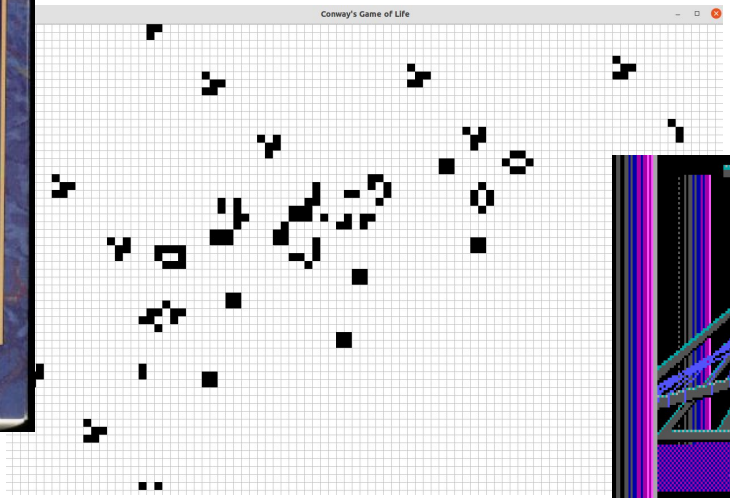
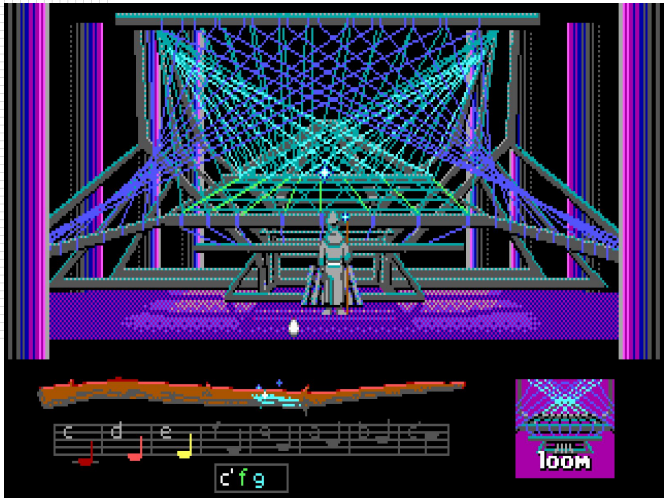
Game of Loom 2: life and dead(lock) of a virtual thread



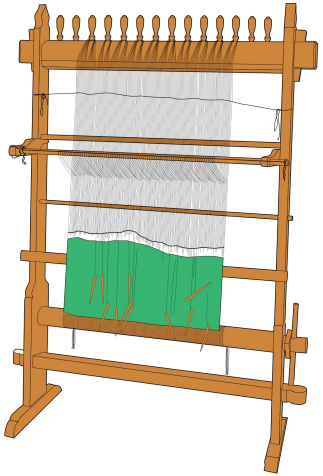


by Mario Fusco

@mariofusco

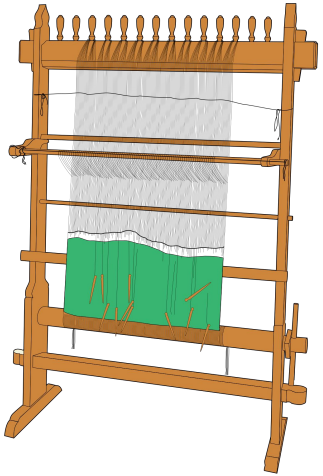
Where did we leave? A recap on virtual threads …

- **Virtual threads** make a better use of OS threads **getting rid of the 1:1 relationship** with operating system threads
  - Many virtual threads can be **multiplexed** on the same platform thread (**carrier**)



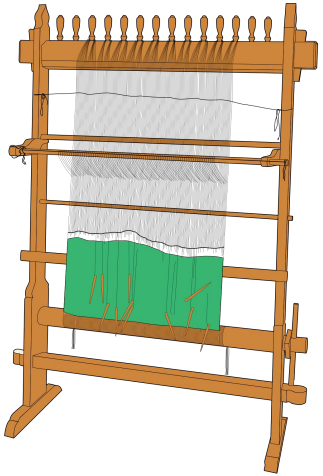
Where did we leave? A recap on virtual threads …

- **Virtual threads** make a better use of OS threads **getting rid of the 1:1 relationship** with operating system threads
  - Many virtual threads can be **multiplexed** on the same platform thread (**carrier**)
- When a virtual thread calls a blocking operation the JDK performs a **nonblocking OS call** and automatically **suspends** the virtual thread **until the operation finishes**
  - Virtual threads perform **blocking calls without consuming resources**, thus allowing to **write asynchronous** code in a way that **looks synchronous**
  - What virtual threads are really good for is … **waiting**



Where did we leave? A recap on virtual threads …

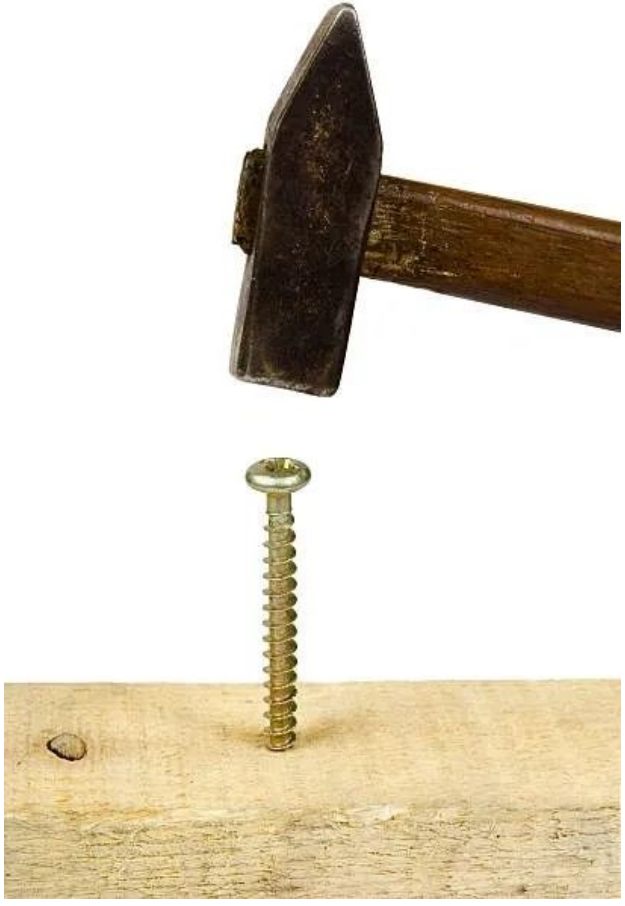
- **Virtual threads** make a better use of OS threads **getting rid of the 1:1 relationship** with operating system threads
  - Many virtual threads can be **multiplexed** on the same platform thread (**carrier**)
- When a virtual thread calls a blocking operation the JDK performs a **nonblocking OS call** and automatically **suspends** the virtual thread **until the operation finishes**
  - Virtual threads perform **blocking calls without consuming resources**, thus allowing to **write asynchronous** code in a way that **looks synchronous**
  - What virtual threads are really good for is … **waiting**
- Virtual threads are **lightweight**
  - 200-300B metadata
  - Pay-as-you-go **stack** (allocated on heap)
  - Some ns (or below 1µs) for **context switch** (in **user space**)



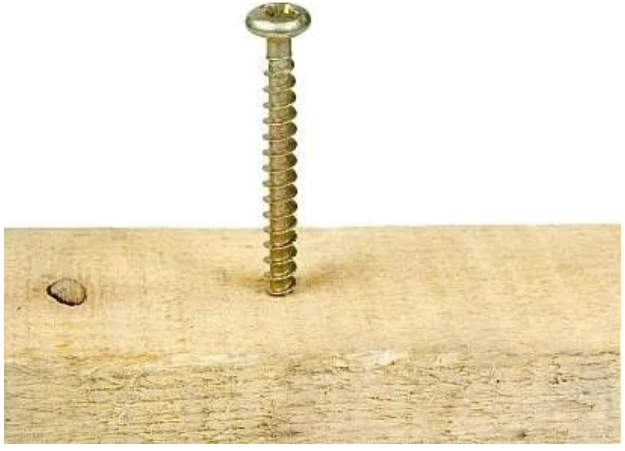

VIRTUAL THREADS
LIGHTYEAR
VIRTUAL THREADS EVERYWHERE
imgflip.com


YOU WANT VIRTUAL THREADS
EVERYBODY WANT VIRTUAL THREADS
imgflip.com

Should we really use virtual threads *EVERYWHERE*?



