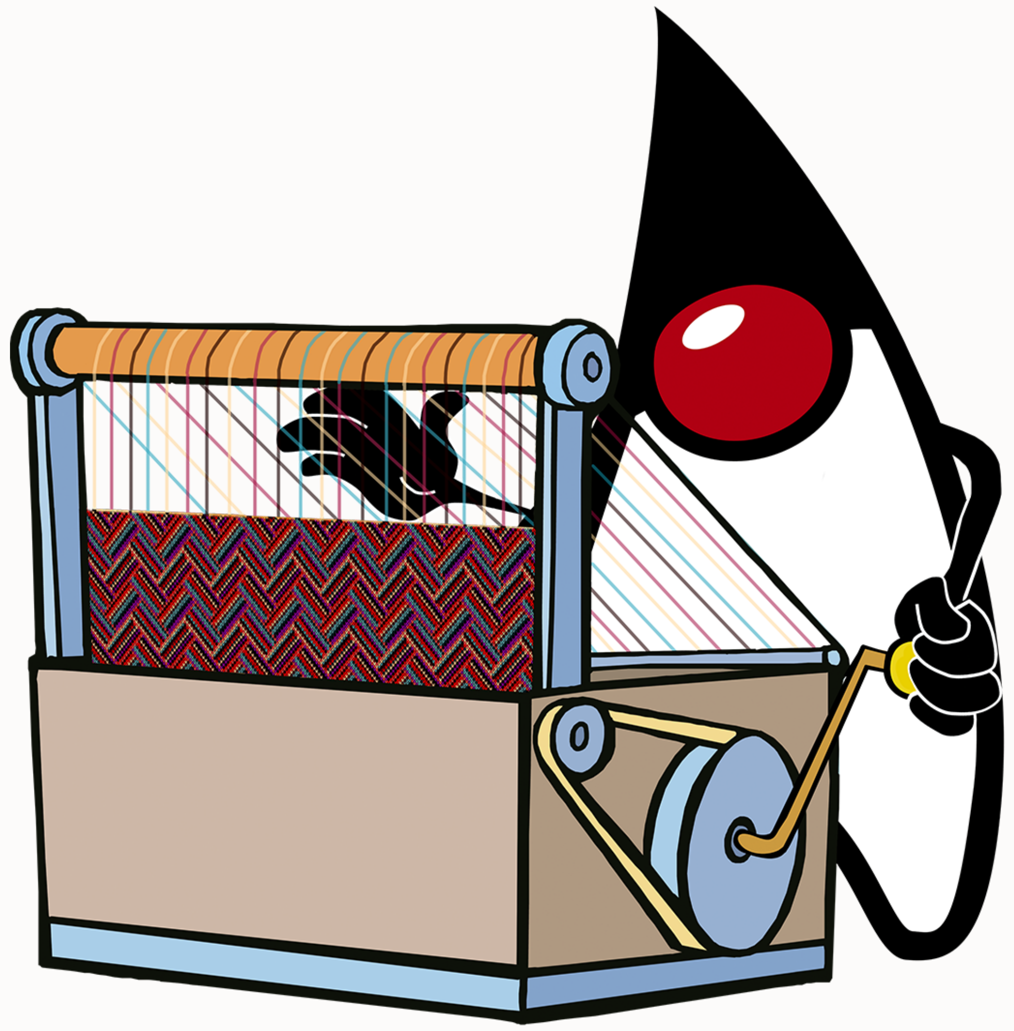


Project Loom

The challenges of introducing Virtual Threads to the Java Platform

Name

Alan Bateman

JVMLS 2023

JVMLS 2019

What is a fiber?

- A *light weight* or *user mode thread*, scheduled by the Java virtual machine, not the operating system
- Fibers are intented to have very low footprint and have negligible task-switching overhead. You can have millions of them!

Why fibers?

- The runtime is well positioned to manage and schedule application threads, esp. when they interleave computation and I/O and interact very often (exactly how many server threads behave)
- Fibers allow developers to write simple synchronous/blocking code that is easy read, maintain, debug and profile, yet scales
- Project mantra: *Make concurrency simple again*!

How are fibers implemented?

- Built on Continuations, implemented in the HotSpot VM, as a lower level construct
- A Continuation (precisely: delimited continuation) is a program object representing a computation that may be suspended and resumed
- TBD on whether API to Continuations will be exposed

How are fibers implemented?

- Fibers = continuation + scheduler
- A fiber wraps a task in a continuation
  - The continuation yields when the task needs to block
  - The continuation is continued when the task is ready to continue
- Scheduler executes tasks on a pool of *carrier* threads
  - java.util.concurrent.Executor in the current prototype
  - Default/built-in scheduler is a ForkJoinPool

Thread vs. Fiber

- Mental model: fiber is a thread
- Is java.lang.Thread the right API for fibers?
  - java.lang.Thread has accumulated a lot of baggage over 20+ years
  - Defines 13 static and 29 instance methods, many are not interesting
- Thread footprint is significant
- Thread locals add to footprint

Thread vs. Fiber

- Trying to "re-imagine threads", leave the java.lang.Thread baggage behind
- Prototypes to date:
  - Fiber <: Thread
  - Fiber <: Strand, Thread <: Strand
  - Fiber and Thread without a common super type
- Current status

What about Thread.currentThread() ?

- A lot of existing code makes direct or indirect use of Thread.currentThread()
- For now, first use of Thread.*currentThread()* in context of a fiber creates an adaptor
  - a.k.a. the "Shadow Thread", Thread object, no stack or VM meta data
  - Emulates legacy Thread API, everything except *stop*, *suspend*, and *resume*
  - Everything *thread local* is *fiber local*, the cost is footprint
  - Pragmatic solution to keep existing code working
- Thread object for the transient *carrier thread* is never leaked to user code

Fast forward to today

- Virtual thread = user mode thread, scheduled by the Java virtual machine, not the operating system

- A *virtual thread* is an instance of `java.lang.Thread`
  - Not tied to a particular OS thread

- A *platform thread* is an instance of `java.lang.Thread` but implemented in the "traditional way"
  - Typically a thin wrapper around an OS thread

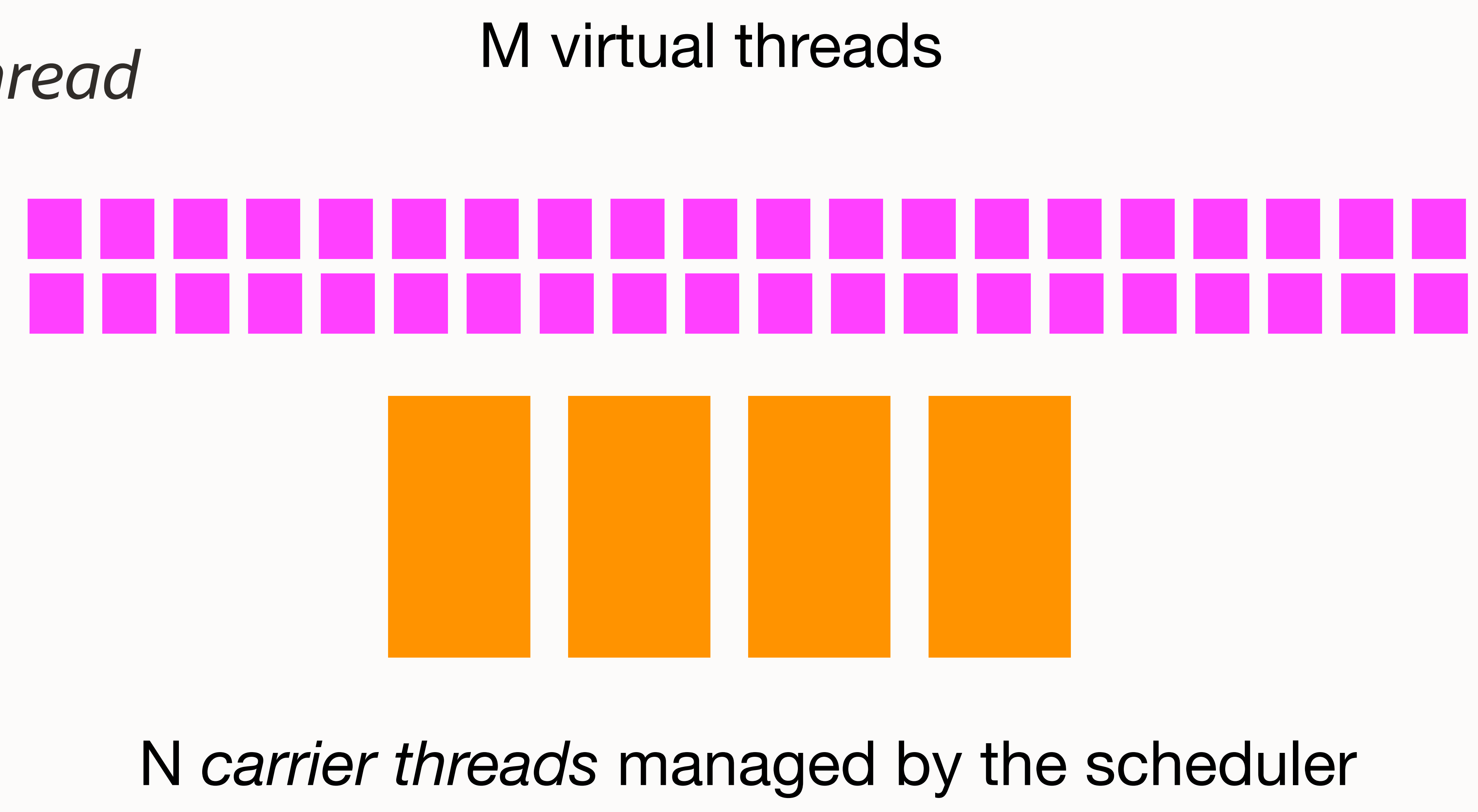
- Summary
  - `java.lang.Thread` is the API for all threads
  - `Thread.currentThread()` returns the `Thread` representing the "current thread"
    - A virtual thread and its carrier are distinct `Thread` objects
  - No change to the programming model, it's the one we already know

How are virtual threads implemented?

