAN GOETZ
WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA
JAVA
CONCURRENCY
IN PRACTICE

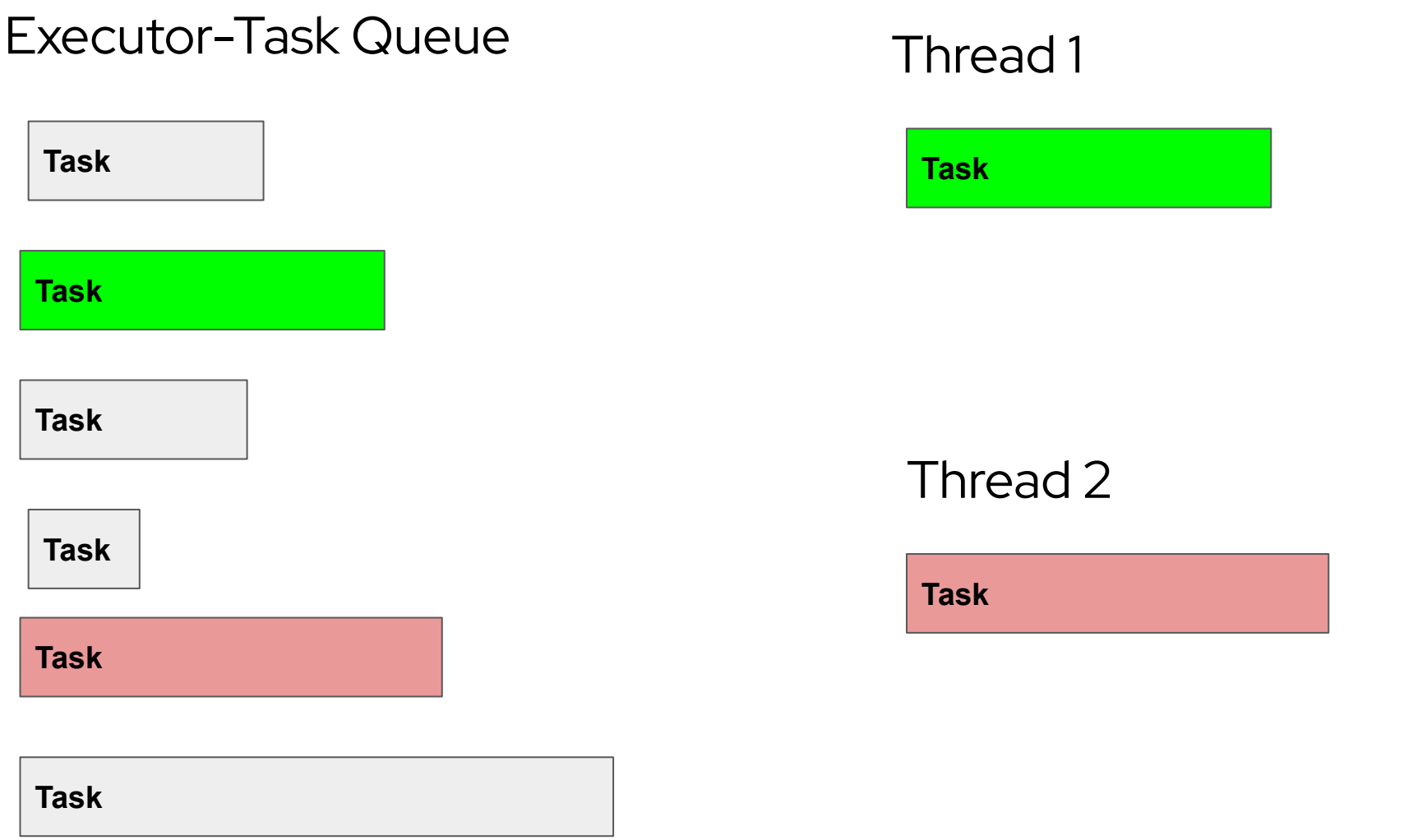
Red Hat

Thread pooling

- Criação de threads centralizadas (normalmente ao subir a aplicação)
- Cada task (tarefa) é executada utilizando uma thread existente e compartilhada do pool

