Java threads are losing weight
Project Loom

Sergey Kuksenko

Java Platform Group
Oracle
July, 2020

Safe Harbor

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

Statements in this presentation relating to Oracle's future plans, expectations, beliefs, intentions and prospects are "forward-looking statements" and are subject to material risks and uncertainties. A detailed discussion of these factors and other risks that affect our business is contained in Oracle's Securities and Exchange Commission (SEC) filings, including our most recent reports on Form 10-K and Form 10-Q under the heading "Risk Factors." These filings are available on the SEC's website or on Oracle's website at http://www.oracle.com/investor. All information in this presentation is current as of September 2019 and Oracle undertakes no duty to update any statement in light of new information or future events.

Who am I?

- Java/JVM Performance Engineer at Oracle, @since 2010
- Java/JVM Performance Engineer, @since 2005
- Java/JVM Engineer, @since 1996

Java and Threads

Close Encounters of the 7th Kind

25 years ago

Java was the first language with threads

25 years ago

Java was the first* language** with threads***

* - first-ish

** - widely used language

*** - threads as a part of the language, not a library

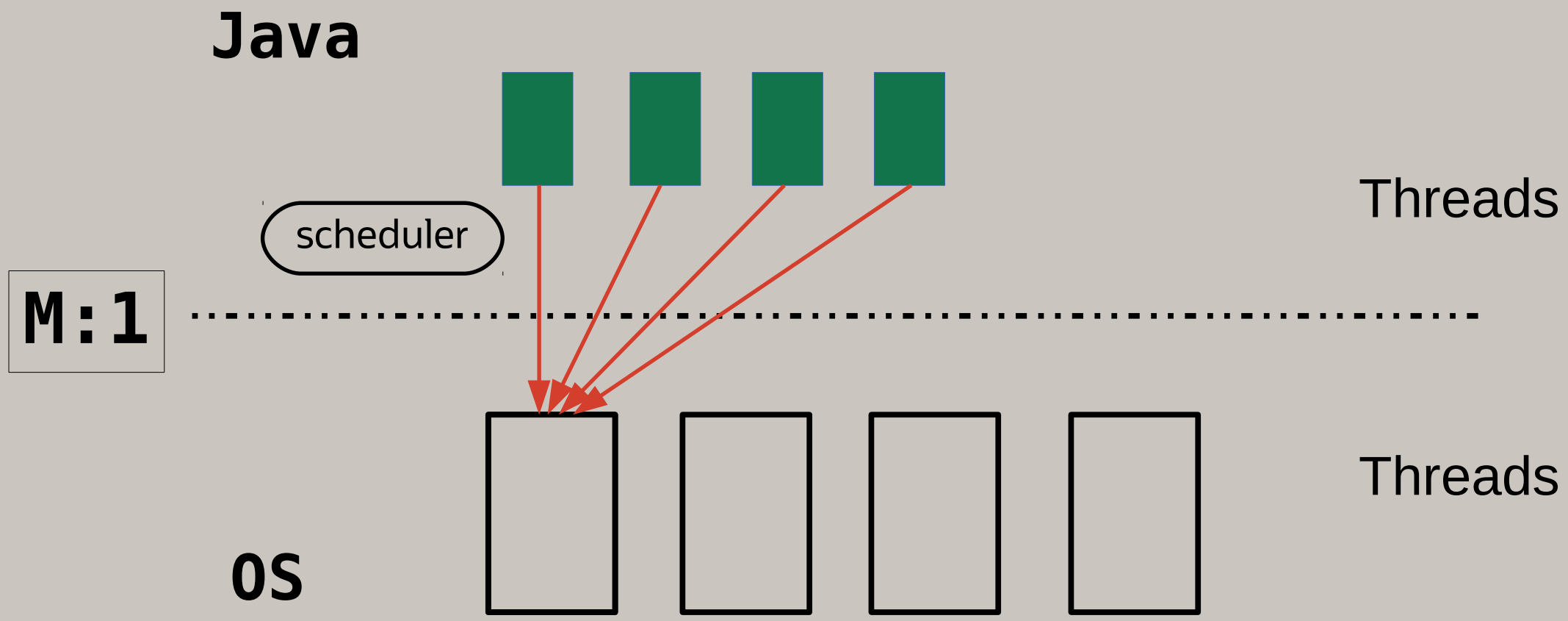
Java threads

- `java.lang.Thread`
  - easy to use
  - platform/OS/HW independent
- *Bring concurrency to the masses*