Should we really use virtual threads *EVERYWHERE*?

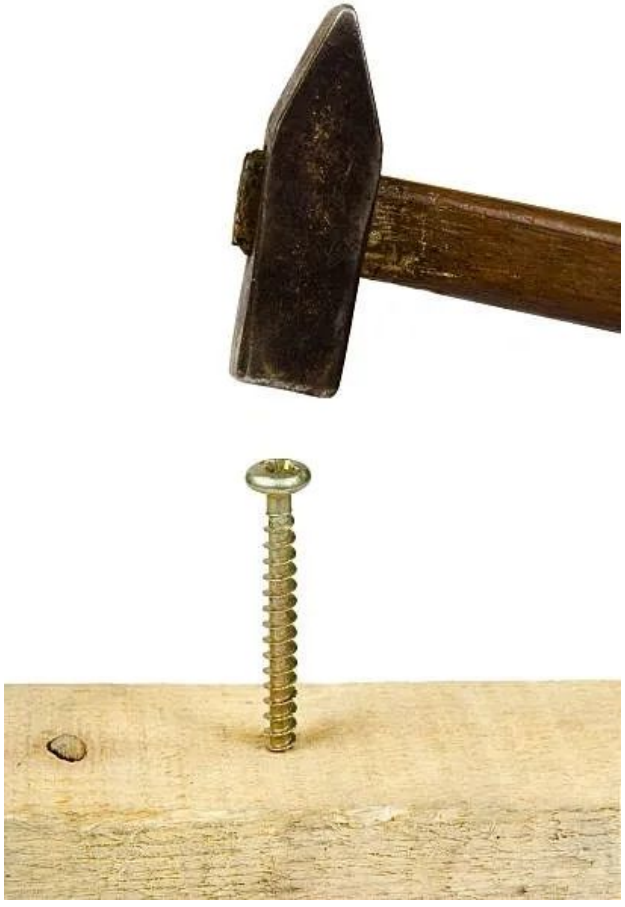


- Not ideal for CPU-bound task

Lack of fairness due to the fact that virtual threads are **not pre-emptive**

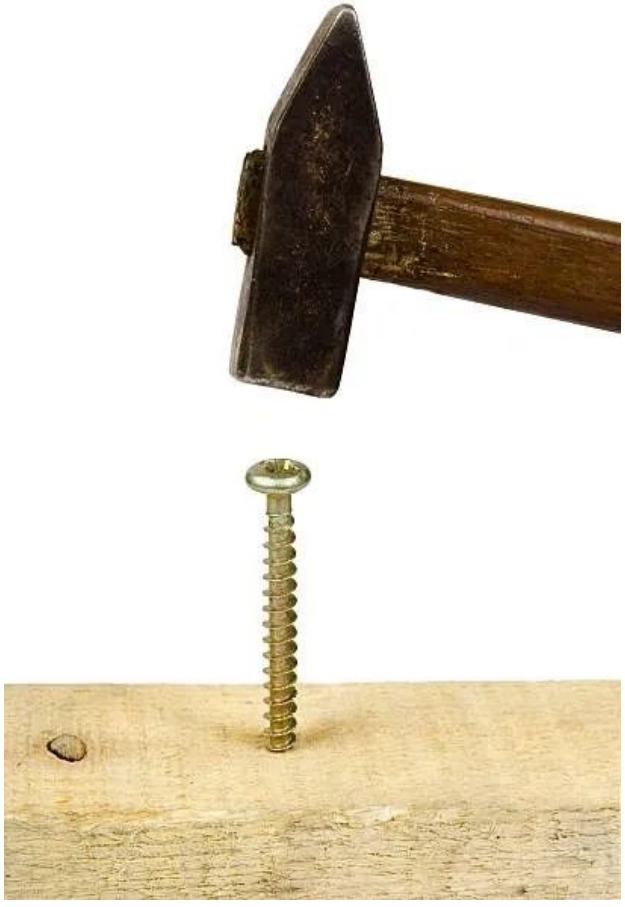
Overall performance concerns caused by overhead of virtual threads scheduler

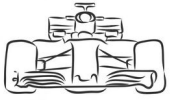
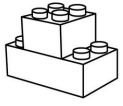
Should we really use virtual threads *EVERYWHERE*?



- Not ideal for CPU-bound task
  - **Lack of fairness** due to the fact that virtual threads are **not pre-emptive**
  - **Overall performance** concerns caused by overhead of virtual threads scheduler
- **Not virtual-threads friendly** frameworks & tools
  - Use of `synchronize` can lead to **pinning**
  - Use of **ThreadLocal**

Should we really use virtual threads *EVERYWHERE*?

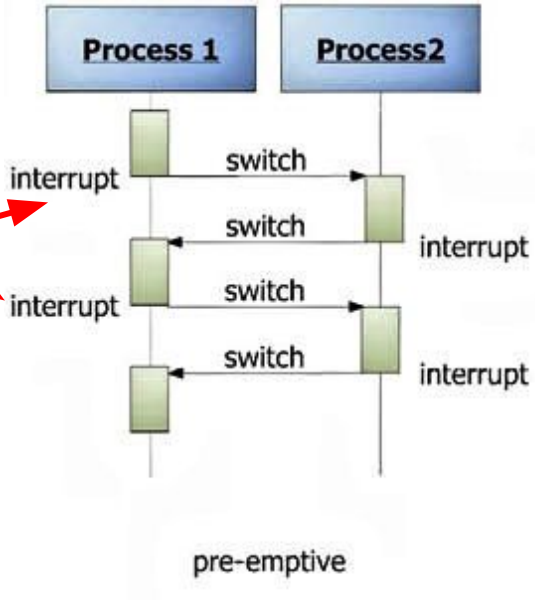


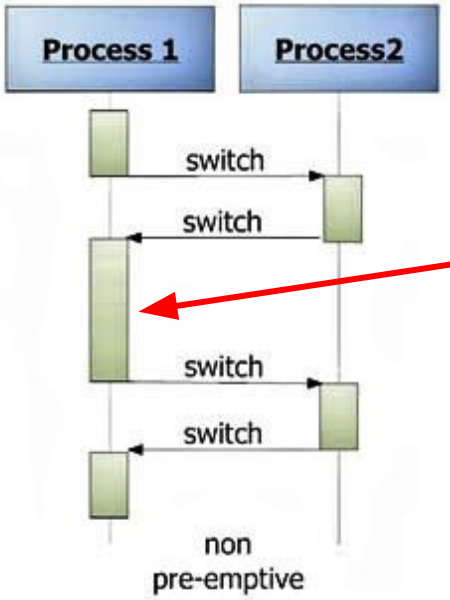
- Not ideal for CPU-bound task
  -  **Lack of fairness** due to the fact that virtual threads are **not pre-emptive**
  -  **Overall performance** concerns caused by overhead of virtual threads scheduler
- **Not virtual-threads friendly** frameworks & tools
  -  Use of `synchronize` can lead to **pinning**
  -  Use of **ThreadLocal**
- **Poorly customizable**
  -  **Carrier** native thread pool is **not pluggable** (but for a very good reason)

Lack of fairness in CPU bound tasks

Virtual Threads are **not pre-emptive**, they cannot be descheduled while running heavy calculations without ever calling a JDK's blocking methods, so they are **not a good fit for CPU-bound** tasks when **fairness** is key

Interrupts the processing of a task and transfers the CPU to another process





Non-preemptive operation usually proceeds towards completion uninterrupted

Lack of fairness in CPU bound tasks

Virtual Threads are **not pre-emptive**, they cannot be descheduled while running heavy calculations without ever calling a JDK's blocking methods, so they are **not a good fit for CPU-bound** tasks when **fairness** is key

The cost of a continuation

In Project Loom, the word **continuation** means **delimited continuation**, also sometimes called a **coroutine**. It can be thought of as sequential code that may suspend or yield execution at some point by itself and can be resumed by a caller.