- Built on continuations, implemented in the HotSpot VM as a lower level construct

- A virtual thread wraps a task in a continuation
  - The continuation yields when the task needs to block
  - The continuation is continued when the task is ready to continue

- Scheduler executes the tasks for virtual threads on a pool of *carrier thread*
  - M:N threading model
  - The scheduler is a `j.u.c.ForkJoinPool`
    - FIFO mode
    - Parallelism defaults to the number of hardware threads



APIs

- Developer facing APIs
  - `java.lang.Thread`
  - `java.util.concurrent.Executors`
    - `ExecutorService` implementation with a policy to create a new thread for each task
- Other APIs and exported interfaces
  - Additions to JNI, JVM TI and JDWP specs
  - Additions to JDI, JFR and `com.sun.management` APIs
  - New thread dump format and `jcmd` command
  - New JDWP agent options
  - New JFR events

Progress to date

- JDK 19
  - JEP 425: Virtual Threads (Preview)
  - JEP 428: Structured Concurrency (Incubator)
- JDK 20
  - JEP 436: Virtual Threads (Second Preview)
  - JEP 437: Structured Concurrency (Second Incubator)
  - JEP 429: Scoped Values (Incubator)
- JDK 21
  - JEP 444: Virtual Threads
  - JEP 453: Structured Concurrency (Preview)
  - JEP 446: Scoped Values (Preview)

Article Talk | Read Edit View history | Search Wikipedia

Virtual threads

From Wikipedia, the free encyclopedia

In computer programming, **virtual threads** are threads that are scheduled by a runtime library instead of natively by the underlying operating system (OS). Virtual threads allows for tens of millions of preemptive tasks and events without swapping on a 2021 consumer-grade computer.[1], compared to low thousands of operating system threads.[2] Preemptive execution[3] is important to performance gains through parallelism and fast preemptive response times for tens of millions of events. Earlier constructs that are not preemptive, such as coroutines or the largely single-threaded Node.js, introduce delays in responding to asynchronous events such as every incoming request in a server application[4]

Contents [hide]

Implementing Raft using Project Loom

Java | Scala | Zio | Distributed Systems | Distributed Consensus | raft | Project Loom

Adam Warski
30 Aug 2022. 24 minutes read

Gunnar Morling

Random Musings on All Things Software Engineering

Blog Projects Conferences Podcasts About | Search...

Loom and Thread Fairness

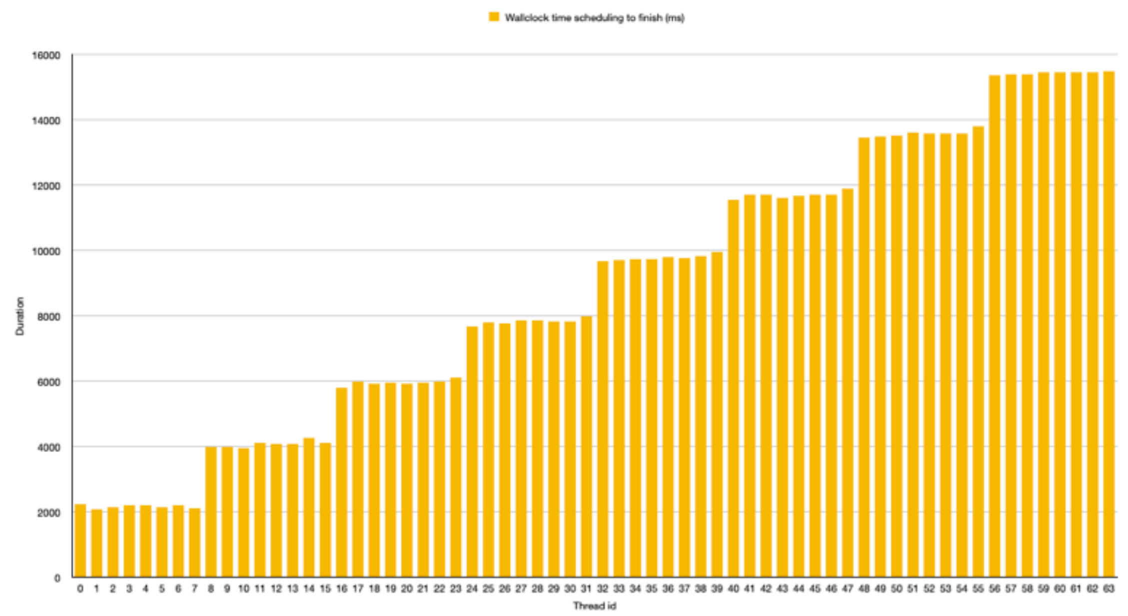
Posted at May 27, 2022

ZIO vs Loom: the verdict

Loom has the upper hand when it comes to syntax familiarity and simpler types (no viral `Future` / `IO` wrappers). ZIO, on the other hand, wins in its interruption implementation, testing capabilities, and uniformity. When it comes to concurrency, to the degree that we've been using it, there haven't been significant differences.

As far as Saft—our Scala Raft implementation—is concerned, I'd say it's a tie. I'm happy with both implementations, and they are hopefully both readable and easy to relate various implementation fragments to the Raft paper.

In wallclock time, it took all the 64 threads roughly 16 seconds to complete. The threads are rather equally scheduled between the available cores of my machine. I.e. we're observing a *fair scheduling scheme*. Now here are the results using virtual threads (by obtaining the executor via `Executors::newVirtualThreadPerTaskExecutor()`):



Tomasz Nurkiewicz

Java Champion and CTO @DevSkiller

Spent half of his life on programming, for the last decade professionally in Java land. Loves back-end and data visualization. Passionate about alternative JVM languages. Disappointed with the quality of software written these days (so often by himself!), hates long methods and hidden side...

Two takeaways, I would say. First of all, I would like people to get really excited because the project Loom may make your concurrent code much, much more readable, much easier to maintain, because you no longer have to deal with these low level details, pooling threads, making sure your queues are long enough. Tuning, monitoring, and so on. You just create treads as if it was a very native, very low footprint abstraction, which is not the case right now. The first takeaway is that this may revolutionize the way you work with concurrent code. That's why I'm excited about it. On the other hand, we can already see that even though the feature wasn't yet released, you have to be aware of the shortcomings. For example, now the garbage collector has to do much more work because the virtual threads that you can create in millions are actually subject to garbage collection, which means you will have a much harder time fine tuning garbage collectors. There's always a tradeoff. And also, there are a few other disadvantages or limitations of Project Loom that you must be aware of. Otherwise, you will just shoot yourself in the foot. There's no free lunch. I want people to get the best idea, what they can get and what are the best use cases for this new project and whether they should use it from day one, the moment it's released. Or maybe it's just a very specialized tool that they should never really look at because it's a matter of framework developers.

Using Java's Project Loom to build more reliable distributed systems

09 May 2022

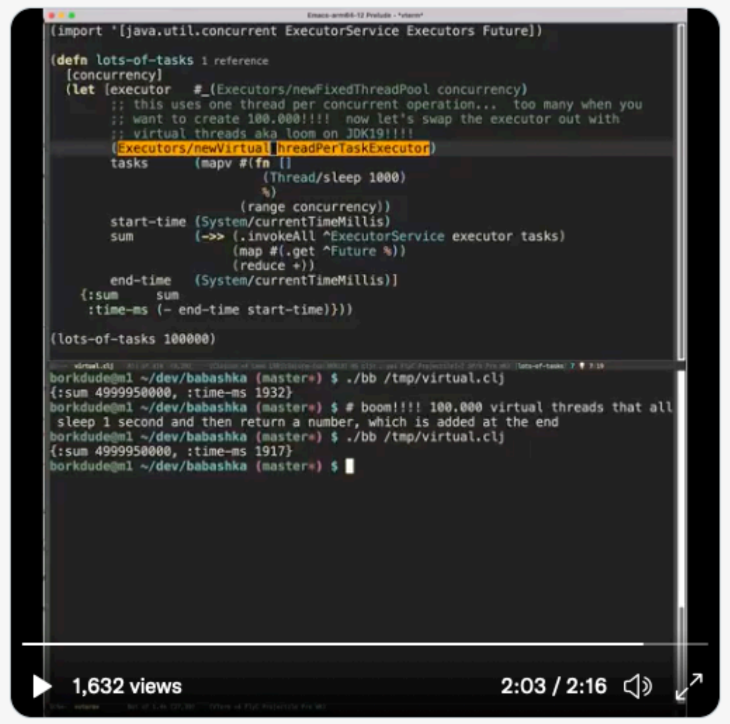
Evaluation

Simulation performance

The simulation was surprisingly performant. I have no clear comparison point, but on my computer with reasonable-looking latency configurations I was able to simulate about 40k Raft rounds per second on a single core, and 500k when running multiple simulations in parallel. This represents simulating hundreds of thousands of individual RPCs per second, and represents 2.5M Loom context switches per second on a single core.