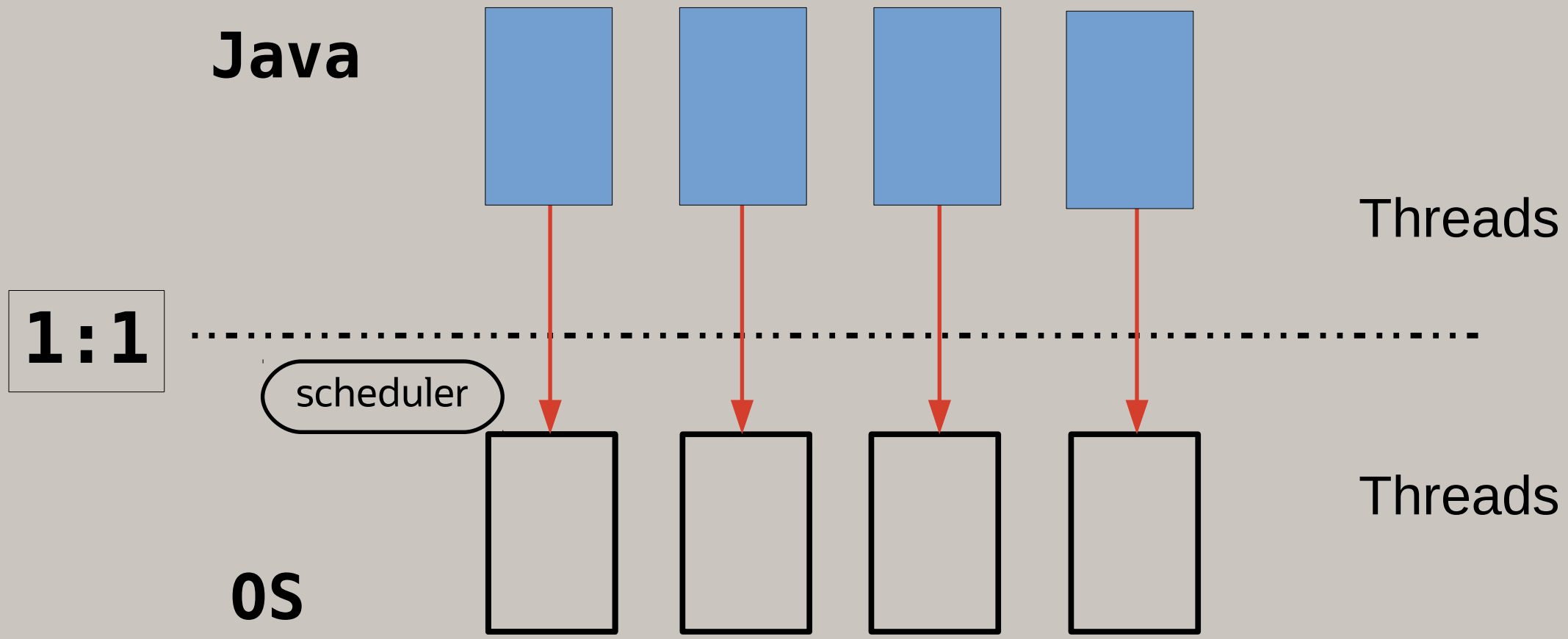
Java threads

- `java.lang.Thread`
  - easy to use
  - platform/OS/HW independent
- *Bring concurrency to the masses*
- The devil is in the ~~detail~~ implementation

Green Threads (Java childhood)



Native Threads



At the turn of the millennium

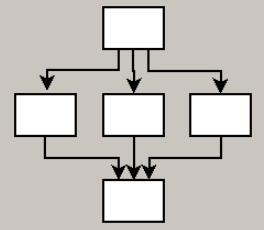
- Green threads are dead
- Java thread is still OS independent abstraction
- Nobody separates Java and OS threads anymore*

** - sometimes abstractions may leak in minds*

Threads are expensive

Expensive to start

- Do nothing* 1024 times



	time, ns
sequentially	3,349
in threads	89,154,422

```
* - JMH's Blackhole.consumeCPU(0)
```

Expensive to start

- Fixed @since Java 1.5
  - Thread pools, Executor, ExecutorService
  - Tasks, Callable<>
  - java.util.concurrent – new collections, locks

Context switch

- Going to kernel
- Cost in tens microseconds

Memory-heavy

- Typical stack size – 1M
- + ~16K of native memory
- + ~1K of Java heap
- + some Java features
    - e.g. ThreadLocals, GC's TLAB

Caches, NUMA

- Kernel scheduler is trying to be good for everyone
  - Bad cache locality
  - Bad NUMA placement

What we've got

- Threads are expensive - can't have millions of them
- Threads are mostly idle (blocked)
    - Our systems are still underutilized
    - Not scalable