When a virtual thread hits a blocking call its continuation is **suspended** and the JVM **unmounts** the thread from its carrier …

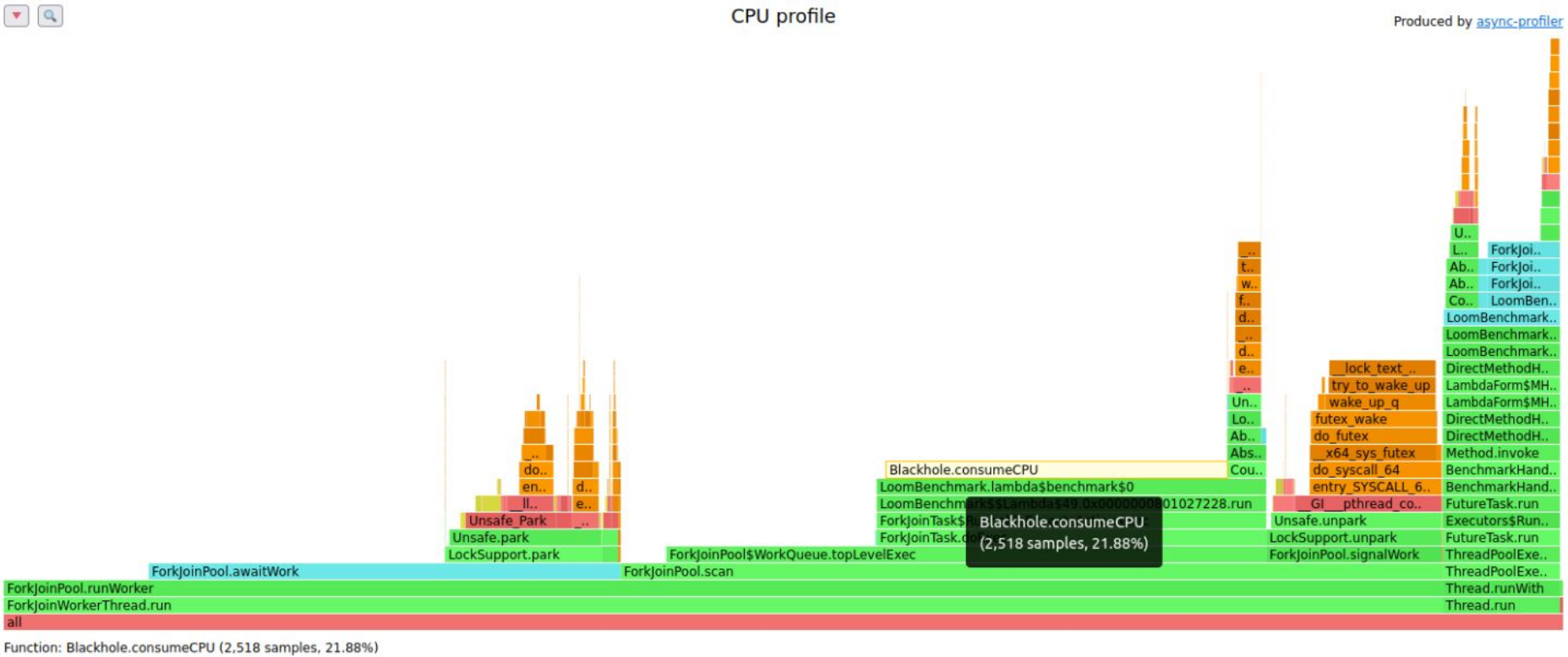
The cost of a continuation

In Project Loom, the word **continuation** means **delimited continuation**, also sometimes called a **coroutine**. It can be thought of as sequential code that may suspend or yield execution at some point by itself and can be resumed by a caller.

When a virtual thread hits a blocking call its continuation is **suspended** and the JVM **unmounts** the thread from its carrier …

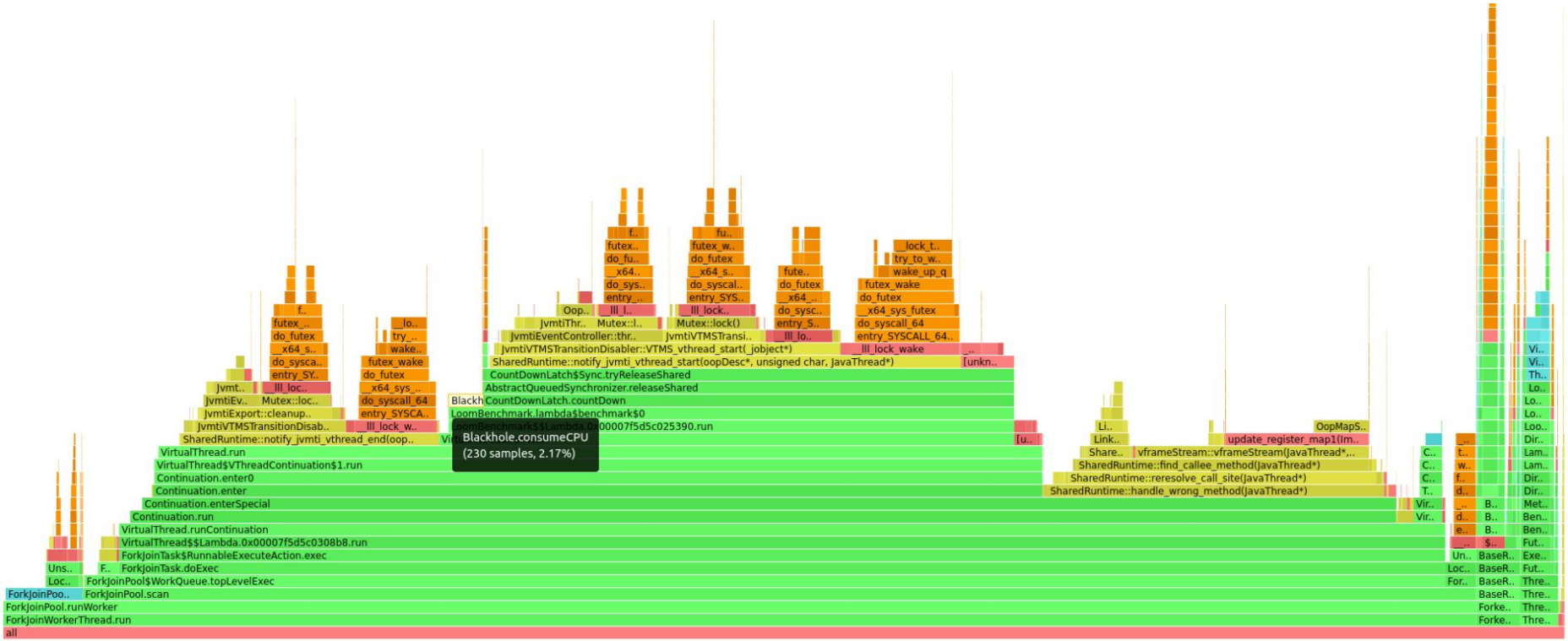
… but this also has a cost!