When I cranked up the rate of timeouts and failures (leading to lots of exceptions being thrown), I saw closer to 15k requests per second processed (with about 100 leader elections during that time) and when I made performance uniformly excellent, I saw single core throughput as high as 85k Raft rounds per second.

#clojure #jdk19 #java



Gamlor's Blog | Home

August 27, 2022

Redis Clone: Improved IO Control

We left with unsuccessful attempts to improve the IO by only using Virtual Threads and the classing blocking IO classes in the previous post.

This time we are using the NIO network API to get more fined grained control to the IO pattern. As a recap: We try to avoid blocking while flushing each individual response. This way we take advantage of the pipelined requests: We get a bunch of requests from the client, answer all of them and amortize the flush over multiple responses.

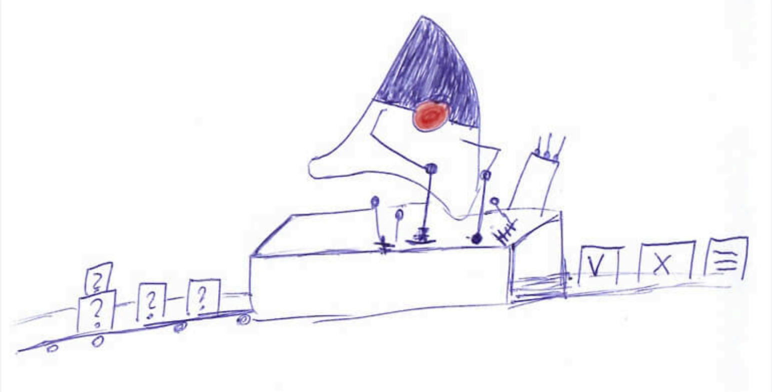
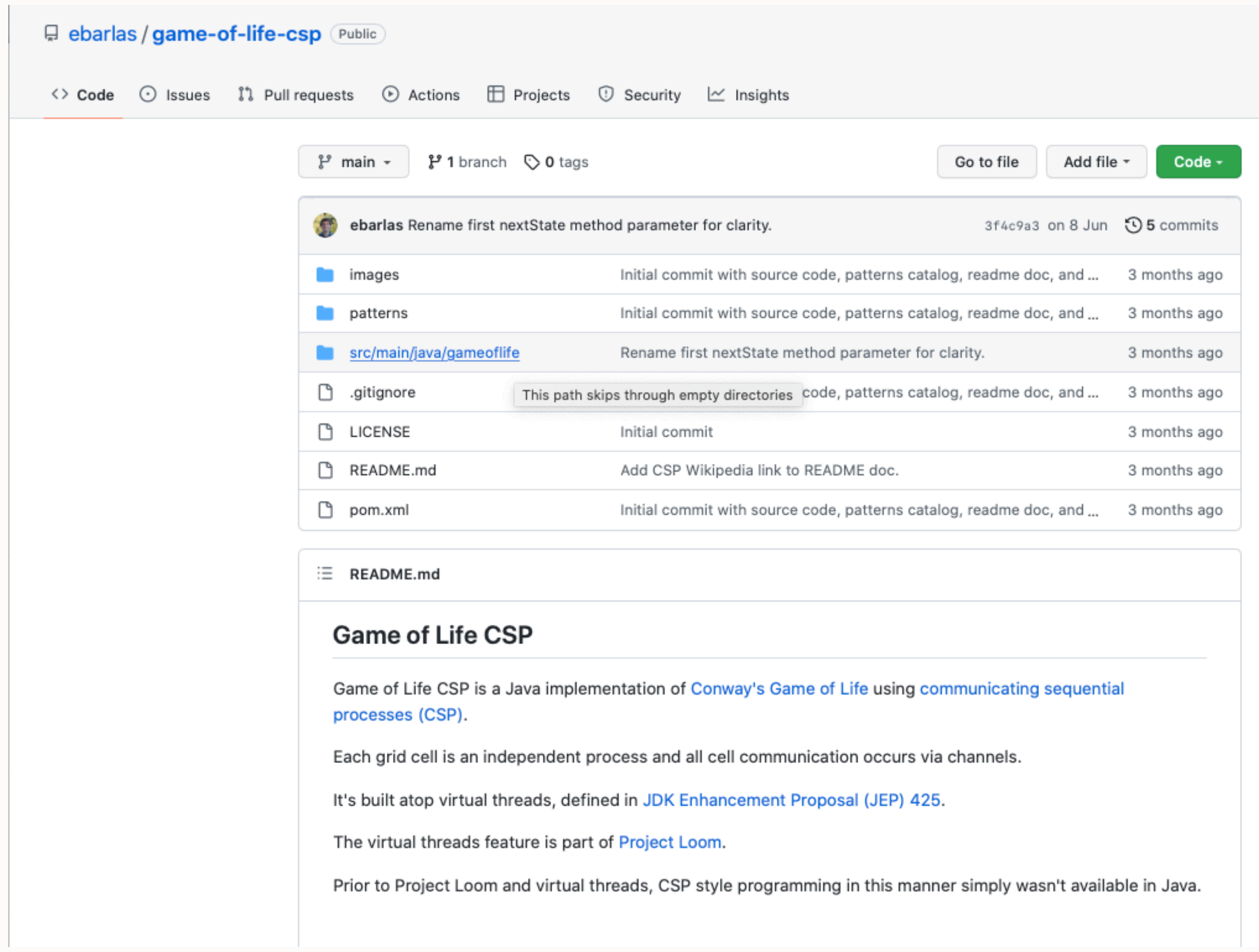
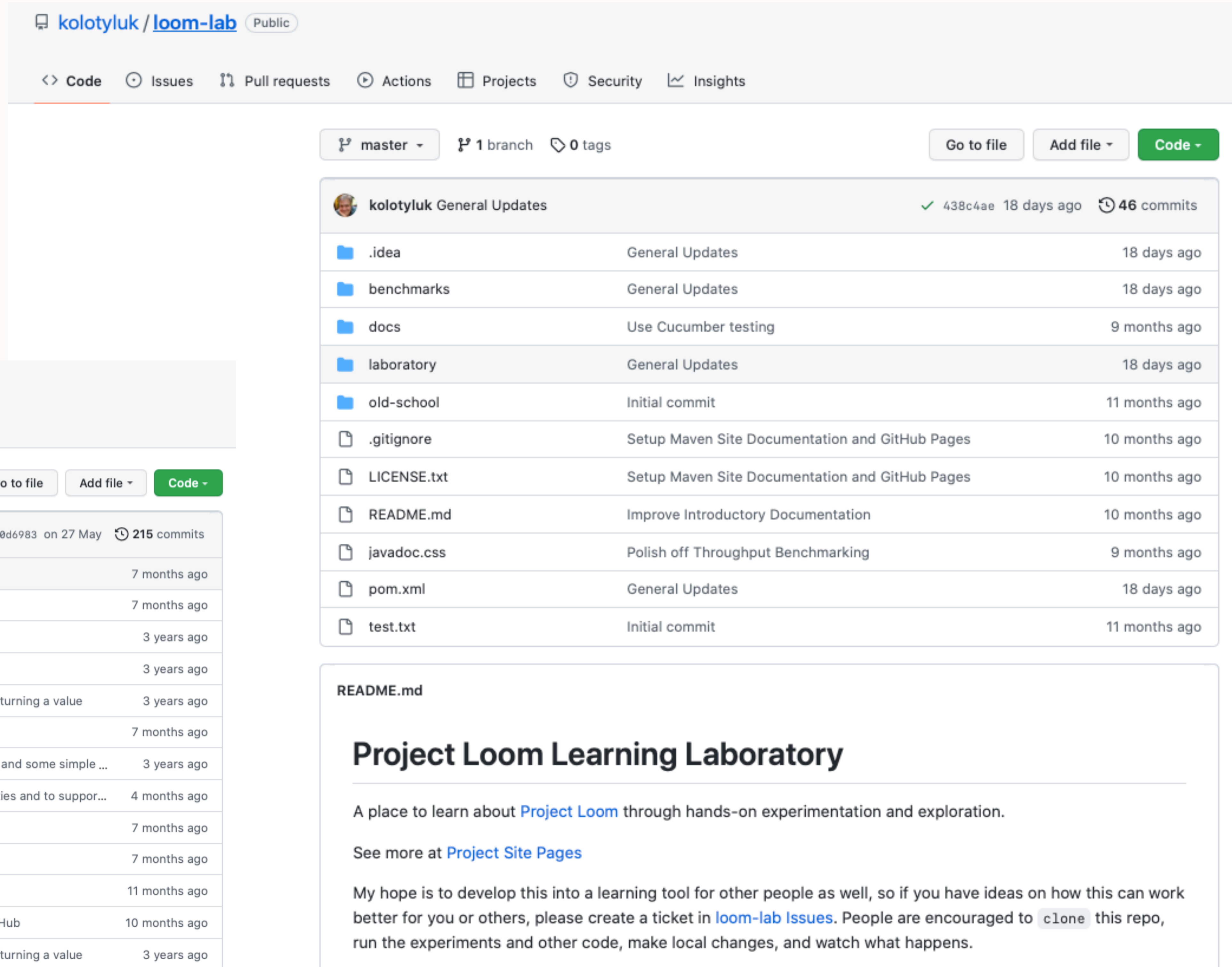