The raise of async
or
We need a chopper

Chopper

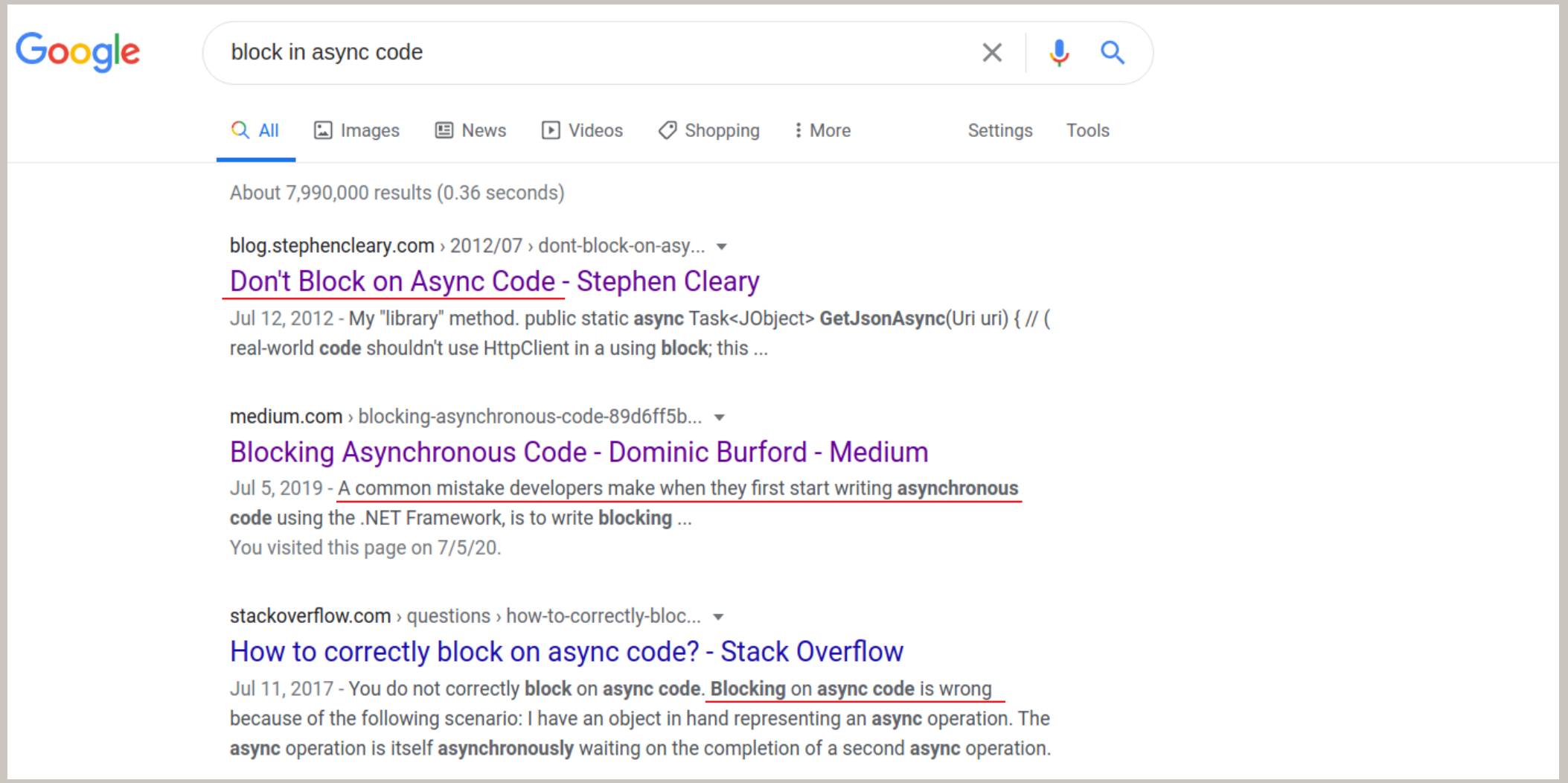
Myriads of them

- Async callbacks
- Promises (e.g. `CompletableFuture`)
- async/await
- Suspendable functions
- Reactive systems
- . . .

The key idea

- Split execution to many pieces
- Evenly distribute to limited amount of threads

Ask Google



The key idea

- Split execution to many **non blocking** pieces
- Evenly distribute to limited amount of threads

Async

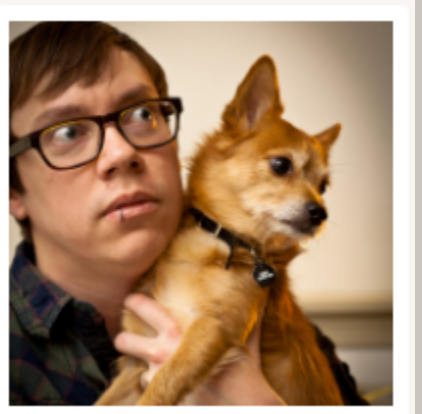
- Inventing new concepts (async) in attempts to solve implementation deficiencies.

Async issues

- Hard to find proper cut places
- Still not cache friendly
- Hard to write and understand code
- Debugging?
- …

Function coloring

What Color is Your Function?



http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

TPC vs TPC

Thread Per Connection	**OR**	Thread Per Core
Developer productivity		System productivity

Some ideas

Expensive Worker

- Given: expensive worker is sitting idle
- Find: how to make the worker useful
- Solution:
    - function to save/restore task state(context)
    - some sort of manager (scheduler)

Expensive Worker

Multythreading

- Worker: CPU
- State: stack
- Function: context switch
- OS Scheduler

Expensive Worker

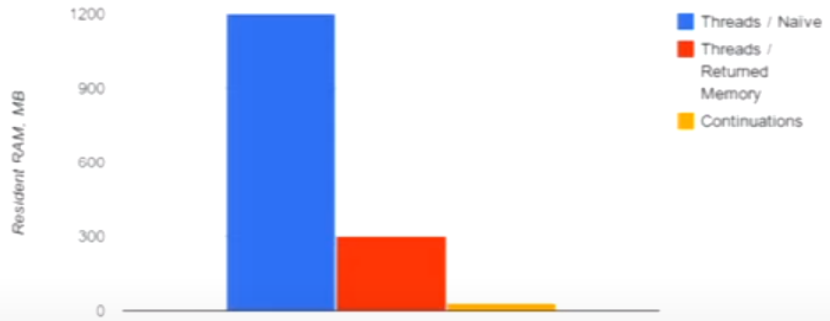
Lightweight/user-level threads

- Worker: OS thread
- State: continuation
- Function: freeze/thaw
- User level scheduler

Ask Google

Google

Hmm… What **about** thread stacks?



- Return RAM to system forcibly
- 10K threads; 1 thread-per-core, 10K continuations
- Okay, but still not great

Ask Parallel Universe

JVM Continuations with Bytecode Instrumentation

```
bar() {
   int pc = isFiber ? s.pc : 0;
   switch(pc) {
   case 0:
     baz();
     if(isFiber) {
       s.pc = 1;
       // store locals -> s
     }
   case 1:
     if(isFiber)
       // load locals <- s
     foo(); // sus
  }
}
```

```
foo() {
   int pc = isFiber ? s.pc: 0;
   switch(pc) {
     …
     if(isFiber) {
       s.pc = 3;
       // store locals -> s
     }
     Fiber.park(); // thrw SE
   case 3:
     if(isFiber)
       // load locals <- s
     …
   }
}
```