Native Threads Execution



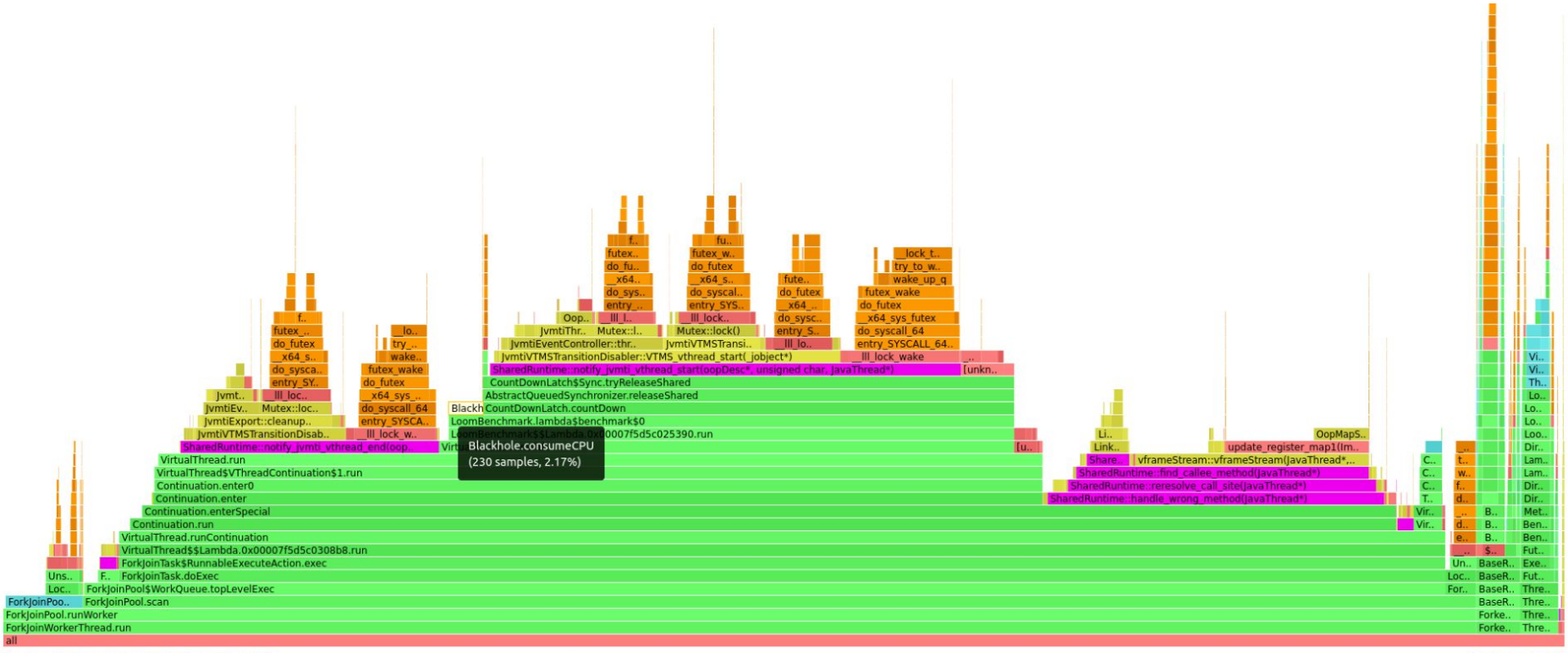
Function: Blackhole.consumeCPU (2,518 samples, 21.88%)

Virtual Threads Execution - The cost of a continuation



Function: Blackhole.consumeCPU (230 samples, 2.17%)

Virtual Threads Execution - The cost of a continuation



ThreadLocals as thread scoped variables

```
enum RightsLevel { ADMIN, GUEST }

record Principal(RightsLevel rights) {
   public boolean canOpen() {
       return rights == RightsLevel.ADMIN;
   }
}

record Request(boolean authorized) { }

class Server {
   final static ThreadLocal<Principal> PRINCIPAL =
                               new ThreadLocal<>();

   String serve(Request request) {
       var level = request.authorized() ?
            RightsLevel.ADMIN : RightsLevel.GUEST;
       PRINCIPAL.set( new Principal(level) );
       return Application.handle( request );
   }
}
```

ThreadLocals as thread scoped variables

```
enum RightsLevel { ADMIN, GUEST }

record Principal(RightsLevel rights) {
   public boolean canOpen() {
       return rights == RightsLevel.ADMIN;
   }
}

record Request(boolean authorized) { }

class Server {
   final static ThreadLocal<Principal> PRINCIPAL =
                                new ThreadLocal<>();

   String serve(Request request) {
       var level = request.authorized() ?
            RightsLevel.ADMIN : RightsLevel.GUEST;
       PRINCIPAL.set( new Principal(level) );
       return Application.handle( request );
   }
}
```

```
static class DBConnection {
   static DBConnection open() {
       var principal = Server.PRINCIPAL.get();
       if (!principal.canOpen())
            throw new IllegalArgumentException();
       return new DBConnection();
   }

   String doQuery() {
       return "Result";
   }
}

static class Application {
   public static String handle(Request request) {
       return DBConnection.open().doQuery();
   }
}
```

ThreadLocals as thread scoped variables

A ThreadLocal is a construct that allows us to store data accessible only by a specific thread.

```
enum RightsLevel { ADMIN, GUEST }

record Principal(RightsLevel rights) {
   public boolean canOpen() {
       return rights == RightsLevel.ADMIN;
   }
}

record Request(boolean authorized) { }

class Server {
   final static ThreadLocal<Principal> PRINCIPAL =
                                new ThreadLocal<>();

   String serve(Request request) {
       var level = request.authorized() ?
            RightsLevel.ADMIN : RightsLevel.GUEST;
       PRINCIPAL.set( new Principal(level) );
       return Application.handle( request );
   }
}
```

```
static class DBConnection {
   static DBConnection open() {
       var principal = Server.PRINCIPAL.get();
       if (!principal.canOpen())
            throw new IllegalArgumentException();
       return new DBConnection();
   }

   String doQuery() {
       return "Result";
   }
}

static class Application {
   public static String handle(Request request) {
       return DBConnection.open().doQuery();
   }
}
```

ThreadLocals as (virtual) thread scoped variables

```
public static void main(String[] args) {
   var server = new Server();

   var authorized = Thread.ofVirtual().name("Authorized").unstarted(() -> callServer(server, true));
   var notAuthorized = Thread.ofVirtual().name("NOT Authorized").unstarted(() -> callServer(server, false));

   authorized.start();
   notAuthorized.start();

   try { Thread.sleep(1000L); } catch (InterruptedException e) { throw new RuntimeException(e); }
}

private static void callServer(Server server, boolean auth) {
   var result = server.serve(new Request(auth));
   System.out.println( "thread " + Thread.currentThread().getName() + " got result " + result );
}
```

ThreadLocals as (virtual) thread scoped variables

```java
public static void main(String[] args) {
    var server = new Server();

    var authorized = Thread.ofVirtual().name("Authorized").unstarted(() -> callServer(server, true));
    var notAuthorized = Thread.ofVirtual().name("NOT Authorized").unstarted(() -> callServer(server, false));

    authorized.start();
    notAuthorized.start();

    try { Thread.sleep(1000L); } catch (InterruptedException e) { throw new RuntimeException(e); }
}

private static void callServer(Server server, boolean auth) {
    var result = server.serve(new Request(auth));
    System.out.println( "thread " + Thread.currentThread().getName() + " got result " + result );
}
```

```
Exception in thread "NOT Authorized" java.lang.IllegalArgumentException
    at org.mfusco.loom.experiments.threadlocal.ThreadLocalMain$DBConnection.open(ThreadLocalMain.java:48)
    at org.mfusco.loom.experiments.threadlocal.ThreadLocalMain$Application.handle(ThreadLocalMain.java:41)

thread Authorized got result Result
```

ThreadLocals as (virtual) thread scoped variables

So? Everything is fine, right?
What is the problem to use
ThreadLocal with virtual threads?