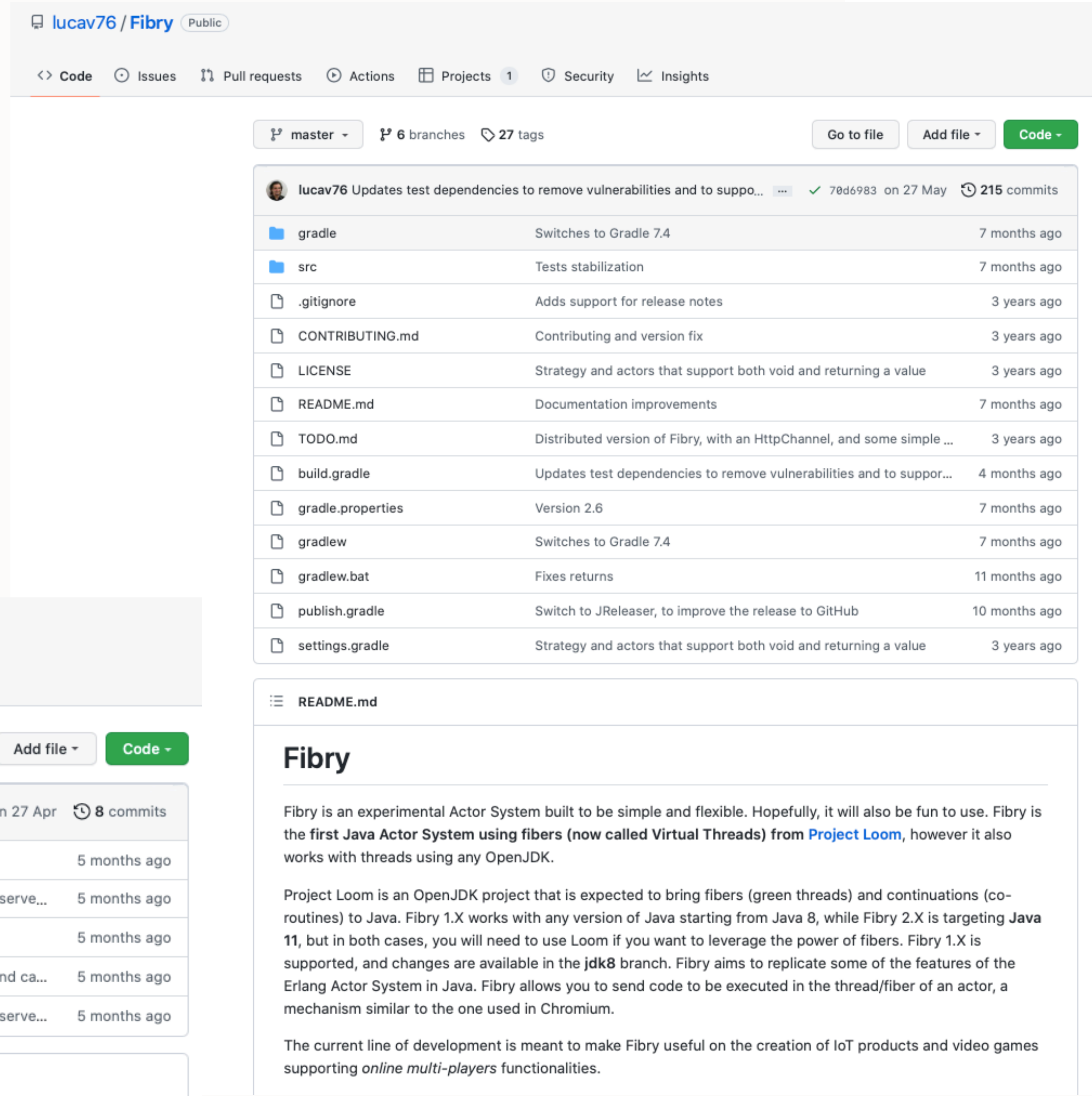
Figure 1. More IO Control

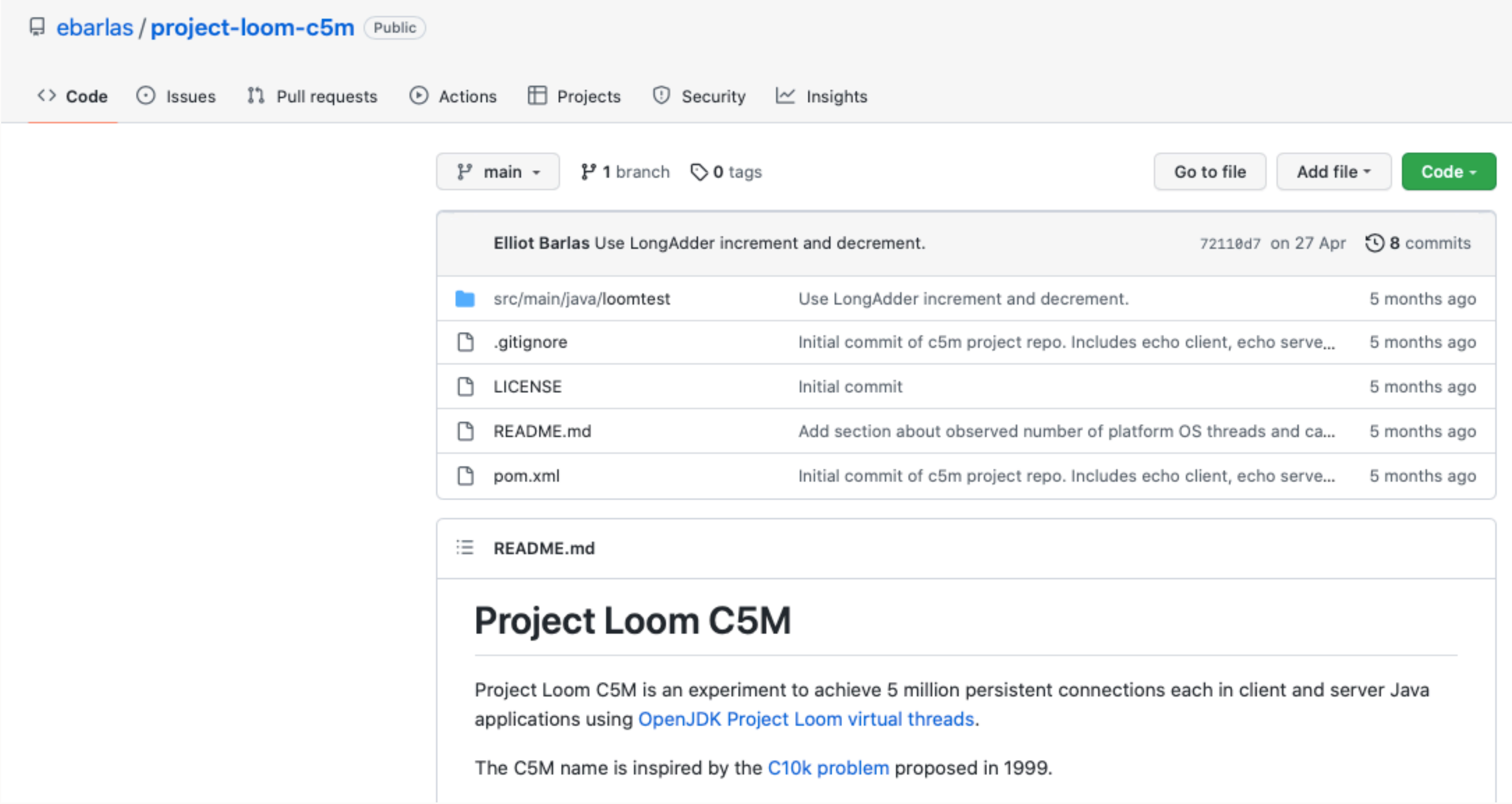
Unfortunately, the NIO API is not great. It felt clunky 20 year ago (yes, it was introduced with Java 1.4 in 2002), and feels very clunky by now. There are improved APIs like the AsynchronousSocketChannel added later, but they didn't quite fit my approach. Generally, I recommend using a library like Netty, Grizzly or others. I actually ended up using the Netty's ByteBuf library, just to have a better buffer abstraction than Java's ByteBuffer.