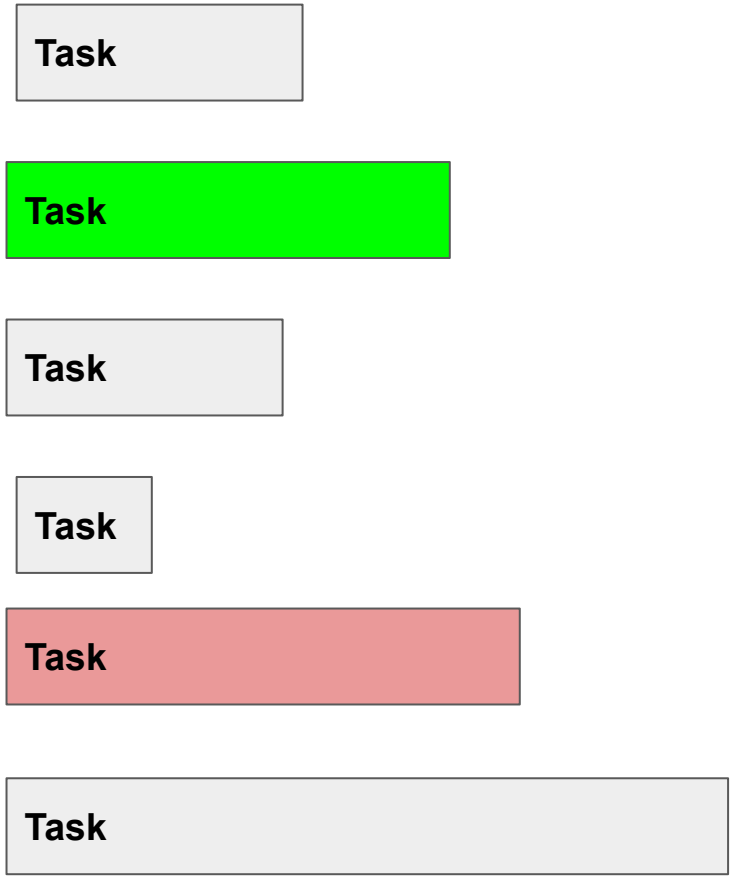
```
ExecutorService executorService =
Executors.newFixedThreadPool(10);

executorService.submit(() -> {
    //task
});
```