ThreadLocals as (virtual) thread scoped variables

So? Everything is fine, right?
What is the problem to use
ThreadLocal with virtual threads?

- That we can have a **huge number of virtual threads**, and each virtual thread will have its own ThreadLocal
  - The **memory footprint** of the application may quickly become **very high**
  - ThreadLocal will be **useless in a one-thread-per-request scenario** since data won't be shared between different requests

ScopedValues to the rescue

```
class Server {
   final static ThreadLocal<Principal> PRINCIPAL = new ThreadLocal<>();

   String serve(Request request) {
       var level = request.authorized() ? RightsLevel.ADMIN : RightsLevel.GUEST;
       PRINCIPAL.set( new Principal(level) );
       return Application.handle( request );
   }
}
```

ScopedValues to the rescue (still preview in JDK 21)

```java
class Server {
   final static ThreadLocal<Principal> PRINCIPAL = new ThreadLocal<>();

   String serve(Request request) {
       var level = request.authorized() ? RightsLevel.ADMIN : RightsLevel.GUEST;
       PRINCIPAL.set( new Principal(level) );
       return Application.handle( request );
   }
}
```


```java
class Server {
   final static ScopedValue<Principal> PRINCIPAL =
                                       ScopedValue.newInstance();

   String serve(Request request) {
       var level = request.authorized() ?
                           RightsLevel.ADMIN : RightsLevel.GUEST;
       try {
           return ScopedValue.where(PRINCIPAL, new Principal(level))
                             .call(() -> Application.handle(request));
       } catch (Exception e) { throw new RuntimeException(e); }
   }
}
```

ThreadLocals as Cache / Objects Pool ???

Do ScopedValues cover all the possible typical usage patterns of ThreadLocals?

ThreadLocals as Cache / Objects Pool ???

Do ScopedValues cover all the possible typical usage patterns of ThreadLocals?

- It's very common to use a ThreadLocal as a **thread-safe object pool**
  - **Easy** to implement and use
  - Does **not** require an **explicit lifecycle**: it is not necessary to put the pooled resource back into the pool when finished

ThreadLocals as Cache / Objects Pool ???

Do ScopedValues cover all the possible typical usage patterns of ThreadLocals?

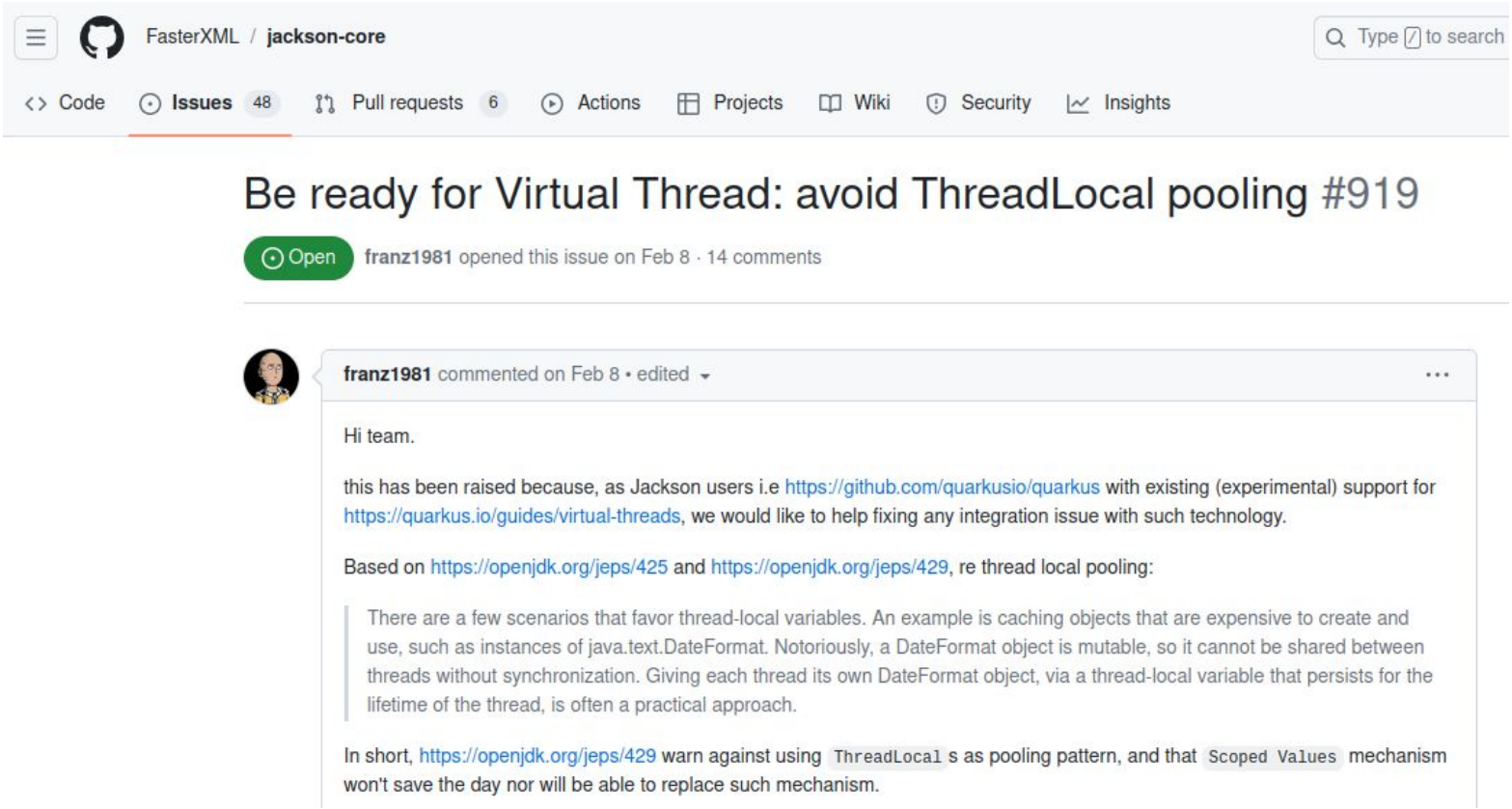
- It's very common to use a ThreadLocal as a **thread-safe object pool**
  - **Easy** to implement and use
  - Does **not** require an **explicit lifecycle**: it is not necessary to put the pooled resource back into the pool when finished
  - ScopedValues don't help in this scenario
  - This pattern **plays particularly bad with virtual threads**

ThreadLocals as Cache / Objects Pool ???

Do ScopedValues cover all the possible typical usage patterns of ThreadLocals?

- It's very common to use a ThreadLocal as a **thread-safe object pool**
  - Easy to implement and use
  - Doesn't require an explicit lifecycle: it is not necessary to put the pooled resource back into the pool when finished
  - ScopedValues don't help in this scenario
  - This pattern **plays particularly bad with virtual threads**

ThreadLocals as Cache / Objects Pool into the wild …



FasterXML / jackson-core

Type / to search

Code | Issues 48 | Pull requests 6 | Actions | Projects | Wiki | Security | Insights

Be ready for Virtual Thread: avoid ThreadLocal pooling #919

Open **franz1981** opened this issue on Feb 8 · 14 comments

franz1981 commented on Feb 8 • edited

Hi team.