Project Loom

Project Loom

- Lightweight threads
- Continuations
- Tail-call recursion elimination

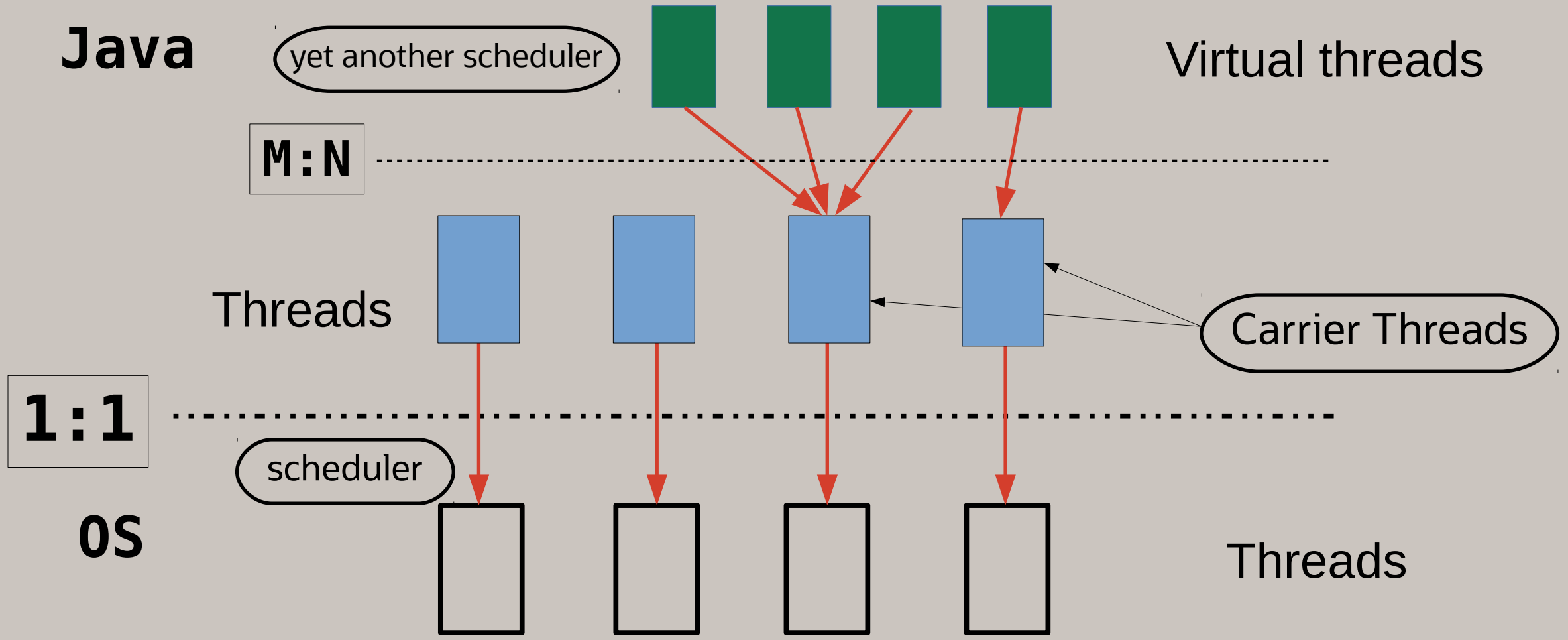
Lightweight thread

```
public class Thread {
    ...
    /**
     * Returns true if this thread scheduled by the Java
     * virtual machine rather than the operating system.
     *
     * @since Loom
     */
    public boolean isVirtual()

    ...
}

class VirtualThread extends Thread { ... }
```

Virtual Threads



Threads

- Virtual threads are threads
  - no need to rewrite code if you want it
- Non virtual threads are not changed
  - still 1:1 mapping to OS threads
  - no need to rewrite code if you don’t want it

Threads

- *Carrier threads* (non-virtual)
  - carry the virtual threads on their backs
- Scheduler:
  - *mount* virtual thread to the carrier
  - *unmount* virtual thread from the carrier

Threads

```
Thread t = Thread.builder().virtual().task(() -> {...}).start();
    ...

Thread t = Thread.builder().virtual().task(() -> {...}).build();
    ...

ThreadFactory tf = Thread.builder().virtual().factory();
    ...

ExecutorService e = Executors.newVirtualThreadExecutor();
```


Scheduler

- Use any executor as scheduler
  - `Thread.builder().virtual(scheduler)...`
- `ForkJoinPool` by default

Scheduling

- Virtual threads are preemptive, not cooperative
  - No explicit yield operation
- Preemption points:
  - I/O blocking
  - synchronization blocking

Scheduling

- Forced preemption (time slice)
  - any thread may be stopped at safepoint
  - not implemented now
  - maybe in a future

Continuation

Continuation

From Wikipedia, the free encyclopedia

In computer science, a **continuation** is an abstract representation of the control state of a computer program. A continuation implements (reifies) the program control state, i.e. the continuation is a data structure that represents the computational process at a given point in the process's execution; the created data structure can be accessed by the programming language, instead of being hidden in the runtime environment. Continuations are useful for encoding other control mechanisms in programming languages such as exceptions, generators, coroutines, and so on.

Continuation in Java

- State a.k.a. stack of the virtual thread

Where to store?

- On thread stack? Really?
- Java heap? Expensive.
- Off-heap? Need to tame GC. Too complex.
- Copying!

Where to store?

- Mounted virtual thread:
  - Use OS thread stack
- Unmounted virtual thread:
  - Copied to Java heap
  - Lazy-copying
  - Chunked copying
  - etc…

Performance is good, but there are places for improvement

Continuations

- Interesting usages (not implemented yet):
  - cloning
  - serialization
  - etc.

Continuations

- Interesting usages (not implemented yet):
  - cloning
  - serialization
  - etc.
- It’s not a goal to expose Continuation API

Some implementation details

Fight for memory

- Typical continuation size ~200-~1000 bytes
- `j.l.Thread` size optimization – now 350-400 bytes

Fight for memory

- `ThreadLocal<T>`
  - designed for rare and exclusive usage
  - pervasive usage over classlibs and frameworks
  - typical source of memory leaks

Fight for memory

- `ThreadLocal<T>`
  - cleaning classlibs (get rid of ThreadLocals)
  - `Thread.builder().disallowThreadLocals()`

Pinning

- Virtual thread may be *pinned* to the carrier
- Pinned thread can't be unmounted
  - Pinned thread can be diagnosed
  - Will be JFR events

Native code

- Native stack frame – thread is pinned

The curse of two locks

Object monitor	java.util.concurrent.locks
@since 1.0	@since 1.5
synchronized(){...} wait(), notify() ...	ReentrantLock, ReadWriteLock, ...
BiasedLocking, thin/fat locks, adaptive spinning	tryLock, fairness

The curse of two locks

Object monitor	java.util.concurrent.locks
The art of assembler Part of runtime	The art of simplicity Built on: CAS, park, unpark Everything else – on Java

The curse of two locks

Object monitor	java.util.concurrent.locks
Large refactoring is required Not yet implemented Virtual threads are pinned	**Loom friendly**

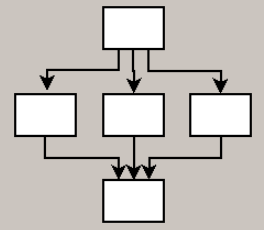
Classlibrary

- Make blocking I/O API Loom friendly
- Migration from Object monitors to j.u.c.locks
- ThreadLocal cleaning
- …

Loom performance

Start cost

- Do nothing* 1024 times

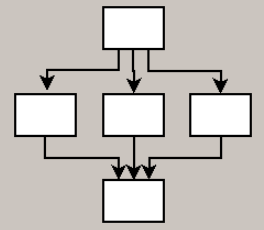


	time, ns
sequentially	3,349
in threads	89,154,422
in virtual threads	1,591,256

```
* - JMH's Blackhole.consumeCPU(0)
```