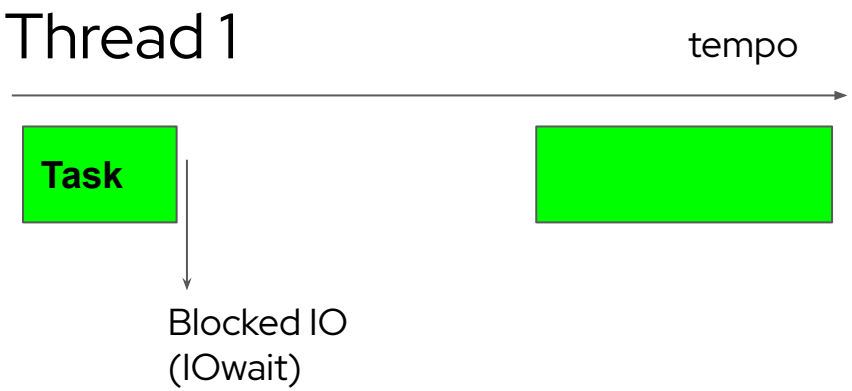
Thread pooling

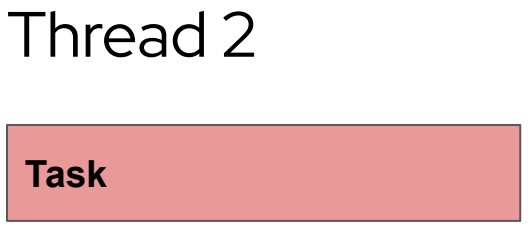


Thread pooling

Executor-Task Queue







I/O bound

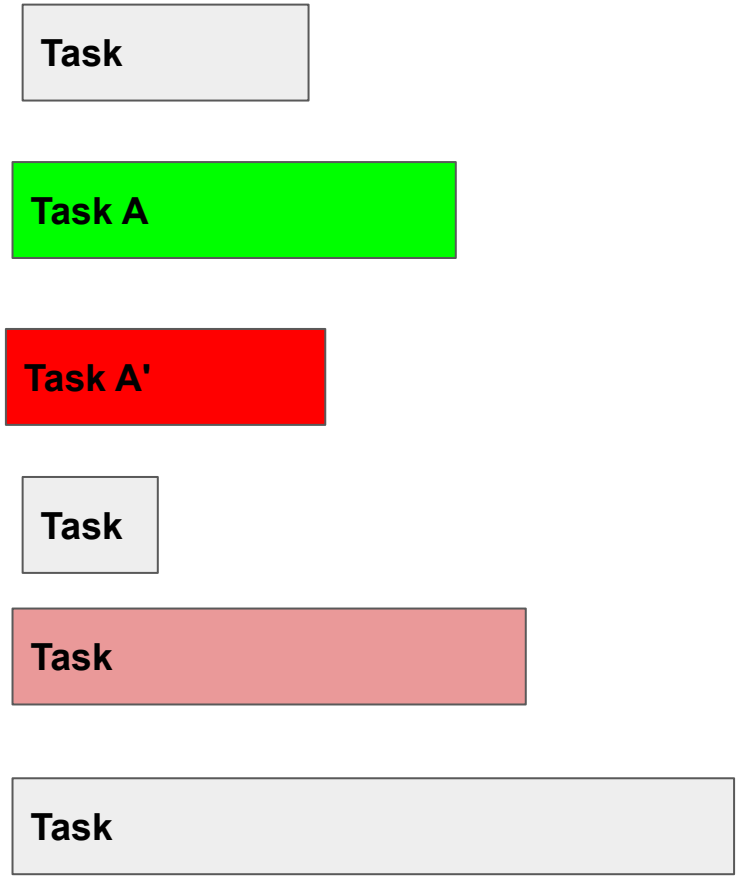
- Exemplos:
  - Ler/escrever arquivos, network, http calls etc.
- Algo não está na RAM e preciso esperar a resposta do hardware;

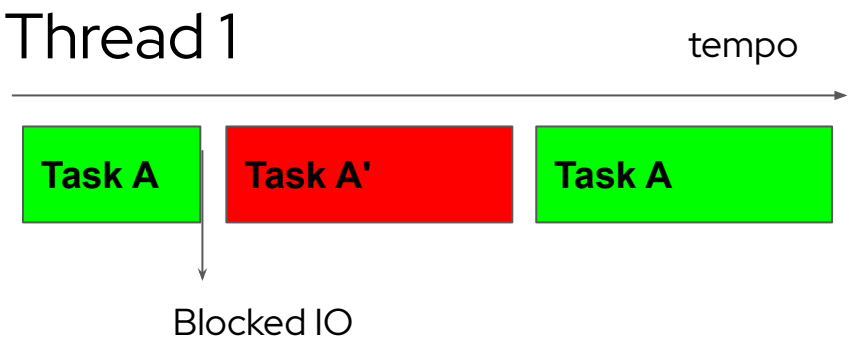
- O que o CPU faz:
  - **IOwait**: system call que fala pro CPU pausar a execução da thread atual, enquanto os dados de resposta do I/O não retornam
  - Thread Java (recurso limitado) vai ficar 'sleeping'

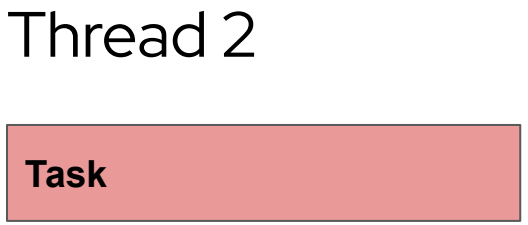
Thread Pools

- Funcionam muito bem para:
  - Operações CPU bound
    - Geralmente processamento no CPU
    - Não bloqueiam
    - Ex. processar uma imagem ou uma matriz
- Não tem bem para:
  - Operações I/O bound
  - File, Socket, Query DB, Thread.sleep. Etc.
  - Porque são operações potencialmente blockantes