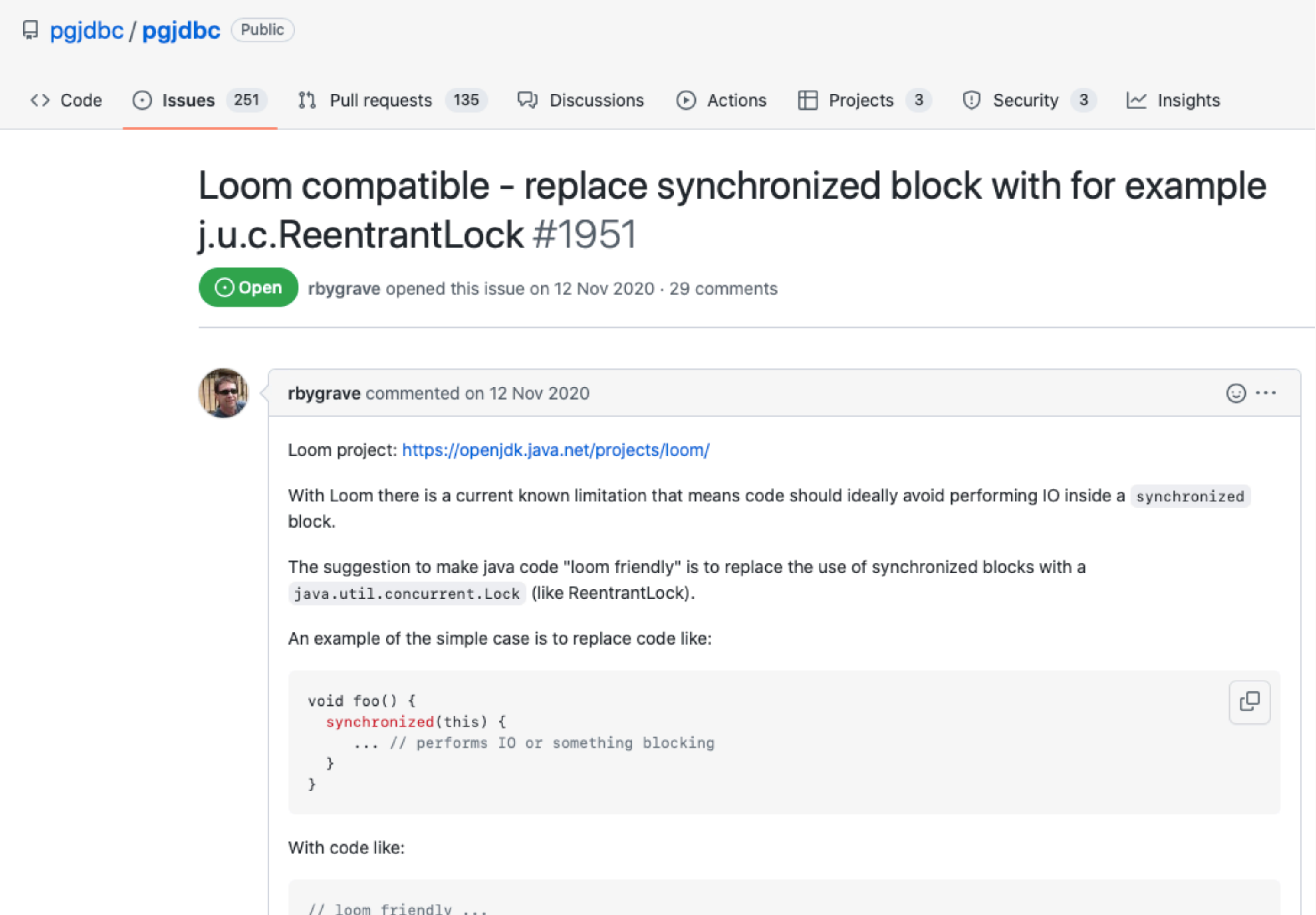
The non-blocking NIO is intended for multiplexing multiple socket operations on that single thread. That usually ends with some callback / event-driven code. However, I wanted to keep classic `blocking` code style code with the virtual threads. The NIO API certainly wasn't designed for that =). Anyway, onwards with the code.


ebarlas / game-of-life-csp Public
Code Issues Pull requests Actions Projects Security Insights
main 1 branch 0 tags
Go to file Add file Code
ebarlas Rename first nextState method parameter for clarity. 3f4c9a3 on 8 Jun 5 commits
images Initial commit with source code, patterns catalog, readme doc, and ... 3 months ago
patterns Initial commit with source code, patterns catalog, readme doc, and ... 3 months ago
src/main/java/gameoflife Rename first nextState method parameter for clarity. 3 months ago
.gitignore This path skips through empty directories code, patterns catalog, readme doc, and ... 3 months ago
LICENSE Initial commit 3 months ago
README.md Add CSP Wikipedia link to README doc. 3 months ago
pom.xml Initial commit with source code, patterns catalog, readme doc, and ... 3 months ago
README.md
Game of Life CSP
Game of Life CSP is a Java implementation of Conway's Game of Life using communicating sequential processes (CSP).
Each grid cell is an independent process and all cell communication occurs via channels.
It's built atop virtual threads, defined in JDK Enhancement Proposal (JEP) 425.
The virtual threads feature is part of Project Loom.
Prior to Project Loom and virtual threads, CSP style programming in this manner simply wasn't available in Java.


kolotyluk / loom-lab Public
Code Issues Pull requests Actions Projects Security Insights
master 1 branch 0 tags
Go to file Add file Code
kolotyluk General Updates 438c4ae 18 days ago 46 commits
.idea General Updates 18 days ago
benchmarks General Updates 18 days ago
docs Use Cucumber testing 9 months ago
laboratory General Updates 18 days ago
old-school Initial commit 11 months ago
.gitignore Setup Maven Site Documentation and GitHub Pages 10 months ago
LICENSE.txt Setup Maven Site Documentation and GitHub Pages 10 months ago
README.md Improve Introductory Documentation 10 months ago
javadoc.css Polish off Throughput Benchmarking 9 months ago
pom.xml General Updates 18 days ago
test.txt Initial commit 11 months ago
README.md
Project Loom Learning Laboratory
A place to learn about Project Loom through hands-on experimentation and exploration.
See more at Project Site Pages
My hope is to develop this into a learning tool for other people as well, so if you have ideas on how this can work better for you or others, please create a ticket in loom-lab Issues. People are encouraged to clone this repo, run the experiments and other code, make local changes, and watch what happens.


lucav76 / Fibry Public
Code Issues Pull requests Actions Projects 1 Security Insights
master 6 branches 27 tags
Go to file Add file Code
lucav76 Updates test dependencies to remove vulnerabilities and to suppo... 70d6983 on 27 May 215 commits
gradle Switches to Gradle 7.4 7 months ago
src Tests stabilization 7 months ago
.gitignore Adds support for release notes 3 years ago
CONTRIBUTING.md Contributing and version fix 3 years ago
LICENSE Strategy and actors that support both void and returning a value 3 years ago
README.md Documentation improvements 7 months ago
TODO.md Distributed version of Fibry, with an HttpChannel, and some simple ... 3 years ago
build.gradle Updates test dependencies to remove vulnerabilities and to suppor... 4 months ago
gradle.properties Version 2.6 7 months ago
gradlew Switches to Gradle 7.4 7 months ago
gradlew.bat Fixes returns 11 months ago
publish.gradle Switch to JReleaser, to improve the release to GitHub 10 months ago
settings.gradle Strategy and actors that support both void and returning a value 3 years ago
README.md
Fibry
Fibry is an experimental Actor System built to be simple and flexible. Hopefully, it will also be fun to use. Fibry is the first Java Actor System using fibers (now called Virtual Threads) from Project Loom, however it also works with threads using any OpenJDK.
Project Loom is an OpenJDK project that is expected to bring fibers (green threads) and continuations (co-routines) to Java. Fibry 1.X works with any version of Java starting from Java 8, while Fibry 2.X is targeting Java 11, but in both cases, you will need to use Loom if you want to leverage the power of fibers. Fibry 1.X is supported, and changes are available in the jdk8 branch. Fibry aims to replicate some of the features of the Erlang Actor System in Java. Fibry allows you to send code to be executed in the thread/fiber of an actor, a mechanism similar to the one used in Chromium.
The current line of development is meant to make Fibry useful on the creation of IoT products and video games supporting online multi-players functionalities.


ebarlas / project-loom-c5m Public
Code Issues Pull requests Actions Projects Security Insights
main 1 branch 0 tags
Go to file Add file Code
Elliot Barlas Use LongAdder increment and decrement. 72110d7 on 27 Apr 8 commits
src/main/java/loomtest Use LongAdder increment and decrement. 5 months ago
.gitignore Initial commit of c5m project repo. Includes echo client, echo serve... 5 months ago
LICENSE Initial commit 5 months ago
README.md Add section about observed number of platform OS threads and ca... 5 months ago
pom.xml Initial commit of c5m project repo. Includes echo client, echo serve... 5 months ago
README.md
Project Loom C5M
Project Loom C5M is an experiment to achieve 5 million persistent connections each in client and server Java applications using OpenJDK Project Loom virtual threads.
The C5M name is inspired by the C10k problem proposed in 1999.


pgjdbc / pgjdbc Public
Code Issues 251 Pull requests 135 Discussions Actions Projects 3 Security 3 Insights
Loom compatible - replace synchronized block with for example j.u.c.ReentrantLock #1951
Open rbygrave opened this issue on 12 Nov 2020 · 29 comments
rbygrave commented on 12 Nov 2020
Loom project: https://openjdk.java.net/projects/loom/
With Loom there is a current known limitation that means code should ideally avoid performing IO inside a synchronized block.
The suggestion to make java code "loom friendly" is to replace the use of synchronized blocks with a java.util.concurrent.Lock (like ReentrantLock).
An example of the simple case is to replace code like:
void foo() {
synchronized(this) {
... // performs IO or something blocking
}
}
With code like:

eclipse / jetty.project Public

Code | Issues 354 | Pull requests 23 | Actions | Projects 5 | Security 12 | Insights

Support Loom #8007

Closed sbordet opened this issue on 16 May · 2 comments · Fixed by #8465

sbordet commented on 16 May — Contributor

Target Jetty version(s)
10.0.x

Enhancement Description
With project Loom being integrated in Java 19+22, we should offer an option to call `Handler`s with a virtual thread.
This would allow testing by early adopters.