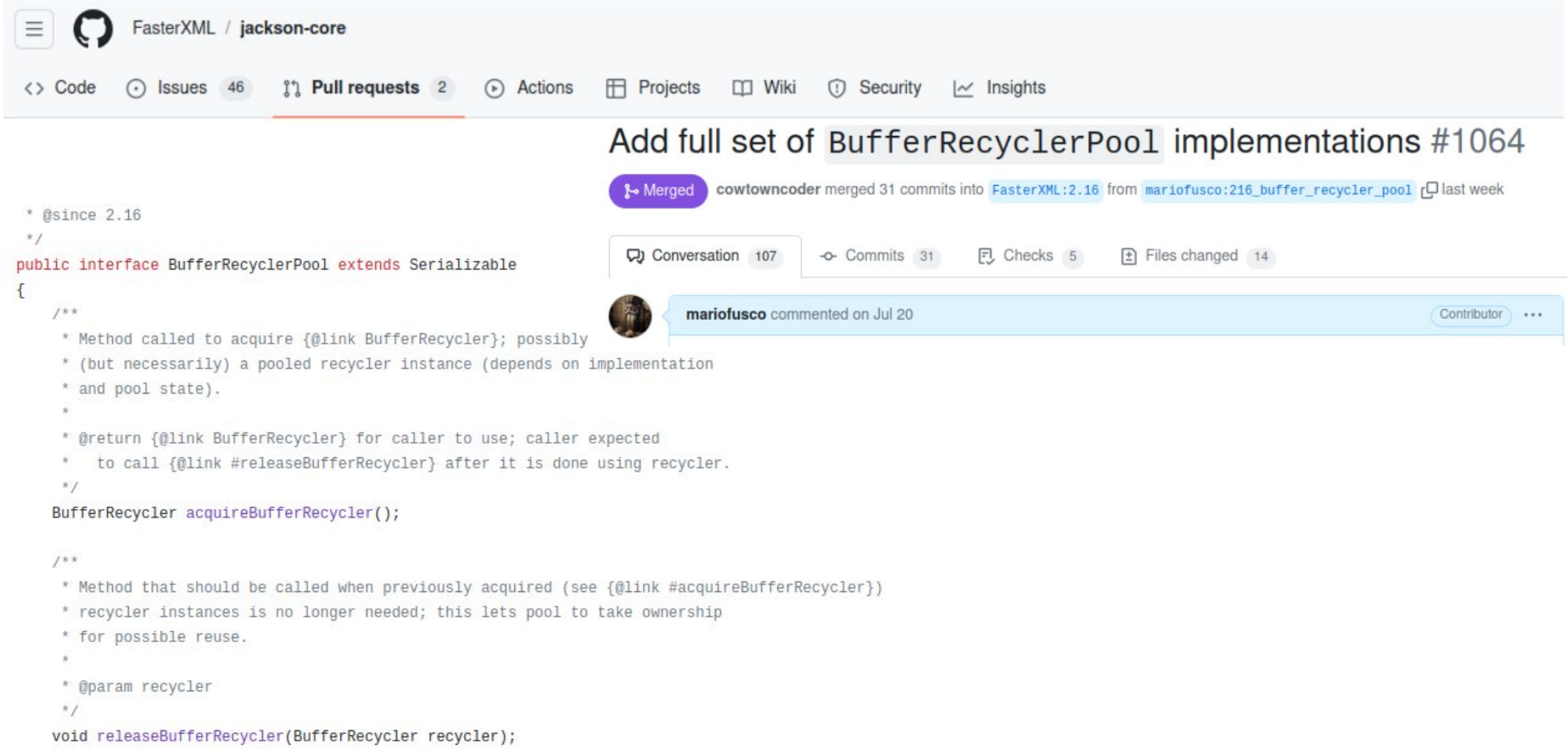
this has been raised because, as Jackson users i.e https://github.com/quarkusio/quarkus with existing (experimental) support for https://quarkus.io/guides/virtual-threads, we would like to help fixing any integration issue with such technology.

Based on https://openjdk.org/jeps/425 and https://openjdk.org/jeps/429, re thread local pooling:

> There are a few scenarios that favor thread-local variables. An example is caching objects that are expensive to create and use, such as instances of java.text.DateFormat. Notoriously, a DateFormat object is mutable, so it cannot be shared between threads without synchronization. Giving each thread its own DateFormat object, via a thread-local variable that persists for the lifetime of the thread, is often a practical approach.

In short, https://openjdk.org/jeps/429 warn against using `ThreadLocal`s as pooling pattern, and that `Scoped Values` mechanism won't save the day nor will be able to replace such mechanism.

… and the fix 🎉



FasterXML / jackson-core

Code | Issues 46 | Pull requests 2 | Actions | Projects | Wiki | Security | Insights

Add full set of `BufferRecyclerPool` implementations #1064

Merged **cowtowncoder** merged 31 commits into `FasterXML:2.16` from `mariofusco:216_buffer_recycler_pool` last week

Conversation 107 | Commits 31 | Checks 5 | Files changed 14

mariofusco commented on Jul 20 — Contributor

```
 * @since 2.16
 */
public interface BufferRecyclerPool extends Serializable
{
    /**
     * Method called to acquire {@link BufferRecycler}; possibly
     * (but necessarily) a pooled recycler instance (depends on implementation
     * and pool state).
     *
     * @return {@link BufferRecycler} for caller to use; caller expected
     *   to call {@link #releaseBufferRecycler} after it is done using recycler.
     */
    BufferRecycler acquireBufferRecycler();

    /**
     * Method that should be called when previously acquired (see {@link #acquireBufferRecycler})
     * recycler instances is no longer needed; this lets pool to take ownership
     * for possible reuse.
     *
     * @param recycler
     */
    void releaseBufferRecycler(BufferRecycler recycler);
```

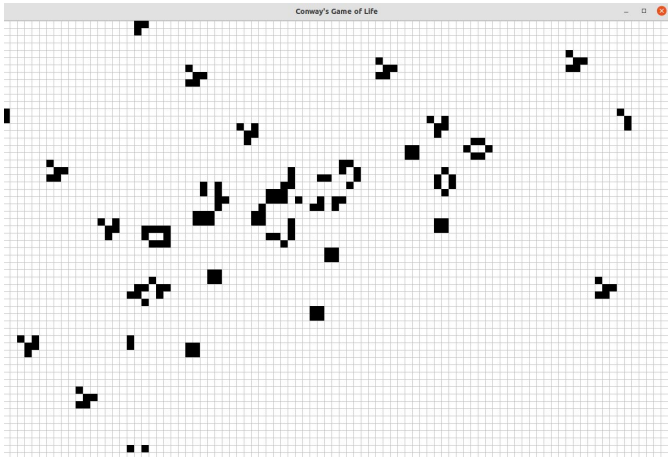
Plugging a different thread pool as carriers

Can we outperform the default Fork/Join pool based carrier thread pool?

Plugging a different thread pool as carriers

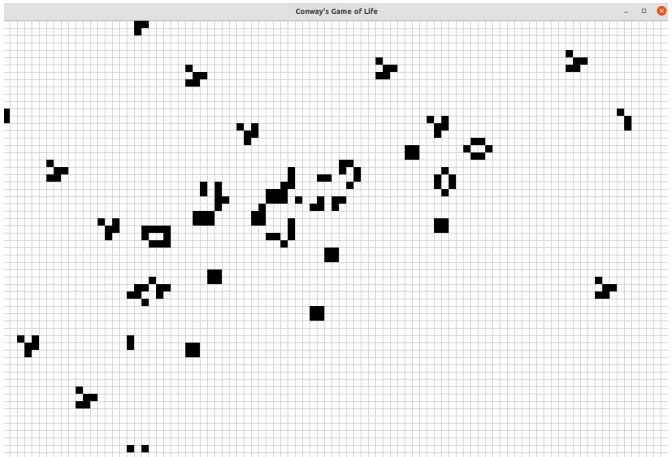
Can we outperform the default Fork/Join pool based carrier thread pool?