O que queremos fazer







Em Java: CompletableFuture<T>

- **CompletableFuture** é um valor, representando uma computação que já está sendo executada em background
- Eventualmente, irá yield ("retornar") um valor do tipo T ou um erro
- Quando tentamos algo blockante, o que ganhamos é uma continuation
  - "O que deve acontecer depois de executado"
- Eles são
  - Baratos para criação
  - Baratos para troca de contexto
  - Ilimitados ('limitados pelo heap)
  - Executados no ForkJoinPool
- **Composable**

```java
CompletableFuture<String> completableFuture
  = CompletableFuture.supplyAsync(() -> "my blocking computation");

CompletableFuture<String> future = completableFuture
  .thenApply(s -> s + " another blocking");

assertEquals("Result", future.get());
```

Mas complicam muito um pouco a nossa vida

```
public void process(Operation op){
    databaseService.process(op);
    auditService.process(op);
    analyticsService.process(op);
    cacheService.process(op);
}
```

Mas complicam muito um pouco a nossa vida

```
public void processAsync(Operation op){
     CompletableFuture<Boolean> db =  CompletableFuture.supplyAsync(() ->
databaseService.process(op));
     CompletableFuture<Boolean> audit =  CompletableFuture.supplyAsync(() ->
auditService.process(op));
     CompletableFuture<Boolean> analytics =  CompletableFuture.supplyAsync(()
-> analyticsService.process(op));
     CompletableFuture<Boolean> cache =  CompletableFuture.supplyAsync(() ->
cacheService.process(op));

     CompletableFuture.allOf( db, audit, analytics, cache).join();
 }
```