Start cost

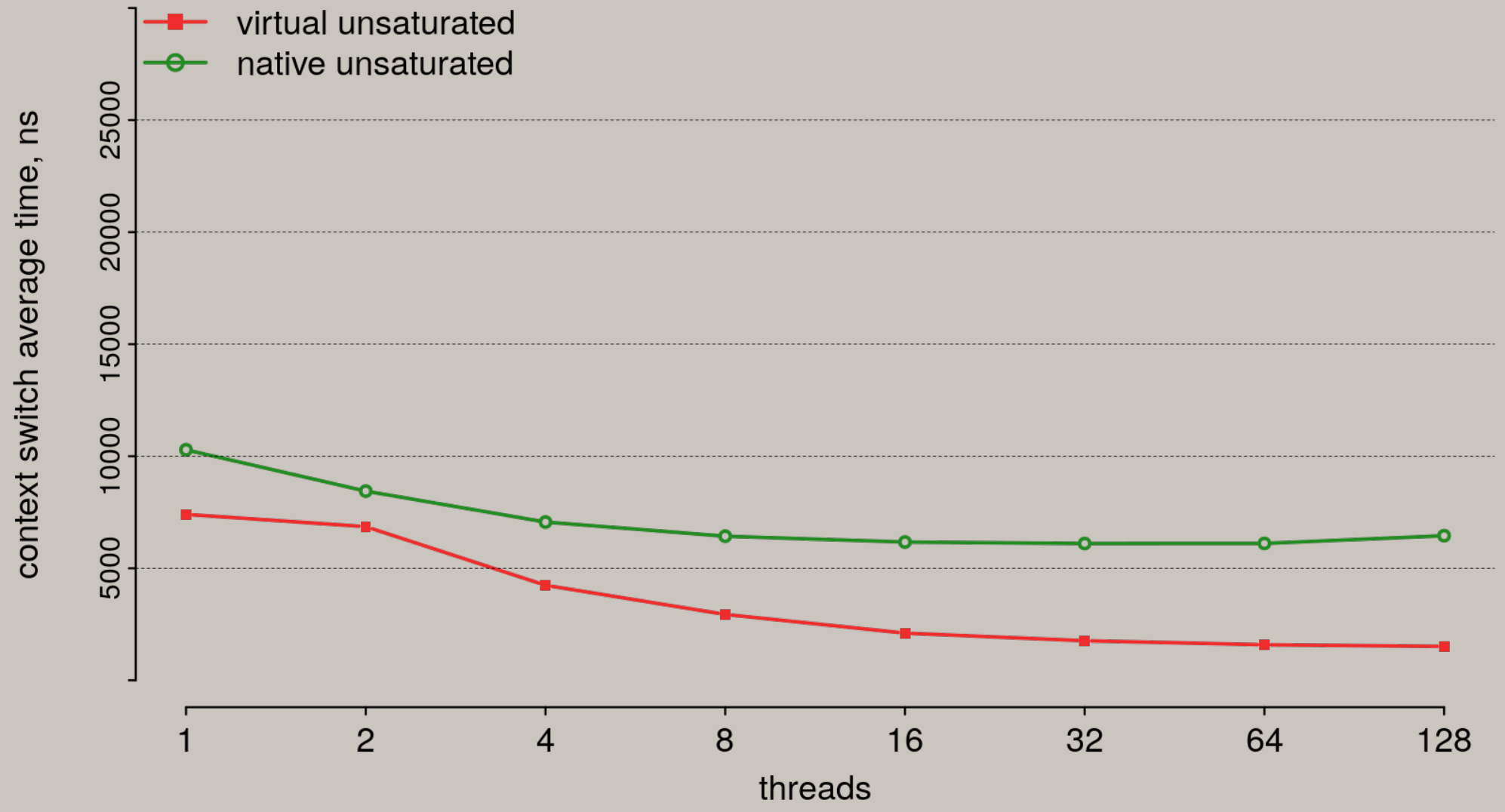
- Do nothing* 1000000 times



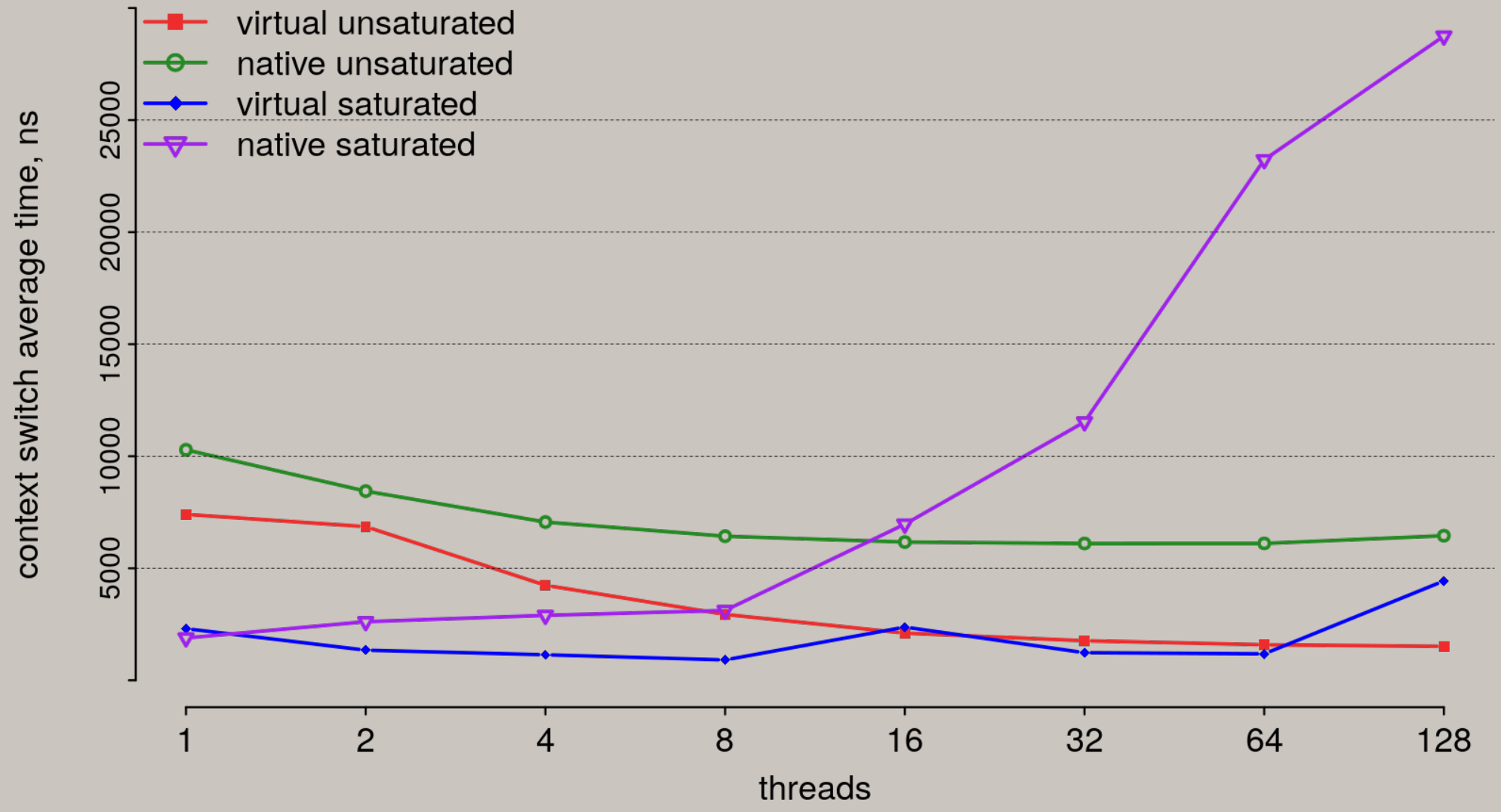
	time, ms
sequentially	3,2
in threads	`OutOfMemoryException`
in virtual threads	1104.7

```
* - JMH's Blackhole.consumeCPU(0)
```