Let’s put at work Game of Life again!



Plugging a different thread pool as carriers

Can we outperform the default Fork/Join pool based carrier thread pool?

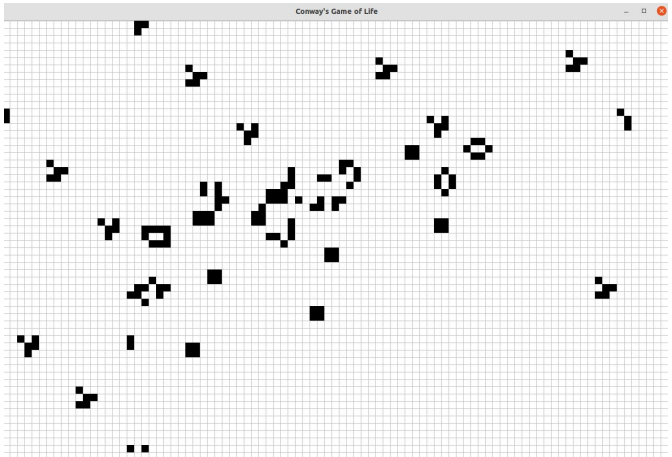


Let’s put at work Game of Life again!

Plugging a different thread pool as carriers

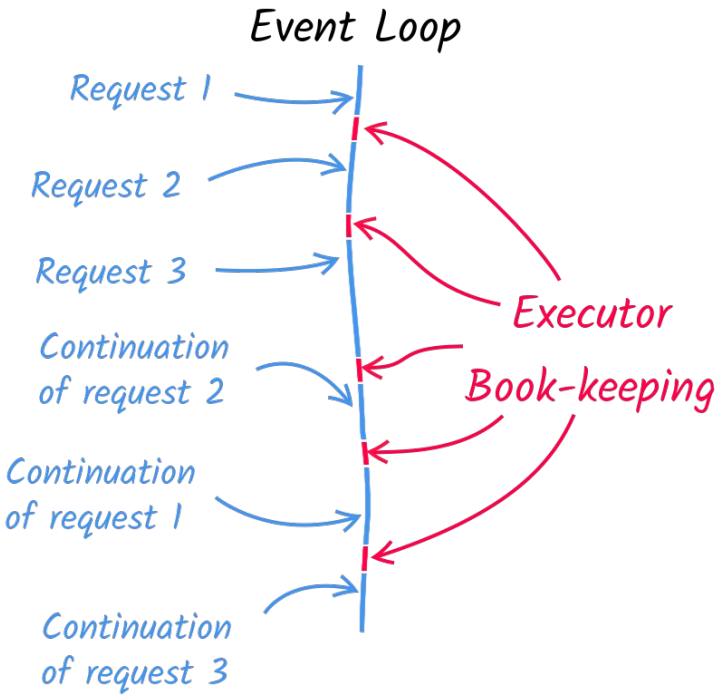
Can we outperform the default Fork/Join pool based carrier thread pool?

Let’s put at work Game of Life again!

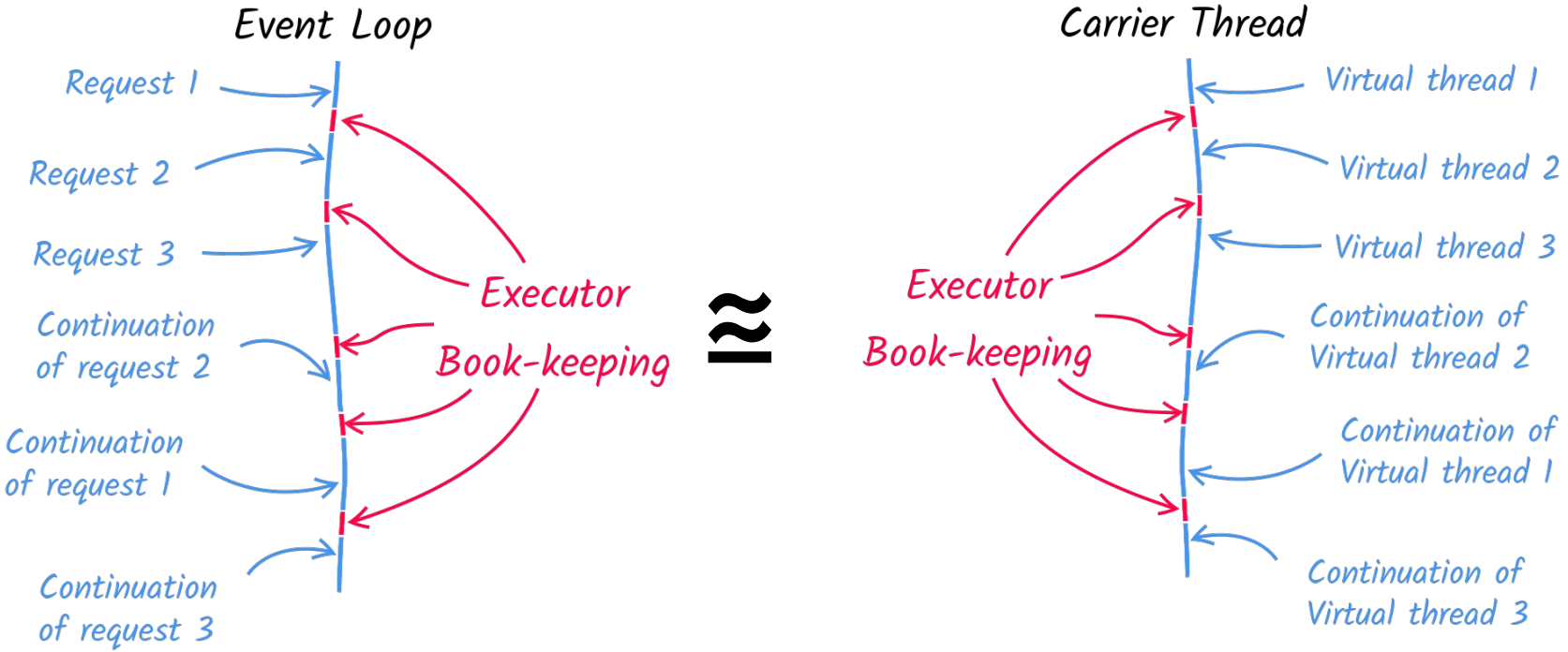



```
Benchmark                                (executionStrategy)   Mode  Cnt    Score   Error  Units
GameOfLifeBenchmark.benchmark                         Native  thrpt   40   40.317 ± 0.353  ops/s
GameOfLifeBenchmark.benchmark                 ForkJoinVirtual  thrpt   40  244.639 ± 1.419  ops/s
GameOfLifeBenchmark.benchmark  FixedCarrierPoolVirtual  thrpt   40  129.525 ± 5.346  ops/s
GameOfLifeBenchmark.benchmark     PinnedCarrierVirtual  thrpt   40   59.071 ± 1.532  ops/s
```

So why we may want to plug a different carrier?
The Quarkus use case - Reactive ≅ Virtual Threads

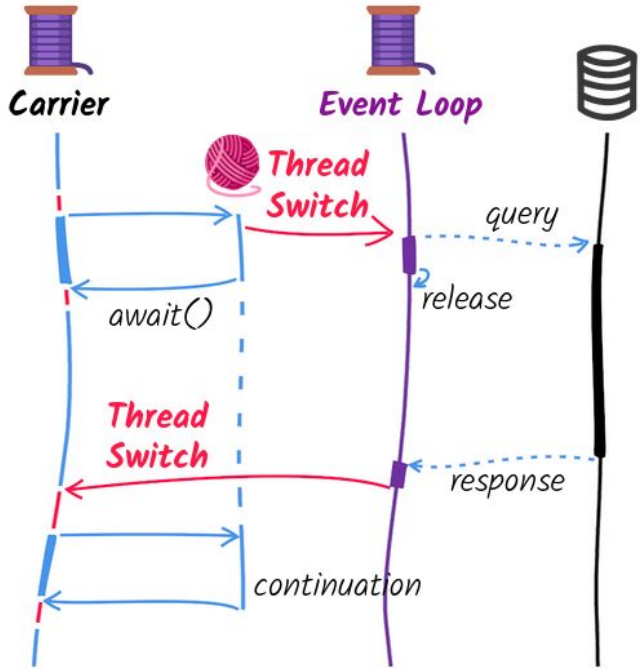


So why we may want to plug a different carrier?
The Quarkus use case - Reactive ≅ Virtual Threads