👍 1 ❤️ 14 🚀 4

sbordet added the Enhancement label on 16 May

sbordet added a commit that referenced this issue on 22 May

Fixes #8007 - Support Loom. … Verified fc11e9a

sbordet linked a pull request on 22 May that will close this issue

Fixes #8007 - Support Loom. #8035 Closed

Vert.x Virtual Threads Incubator

CI passing

Incubator for virtual threads based prototypes.

Prerequisites

- Vert.x 4.3.3
- Java 19 using preview feature
  - OpenJDK 19 EA
  - Maven
  - Intellij

Projects

- Async/await incubator
- Execute blocking incubator
- Examples

mp911de / spring-boot-virtual-threads-experiment Public

Code | Issues | Pull requests | Actions | Security | Insights

main · 2 branches · 0 tags · Go to file · Add file · Code

mp911de Update build.yml · 6efa152 on 4 Aug · 10 commits

.github/workflows	Update build.yml	2 months ago
.mvn/wrapper	Initial commit	2 years ago
img	Reduce Kernel Thread scenario to achieve similar throughput as Vir…	2 years ago
src/main	Migrate to Spring Boot 3 and Java 19	2 months ago
.gitignore	Initial commit	2 years ago
.sdkmanrc	Adding pipeline to build prototype with latest Java 19 EA build #2	2 months ago
README.md	Migrate to Spring Boot 3 and Java 19	2 months ago
mvnw	Initial commit	2 years ago
mvnw.cmd	Initial commit	2 years ago
pom.xml	Removing unneeded dependency exclusion #1	2 months ago

README.md

Project Loom Experiment using Spring Boot, Spring WebMVC, and Postgres

This repository contains an experiment that uses a Spring Boot application with Virtual Threads.

Involved components:

- Spring Framework 6.0 M5
- Spring Boot 3.0 M4
- Apache Tomcat 10.1.0 M17
- HikariCP 5.0.1 (Loom issue: brettwooldridge/HikariCP#1463)
- PGJDBC 42.4.0 (PR that turns `synchronized` into Loom-friendly Locks: pgjdbc/pgjdbc#1951)

This experiment evolves incrementally, find the previous state at https://github.com/mp911de/spring-boot-virtual-threads-experiment/tree/boot-2.4.

You need Java 19 (EAP) with `--enable-preview` to run the example.

June 22, 2022 #release

Quarkus 2.10.0.Final released - Preliminary work on Loom's virtual threads and various refinements all over the place

By Guillaume Smet

New month, new Quarkus feature release, you know the drill: Quarkus 2.10.0.Final has landed.

This version is a mix of exploratory work and refinements on existing extensions:

- Preliminary work on Loom's virtual threads

Virtual Threads and Tomcat

Virtual threads are the ideal mechanism for running mostly blocking tasks, providing a high level of concurrency without requiring asynchronous acrobatics from business logic programmers. I show that it is easy to configure Tomcat for virtual threads, provided one makes a small change to the Tomcat source code.

TM

Virtual Threads

Java 19 has *virtual threads* as a preview feature, described in JEP 425. Virtual threads are scheduled to run in platform threads. When a virtual thread blocks, it is *parked* and another virtual thread can run in its place. Large numbers of virtual threads can run concurrently, provided that they mostly block. This workload is typical in web applications where requests spend much of their time waiting for responses from database queries or other external services.

Some developers are making mistakes

- Thinking that virtual threads are *faster threads*
- Replacing platform threads with virtual threads rather than tasks with virtual threads
- Changing the `ThreadFactory` for a thread pool, thus pooling virtual threads
- Pinning issues and assuming that all uses of monitors must be replaced
- Using framework/libraries that make heavy use of thread locals
- Warmup issues
- Doing over complicated stuff
- Some misunderstanding as to where the performance benefits come from

Migration: The guidance to developers in JEP 444

- Move to simpler blocking/synchronous code
- Migrate tasks to virtual threads, not platform threads to virtual threads
- Use Semaphores or similar to limit concurrency
- Don’t cache expensive objects in thread locals
- Avoid lengthy and frequent pinning (for now anyway)

Summary up to JDK 21 RDP2

- Virtual threads have been well received by developers and eco system
- Significant interest in avoiding async/reactive, go back to simpler synchronous/blocking code instead
- Frameworks learning how to expose virtual threads to developers
- Performance is good, with the exception of a few areas
  - JVM TI based performance profilers
  - Timers aren’t as scalable as they could be
- Reliability is good
- Pinning due to synchronization is the main issue that comes up

Challenges

- Big challenge at JVMLS 2019
  - How to expose virtual threads to developers
  - Adding continuations to HotSpot VM
- Implementing a thread library in Java
  - All the challenges of “Java on Java”
    - Can only use a subset of platform to avoid bootstrapping, nested parking, and other issues
    - Hard to reliably recover/continue after stack overflow or OOME
  - *All your blocking belongs to us*
    - There are hundreds of potentially blocking APIs
- Herding threads
- Thread locals
- Serviceability, e.g. JVM TI, thread dumps

Herding Threads

- A virtual thread per task means lots of virtual threads
- How should lot of threads be organised? Do they need to be organised?
- Long standing thread-organizing construct is `java.lang.ThreadGroup`
  - Select or inherit the `ThreadGroup` at `Thread` create time
  - Doesn't play well with `ThreadFactory`, no dynamic placement
- Serviceability is an interested party

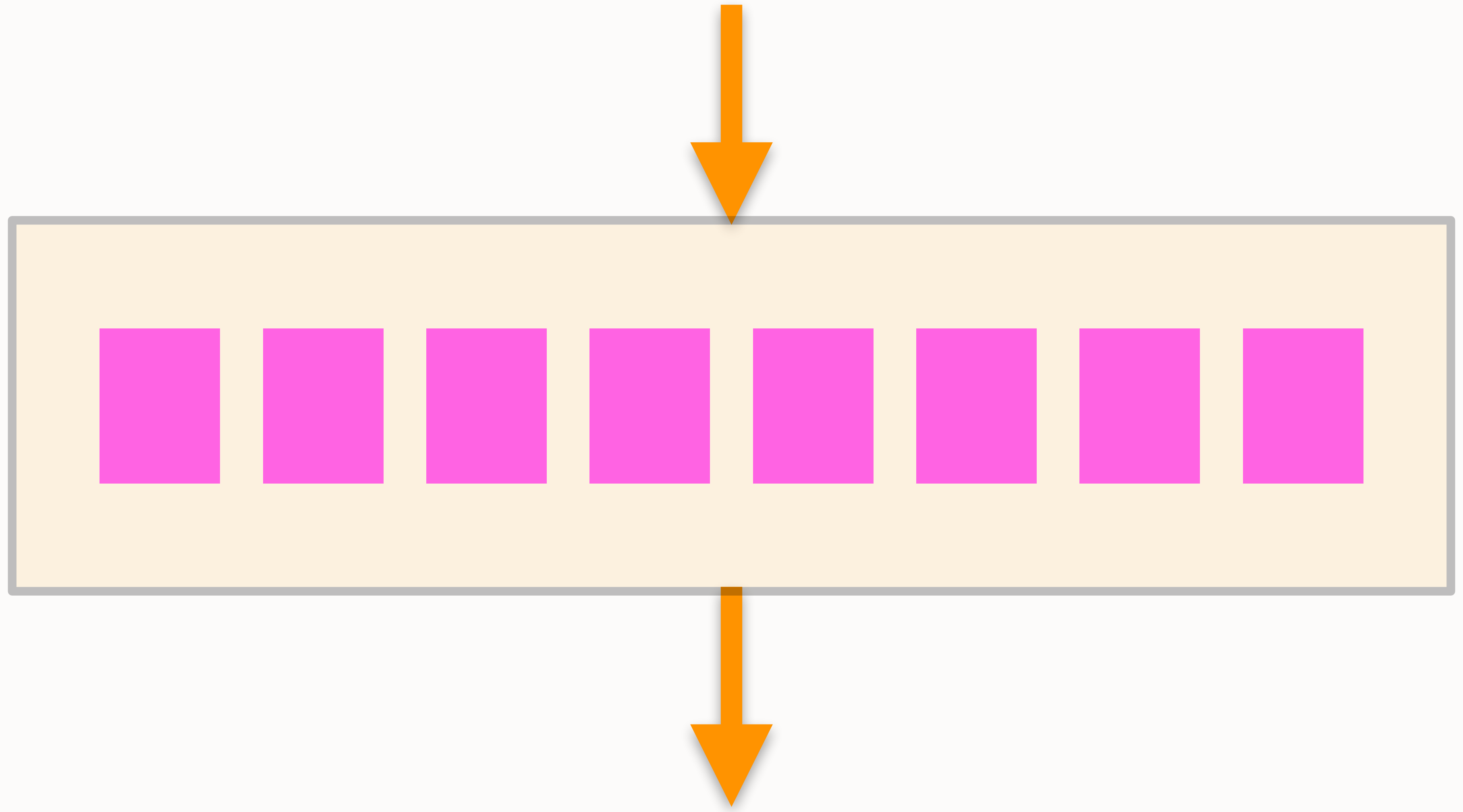
… but none of the tools or APIs scale to millions of virtual threads

Herding Threads

- De-emphasize, and eventually deprecate, legacy `ThreadGroup`
- Introduce *Structured Concurrency* to treat groups of related tasks running in different threads as a single unit of work
- Acknowledge other natural groupings of threads
  - A thread pool is a grouping of worker threads
  - A thread-per-task executor is a grouping of (typically virtual) threads
- Allow serviceability tools to discover/enumerate threads

Structured Concurrency

- When *control splits into multiple concurrent paths, we want to make sure that they join up again*