Synchronous

Asynchronous

Fácil de ler

Combina bem com nosso jeito de programar Java

Mas é um recurso custoso

Bom pra programadores, ruim para hardware

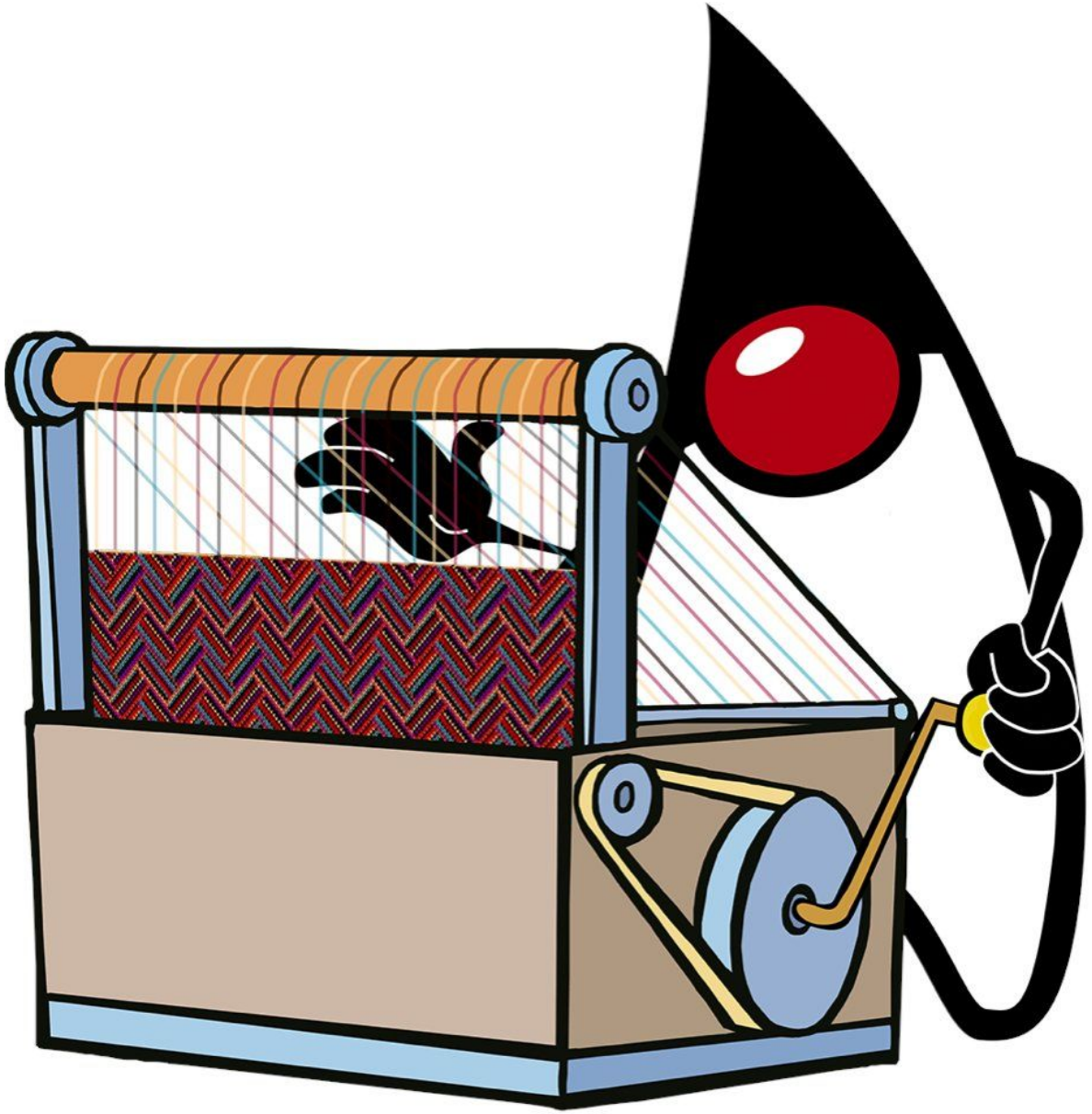
Escalável

Dificil de ler

Perde o contexto da execução

(dificulta debug e profile)