Context switch cost



Context switch cost



What about latency?

ORACLE

Example with existing code/libraries

- Assume servlet or REST service that spends a long time waiting

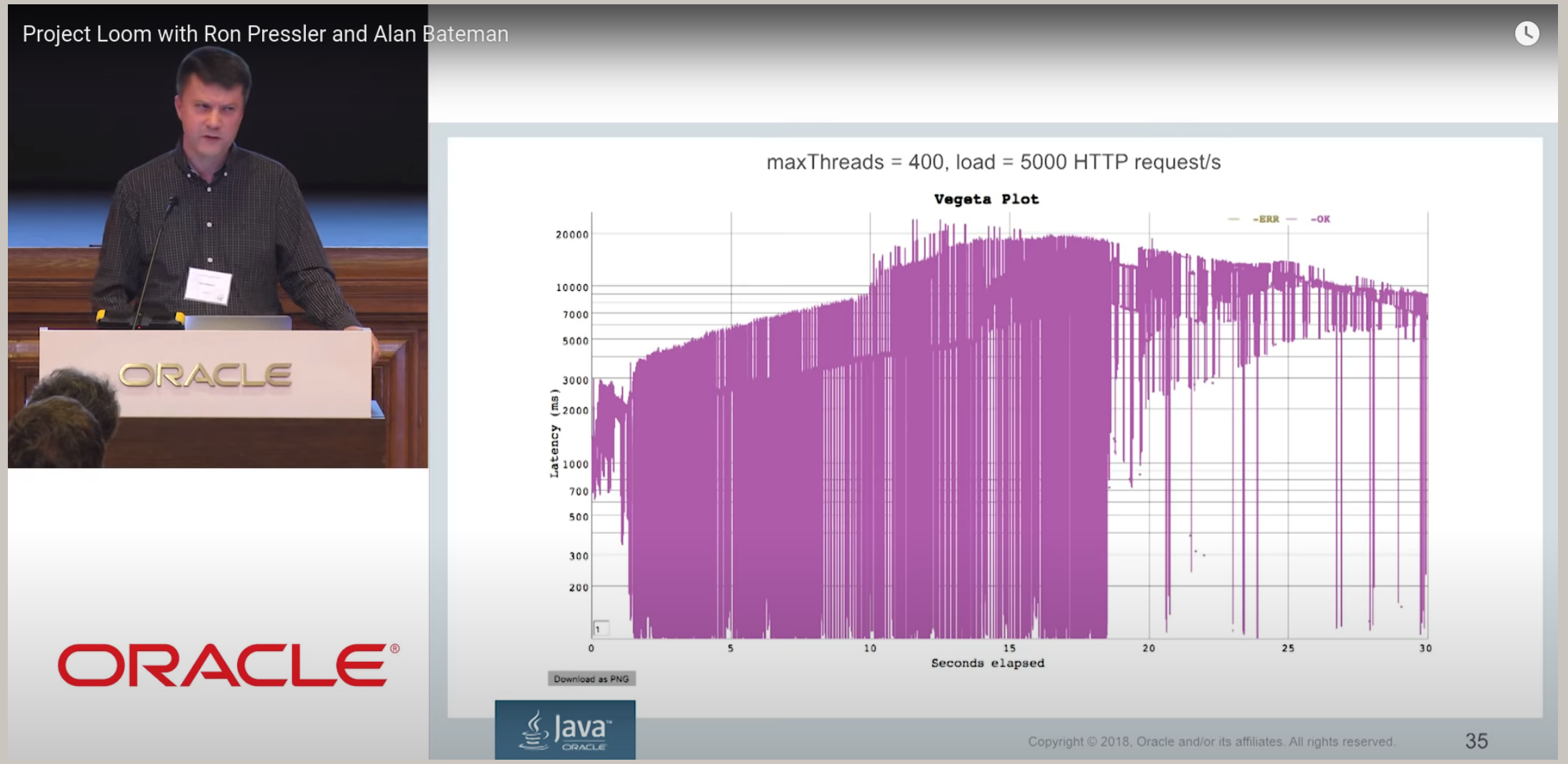
```
@GET
@Path("greeting")
@Produces(MediaType.APPLICATION_JSON)
public String greeting() {
    return "{ \"message\": \"" + computeValue() + "\" }";
}
```