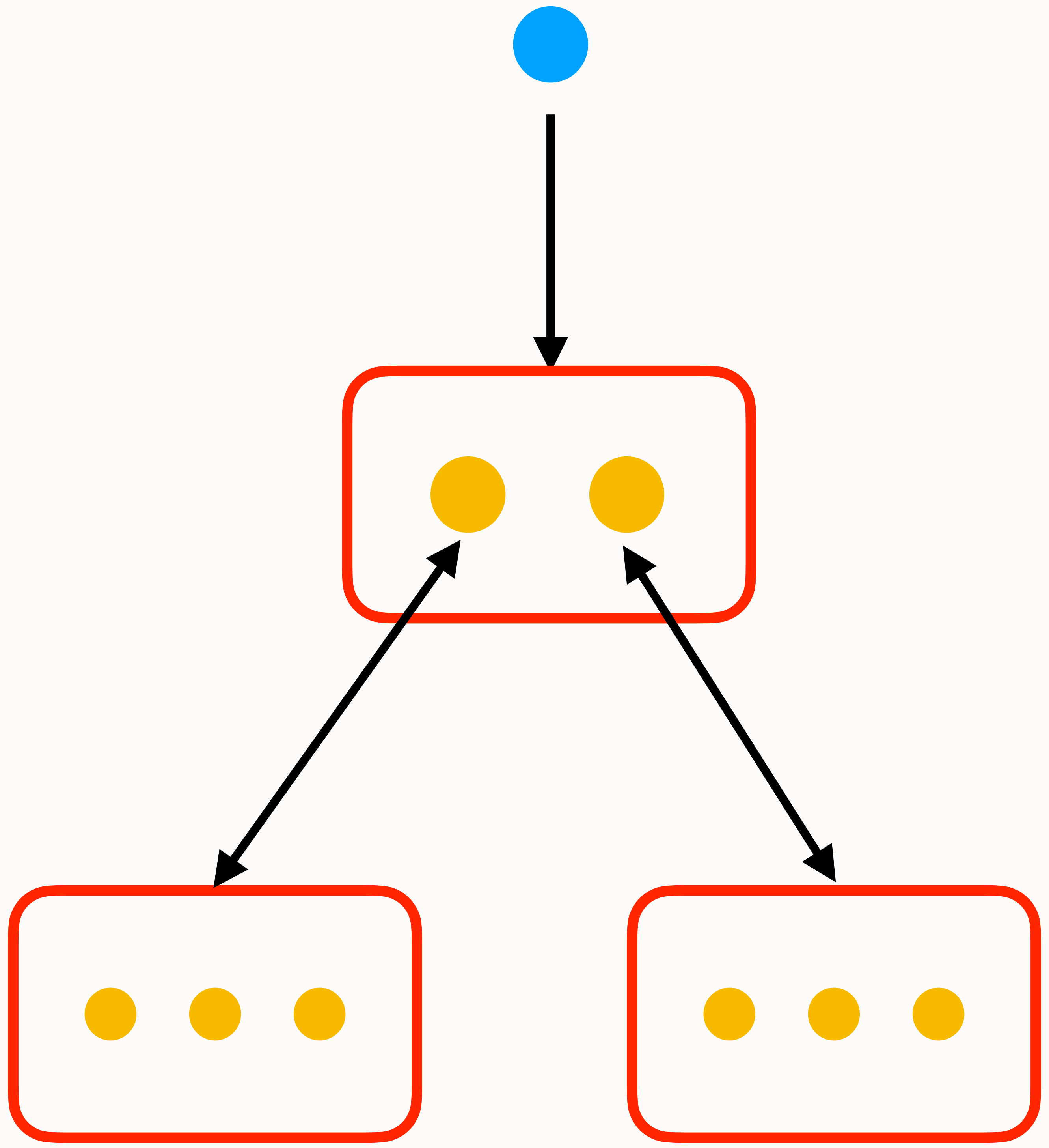
- A big plus is that it helps to *eliminate common risks arising from cancellation and shutdown, such as thread leaks and cancellation delays*

StructuredTaskScope (JEP 453)

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

    Supplier<String> result1 = scope.fork(task1);
    Supplier<String> result2 = scope.fork(task2);

    scope.join();

    scope.throwIfFailed(e -> new WebApplicationException(e));

    // both subtasks completed successfully
    String result = Stream.of(result1, result2)
            .map(Supplier::get)
            .collect(Collectors.joining(", ", "{ ", " }"));

    ...
 }
```


StructuredTaskScope (JEP 453)

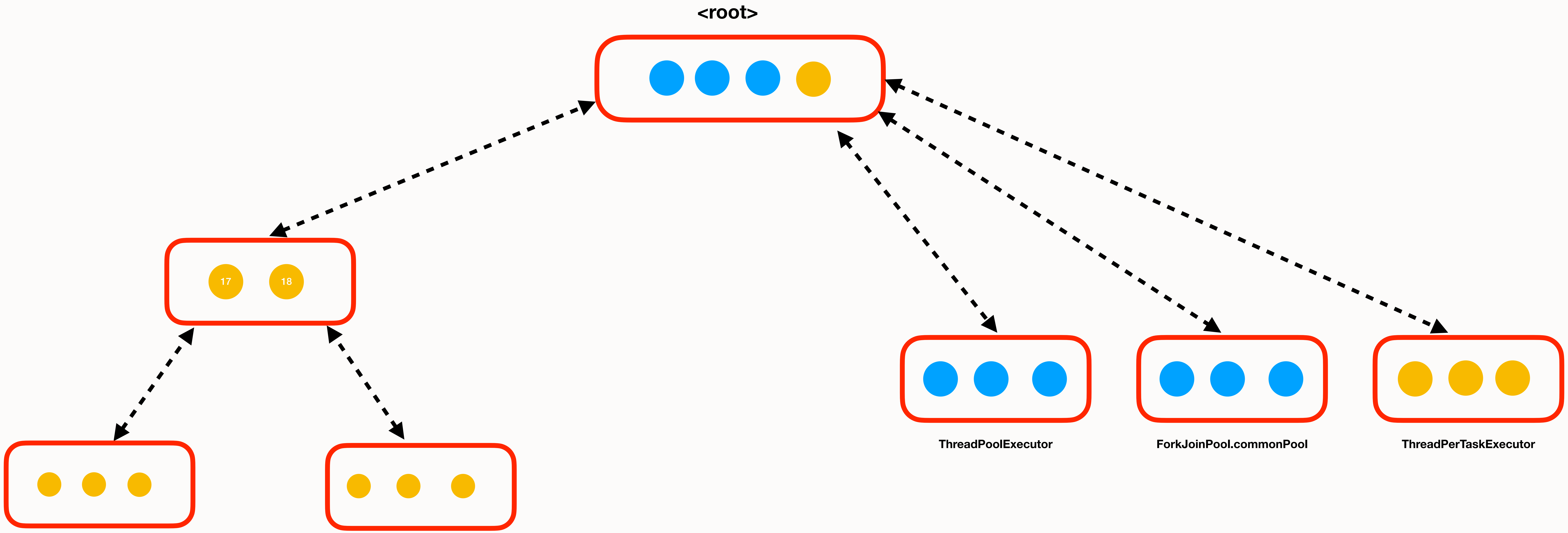
```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

     Supplier<String> result1 = scope.fork(task1);
     Supplier<String> result2 = scope.fork(task2);

     scope.join();

     scope.throwIfFailed(e -> new WebApplicationException(e));

     // both subtasks completed successfully
     String result = Stream.of(result1, result2)
             .map(Supplier::get)
             .collect(Collectors.joining(", ", "{ ", " }"));