Ruim para programadores, bom para o hardware



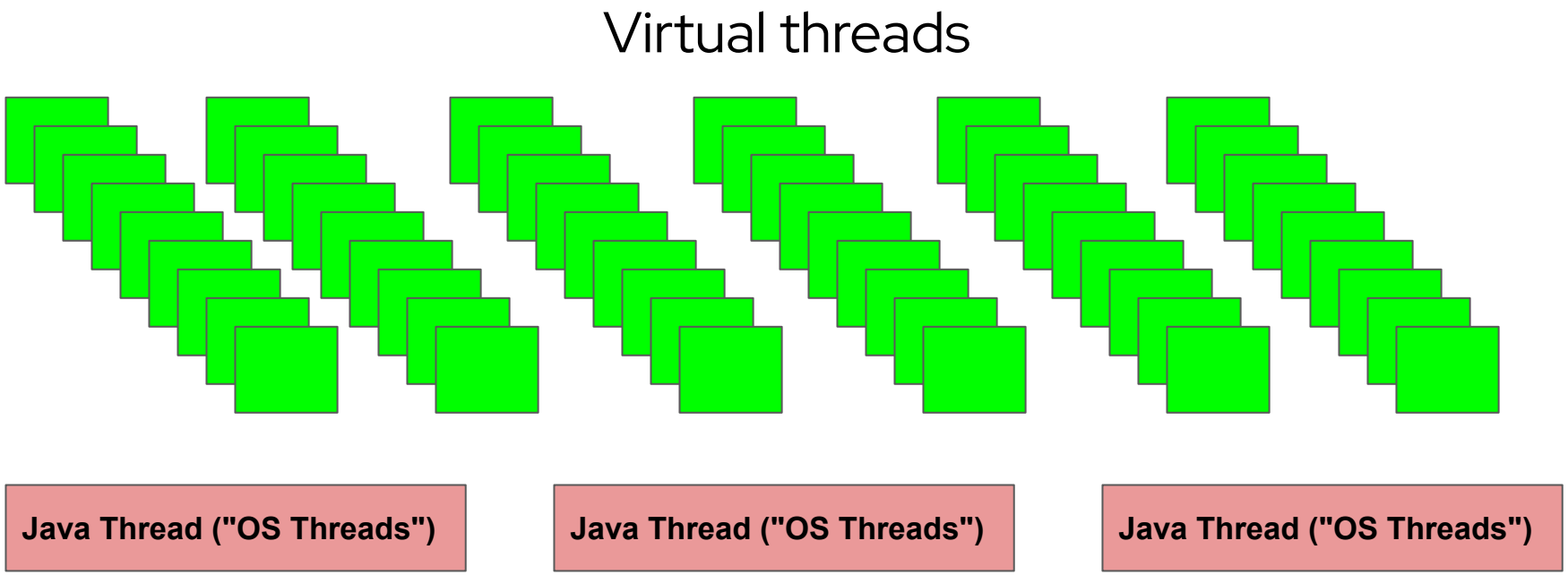
Project Loom

Programe de forma síncrona e escale como se fosse assíncrono

https://wiki.openjdk.java.net/display/loom/Main

Virtual threads

- Fim do mapeamento 1:1 de "Threads" do Java com Threads do Sistema Operacional
- Extensao da API de Threads
- Mesmo conceito que nós já conhecemos
- São multiplexadas em cima de um thread pool do OS
-



Virtual threads

- Qdo o código numa virtual thread blocka, a continuation é suspensa
- Desta forma, código que pode ser resumido de uma virtual thread pra outra
- O código Java não sabe que isto acontece (scheduling) por detrás dos panos
- Footprint (virtual threads x Java Threads)
  - 200-300B metadata vs > 2kb
  - Pay as you go stack vs 1mb stack
  - ~200ns vs 1-10us (context switching)

Virtual threads

```
Thread virtualThread1 = Thread.startVirtualThread(() -> {
 //task longa
});

Thread virtualThread2 = Thread.builder().virtual().task(() -> {
 //task longa com blocking I/O
}).build();
virtualThread2.start()
```

DEMO TIME

DEMO TIME

Lembram do nosso exemplo?

```
public void process(Operation op){
    databaseService.process(op);
    auditService.process(op);
    analyticsService.process(op);
    cacheService.process(op);
}
```

Structured Concurrency

- Structured concurrency possibilita desenvolvedores escreverem código concorrente num bloco de código visível
- Código parece síncrono, mas é assíncrono
- Todas as tasks são finalizadas depois de terminar o bloco de código
- Futuro de todas as APIs Java

```
try (var executor = Executors.newVirtualThreadExecutor()) {
        executor.submit(() -> databaseService.process(op));
        executor.submit(() -> auditService.process(op));
        executor.submit(() -> analyticsService.process(op));
        // for loop pra criar 'n'
        executor.submit(() -> cacheService.process(op));
}
```

Thank you

Eder Ignatowicz

Principal Software Engineer

@ederign

William Siqueira

Senior Software Engineer

@William_Antonio


linkedin.com/company/red-hat


youtube.com/user/RedHatVideos


facebook.com/redhatinc


twitter.com/RedHat

Red Hat

Thread pooling

Effective Java, 2nd Edition, by Joshua Bloch, chapter 10, Item 68 :

"Choosing the executor service for a particular application can be tricky.

If you're writing a small program, or a lightly loaded server, using Executors.new- CachedThreadPool is generally a good choice, as it demands no configuration and generally "does the right thing." But a cached thread pool is not a good choice for a heavily loaded production server!

In a cached thread pool, submitted tasks are not queued but immediately handed off to a thread for execution. If no threads are available, a new one is created. If a server is so heavily loaded that all of its CPUs are fully utilized, and more tasks arrive, more threads will be created, which will only make matters worse.

Therefore, in a heavily loaded production server, you are much better off using Executors.newFixedThreadPool, which gives you a pool with a fixed number of threads, or using the ThreadPoolExecutor class directly, for maximum control."

Limitação

- Blocking while holding monitors (pinned threads)
  - java.util.lock.* is safe to use
- Blocking while doing JNI blocking

- We’ve seen these promises before. People have been trying to implement “transparent” RPC (Remote Procedure Calls) multiple times. As Martin Kleppmann writes in his book “Designing Data-Intensive Applications”:
- The RPC model tries to make a request to a remote network service look the same as calling a function or method in your programming language, within the same process (this abstraction is called location transparency). Although RPC seems convenient at first, the approach is fundamentally flawed. A network request is very different from a local function call: (...)
-