assume this takes 100ms

Java ORACLE

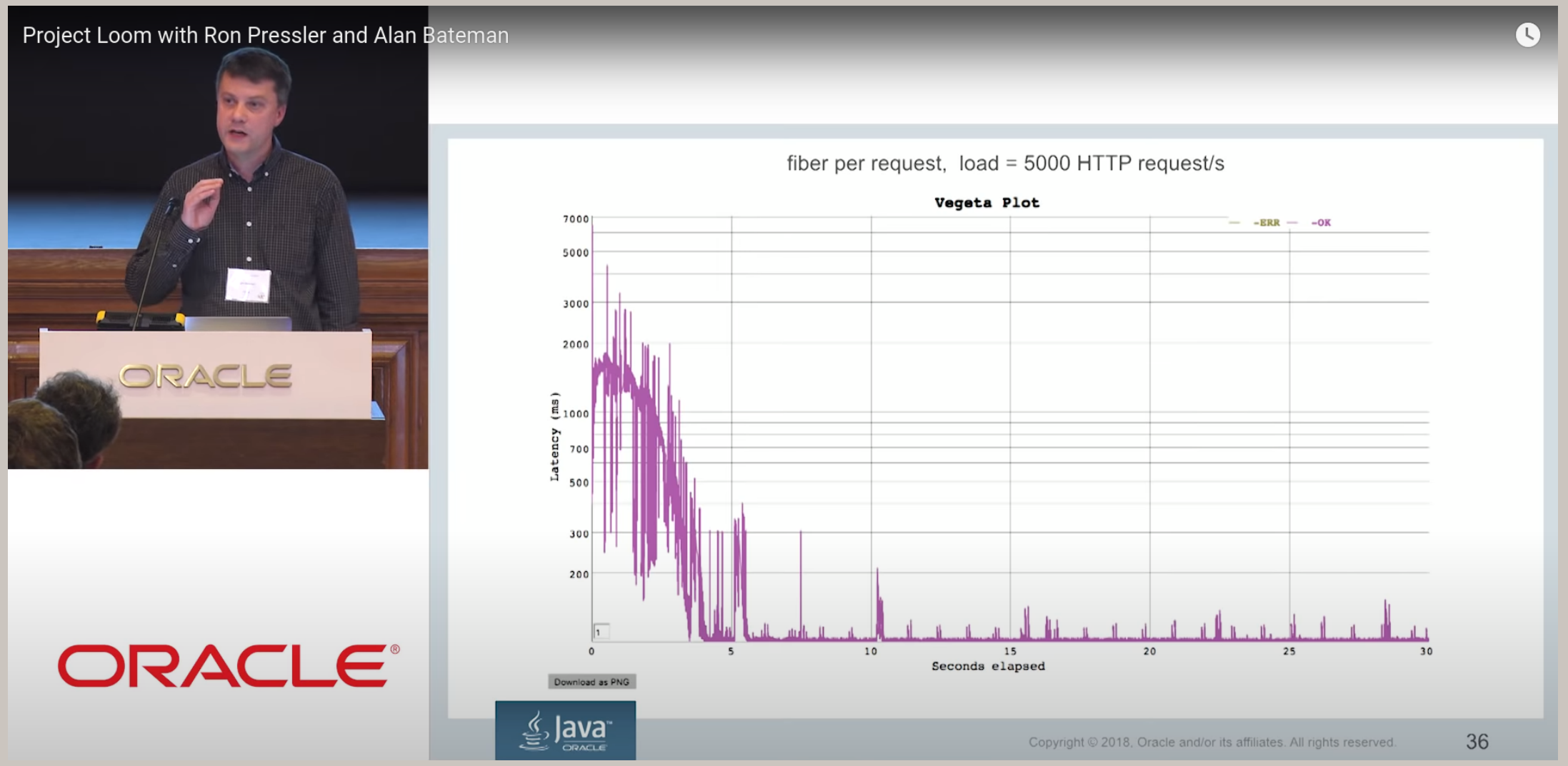
33

What about latency?