     ...
 }
```



- Usages leads to a tree of thread groupings

Herding Threads

- Bring all thread groups into a tree
- Observe with serviceability tools (e.g. `jcmd`), maybe APIs in the future



Thread locals

- Overly general mechanism to associate data with the current thread of execution

- Used for a wide range of purposes, e.g.
  - Caching, esp. mutable objects that are expensive to create
  - Implicit parameters and return values

- TLs have several problems
  - Unbounded lifetime
  - Unconstrained mutability
  - Unconstrained memory usage
  - InheritableThreadLocal = expensive inheritance

Virtual threads and thread locals

- Virtual threads support thread locals
  - Forced moved: too much existing code uses TLs
  - Dropped (preview) API to opt-out of thread locals out of concern that is bifurcate eco system

- Migration is from tasks to virtual threads, not platform threads to virtual threads
  - Problematic for code that assumes running in a thread pool or event loop
  - Caching results in negative performance

- Migration to virtual threads means moving away from caching expensive objects in thread locals
  - Move to immutable objects where possible, e.g `SimpleDateFormat` to `DateTimeFormatter`
  - A global cache can work for some use-cases

Moving away from thread locals

- Re-evaluate use of TLs
  - JDK dropped several usages from java.base after re-evaluating the performance

- Scoped Values - JEP 446
  - A value that may be safely and efficiently shared to methods without using method parameters
  - Use for cases where there is "one-way transmission" of data without using method parameters
    - Callbacks, detect recursion initialization, …

- In JVMLS 2019 we talked about
  - Processor locals
    - Prototype on based on RSEQ has been parked, may pick this up again
  - Task locals

Status of libraries

- Networking
  - All blocking ops release carrier to do other work

- Internet-Address Resolution (DNS look-up, reverse look-up)
  - JEP 418 introduced a provider mechanism
    - Not proposing that JDK include its own DNS client at this time
    - Published samples based on JNDI-DNS and Netty DNS, need eco system to step up

- File I/O
  - All potentially blocking ops currently compensate by temporarily increasing parallelism during op
  - Work in progress to allow implementation to be based on async or `io_uring`

Java Virtual Machine Tool Interface (JVM TI)

- 2-way native interface to support a broad range of tool agents
  - Debuggers, instrumentation based tools, heap profilers, …
  - Very large API surface (150 functions, 50 callbacks), deeply invasive

- Virtual threads are implemented in Java, the scheduler is in Java, lots of challenges to support JVM TI

- The main challenge is the thread identity changes when a virtual thread mounts or unmounts
  - Complicated interaction with thread suspend/resume

- Main break through since JVMLS 2019 is to treat the carrier as *blocked* when a virtual thread is mounted
  - The carrier thread is unblocked when the virtual thread unmounts

- A forced move is to hide events when executing the mount/unmount code
  - JVM TI meets "Java on Java"

Java Virtual Machine Tool Interface (JVM TI)

- Debugger
- Heap profilers
- Performance profilers
  - Currently overwhelmed by bookkeeping overhead - multi-step plan to reduce overhead
  - Main question is if JVMTI is the right interface for performance profilers?
- No equivalent of `GetAllThreads` for virtual threads

Status of other serviceability areas

- Java Debug Wire Protocol (JDWP) + Java Debug Interface (JDI)
  - All IDE/debuggers are working with virtual threads
  - Missing debugger supporting for discovering threads and navigating groupings of threads
- JMX and java.lang.management
- JDK Flight Recorder
- HPROF heap dumps
- Thread dumps

Thread dumps

- Usually the first port of call when troubleshooting
- HotSpot VM thread dump has organically grown over many years to include a lot of information
  - But doesn’t scale to millions of virtual threads
  - Virtual threads are just objects in the heap
- New thread dump format
  - Provides a *weakly consistent* view of the threads in each “thread grouping”
  - `HotSpotDiagnosticMXBean` API or `jcmd`
  - Plain text or JSON format for now
    - JSON format intended to be parsed, enables tools to visualize, deduplicate, …
  - No lock information or deadlock detection at this time

```
{
  "threadDump": {
    "processId": "78091",
    "time": "2023-08-07T20:26:44.186396Z",
    "runtimeVersion": "22-internal-adhoc.albatem.open",
    "threadContainers": [
      {
        "container": "<root>",
        "parent": null,
        "owner": null,
        "threads": [
         {
           "tid": "1",
           "name": "main",
           "stack": [
              "java.base\/java.lang.Thread.getStackTrace(Thread.java:2421)",
              "java.base\/jdk.internal.vm.ThreadDumper.dumpThreadToJson(ThreadDumper.java:264)",
              "java.base\/jdk.internal.vm.ThreadDumper.dumpThreadsToJson(ThreadDumper.java:237)",
              "java.base\/jdk.internal.vm.ThreadDumper.dumpThreadsToJson(ThreadDumper.java:201)",
              "java.base\/jdk.internal.vm.ThreadDumper.dumpThreadsToJson(ThreadDumper.java:176)",
              "jdk.management\/com.sun.management.internal.HotSpotDiagnostic.dumpThreads(HotSpotDiagnostic.java:197)",
              "jdk.management\/com.sun.management.internal.HotSpotDiagnostic.lambda$dumpThreads$1(HotSpotDiagnostic.java:178)",
              "java.base\/java.security.AccessController.doPrivileged(AccessController.java:571)",
              "jdk.management\/com.sun.management.internal.HotSpotDiagnostic.dumpThreads(HotSpotDiagnostic.java:182)",
              "Test.main(Test.java:28)",
              "java.base\/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(DirectMethodHandleAccessor.java:103)",
              "java.base\/java.lang.reflect.Method.invoke(Method.java:580)",
              "jdk.compiler\/com.sun.tools.javac.launcher.Main.execute(Main.java:484)",
              "jdk.compiler\/com.sun.tools.javac.launcher.Main.run(Main.java:208)",
              "jdk.compiler\/com.sun.tools.javac.launcher.Main.main(Main.java:135)"
           ]
         },
         {
           "tid": "9",
           "name": "Reference Handler",
           "stack": [
              "java.base\/java.lang.ref.Reference.waitForReferencePendingList(Native Method)",
              "java.base\/java.lang.ref.Reference.processPendingReferences(Reference.java:246)",
              "java.base\/java.lang.ref.Reference$ReferenceHandler.run(Reference.java:208)"
           ]
         },
         {
           "tid": "10",
           "name": "Finalizer",
           "stack": [
              "java.base\/java.lang.Object.wait0(Native Method)",
              "java.base\/java.lang.Object.wait(Object.java:375)",
              "java.base\/java.lang.Object.wait(Object.java:348)",
              "java.base\/java.lang.ref.NativeReferenceQueue.await(NativeReferenceQueue.java:48)",
              "java.base\/java.lang.ref.ReferenceQueue.remove0(ReferenceQueue.java:158)",
              "java.base\/java.lang.ref.NativeReferenceQueue.remove(NativeReferenceQueue.java:89)",
              "java.base\/java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:173)"
           ]
         },
         {
           "tid": "11",
           "name": "Signal Dispatcher",
           "stack": [
           ]
         },
         {
           "tid": "18",
           "name": "Common-Cleaner",
           "stack": [
              "java.base\/jdk.internal.misc.Unsafe.park(Native Method)",
              "java.base\/java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:269)",
              "java.base\/java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:1847)",
              "java.base\/java.lang.ref.ReferenceQueue.await(ReferenceQueue.java:71)",
              "java.base\/java.lang.ref.ReferenceQueue.remove0(ReferenceQueue.java:143)",
              "java.base\/java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:218)",
              "java.base\/jdk.internal.ref.CleanerImpl.run(CleanerImpl.java:140)",
              "java.base\/java.lang.Thread.run(Thread.java:1570)",
              "java.base\/jdk.internal.misc.InnocuousThread.run(InnocuousThread.java:186)"
           ]
         },
         {
           "tid": "19",
           "name": "Notification Thread",
           "stack": [
           ]
         }
        ],
        "threadCount": "6"
      },
      {
        "container": "java.util.concurrent.ThreadPoolExecutor@7e5afaa6",
        "parent": "<root>",
        "owner": null,
        "threads": [
         {
           "tid": "25",
           "name": "pool-1-thread-1",
           "stack": [
              "java.base\/java.lang.Thread.sleepNanos0(Native Method)",
              "java.base\/java.lang.Thread.sleepNanos(Thread.java:491)",
              "java.base\/java.lang.Thread.sleep(Thread.java:522)",
              "Test.lambda$main$0(Test.java:13)",
              "java.base\/java.util.concurrent.FutureTask.run(FutureTask.java:317)",
              "java.base\/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1144)",
              "java.base\/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:642)",
              "java.base\/java.lang.Thread.run(Thread.java:1570)"
           ]
         }
        ],
        "threadCount": "1"
      },
      {
        "container": "ForkJoinPool-1\/jdk.internal.vm.SharedThreadContainer@3e27ba32",
        "parent": "<root>",
        "owner": null,
        "threads": [
         {
           "tid": "27",
           "name": "ForkJoinPool-1-worker-1",
           "stack": [
              "java.base\/jdk.internal.misc.Unsafe.park(Native Method)",
              "java.base\/java.util.concurrent.locks.LockSupport.parkUntil(LockSupport.java:449)",
              "java.base\/java.util.concurrent.ForkJoinPool.awaitWork(ForkJoinPool.java:1891)",
              "java.base\/java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1809)",
              "java.base\/java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:188)"
           ]
         }
        ],
        "threadCount": "1"
      },
      {
        "container": "java.util.concurrent.ScheduledThreadPoolExecutor@2f217633",
        "parent": "<root>",
        "owner": null,
        "threads": [
        ],
        "threadCount": "0"
      },
      {
        "container": "ForkJoinPool.commonPool\/jdk.internal.vm.SharedThreadContainer@7ef82753",
        "parent": "<root>",
        "owner": null,
        "threads": [
        ],
        "threadCount": "0"
      },
      {
        "container": "java.util.concurrent.ThreadPoolExecutor@534a5a98",
        "parent": "<root>",
        "owner": null,
        "threads": [
        ],
        "threadCount": "0"
      },
      {
        "container": "java.util.concurrent.StructuredTaskScope@3bf9ce3e",
        "parent": "<root>",
        "owner": "1",
        "threads": [
        ],
        "threadCount": "0"
      }
    ]
  }
}
```

Current exploration/work in progress

- "Quality of implementation" and performance
  - Java monitors
  - Compressed frames
  - Re-implement file I/O so it can be backed by async or io_uring
  - Include lock information in thread dumps
  - More scalable timer support
  - More scalable tracking of threads; APIs for enumeration/navigating
  - JVM TI performance
- Custom Schedulers
- Get feedback and progress Structured Concurrency and Scoped Values to be permanent features

Links

- JEP 444: Virtual Threads
  https://openjdk.org/jeps/444

- JEP 453: Structured Concurrency (Preview)
  https://openjdk.org/jeps/453

- JEP 446: Scoped Values (Preview)
  https://openjdk.org/jeps/446

- Repository
  https://github.com/openjdk/loom/ (several branches)

- Mailing list
  https://mail.openjdk.org/mailman/listinfo/loom-dev