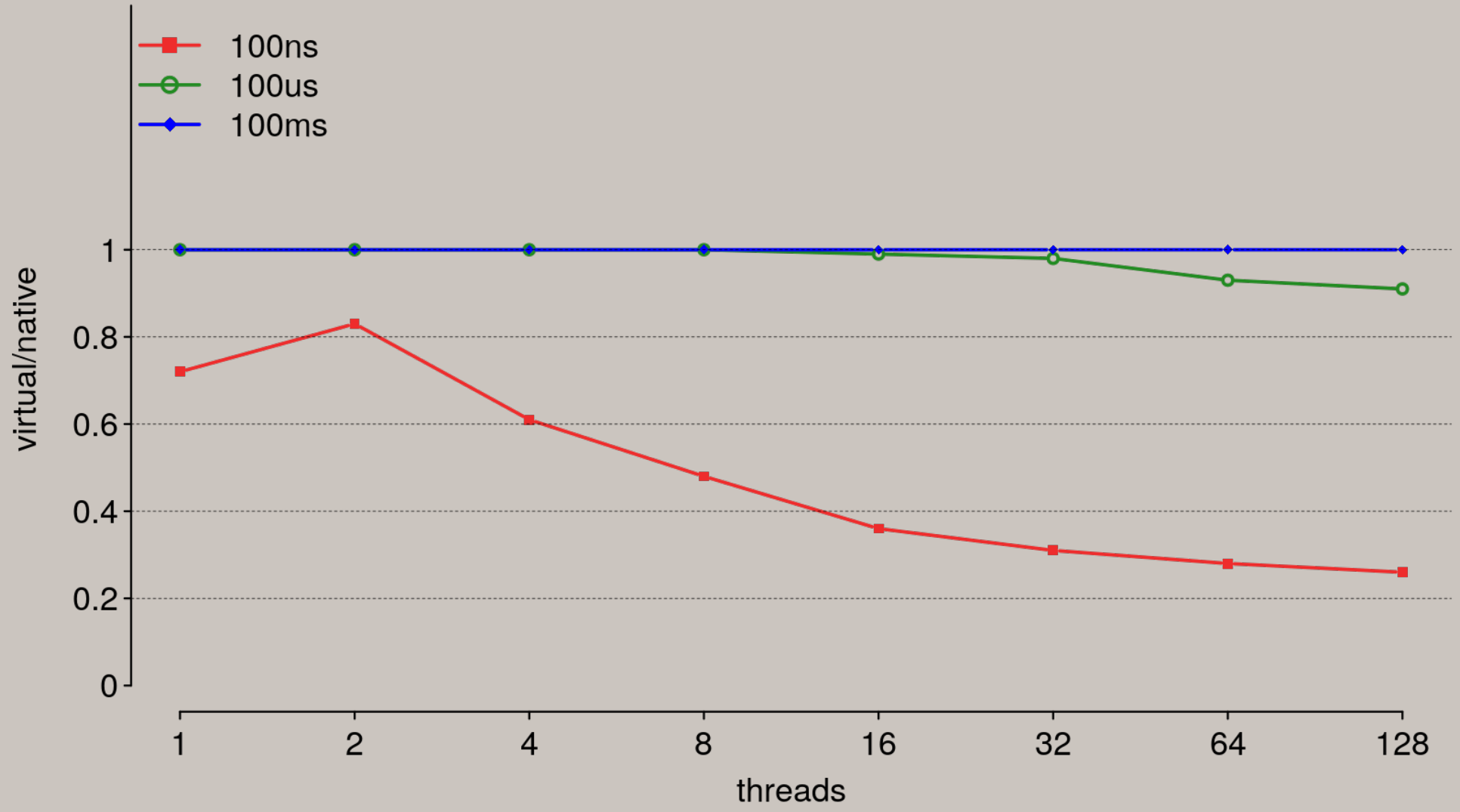


ORACLE®

What about latency?



CPU intensive computations



TPC + TPC

Virtual Thread Per Connection

AND

Native Thread Per Core

Developer productivity

System productivity

Beyond the scope of this talk

IdontWantToTalkAboutItYetException

- **Channels**
- **Structured Concurrency**
- **Scope Variables**
- **Processor Locals**
- **Timeouts and cancellation**
- **…**

Links

- Wiki:

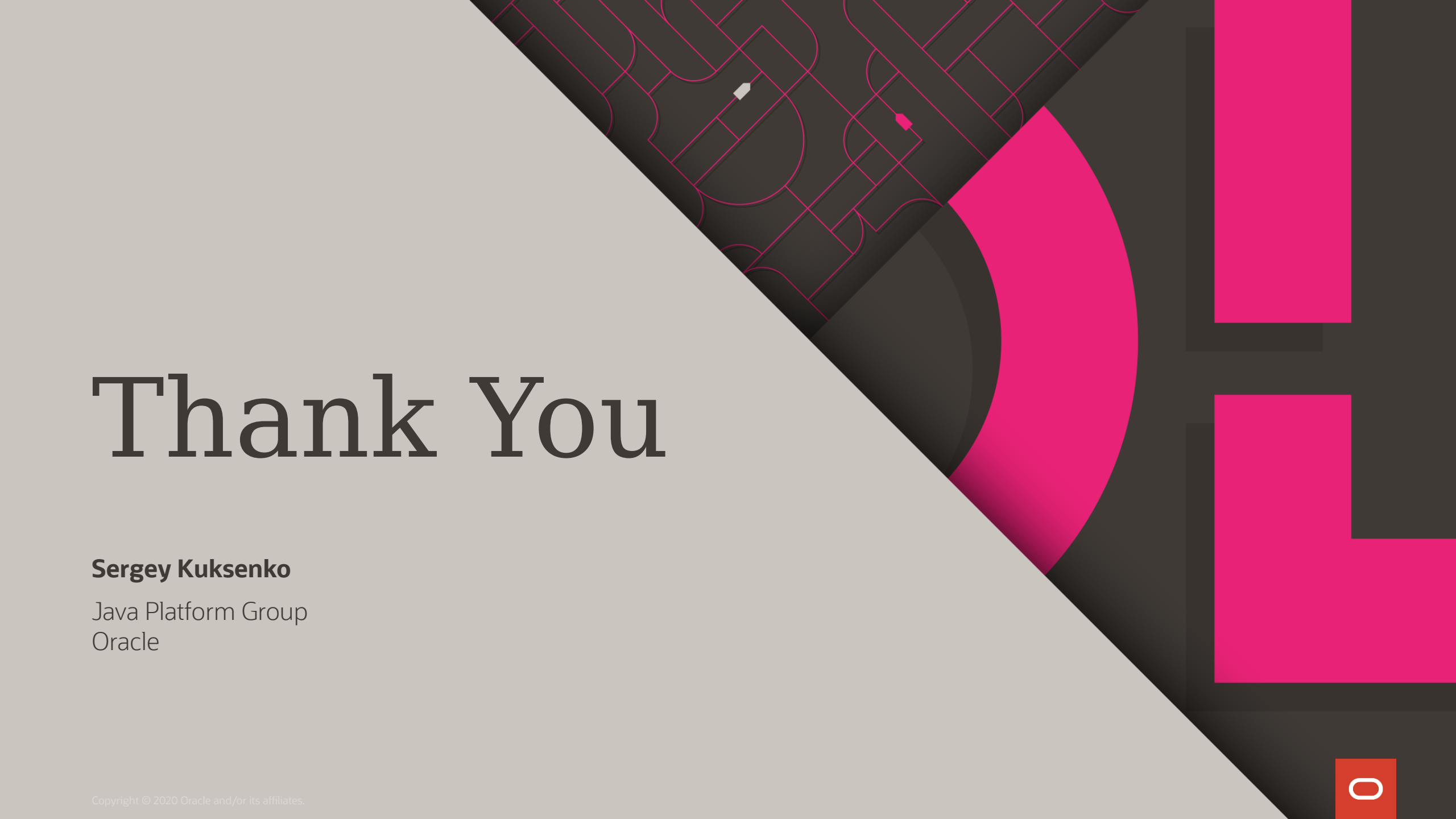
  https://wiki.openjdk.java.net/display/loom/Main

- Mailing lists:

  http://mail.openjdk.java.net/pipermail/loom-dev/

- Repository:

  https://github.com/openjdk/loom



Thank You

Sergey Kuksenko

Java Platform Group
Oracle