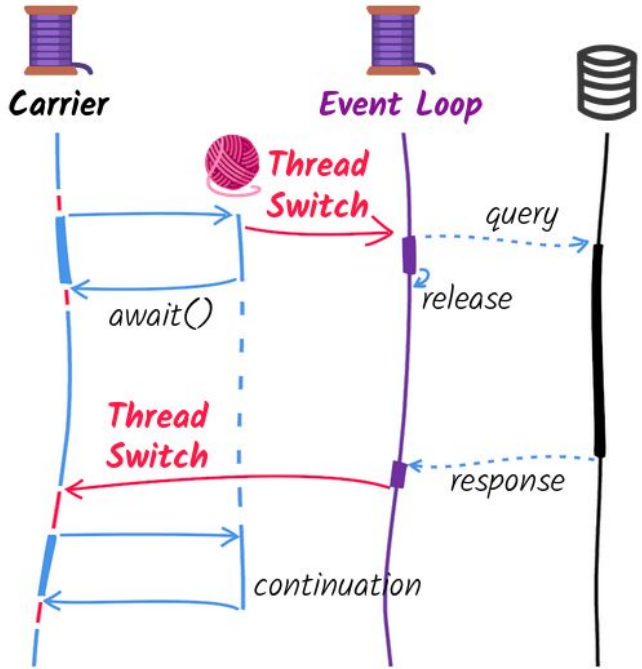
So why we may want to plug a different carrier?
The Quarkus use case - The Event Loop as carrier

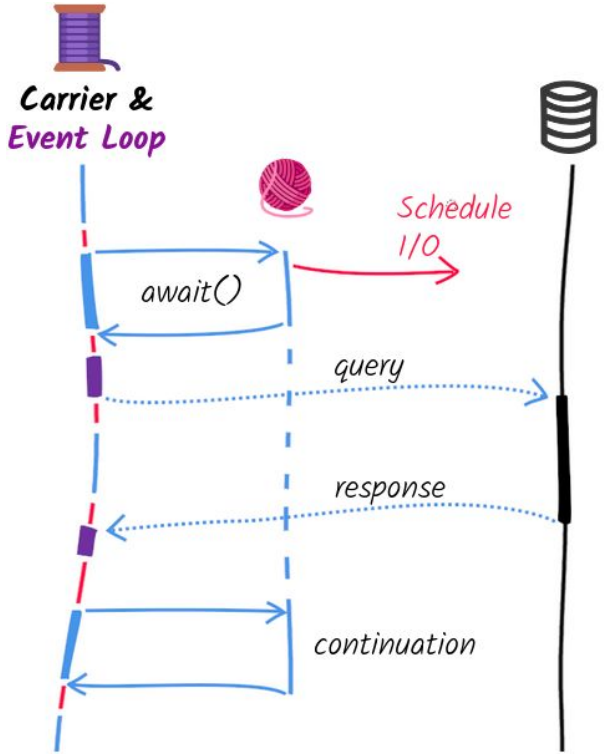
```
@RunOnVirtualThread
```



So why we may want to plug a different carrier?
The Quarkus use case - The Event Loop as carrier

@RunOnVirtualThread





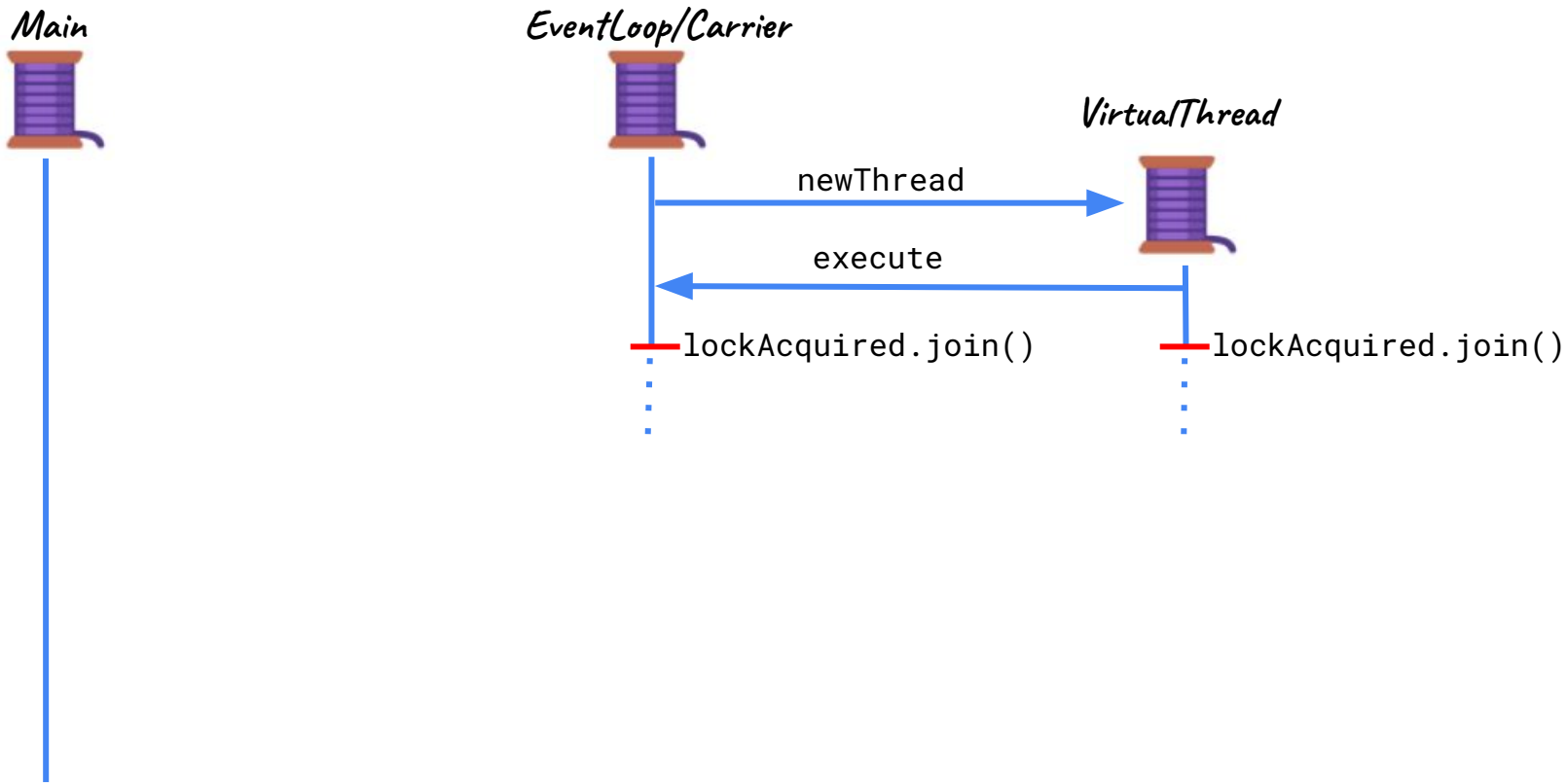
Why not using the event-loop as carrier thread???

- No (more) API to do that in Loom - API removed in fall 2021

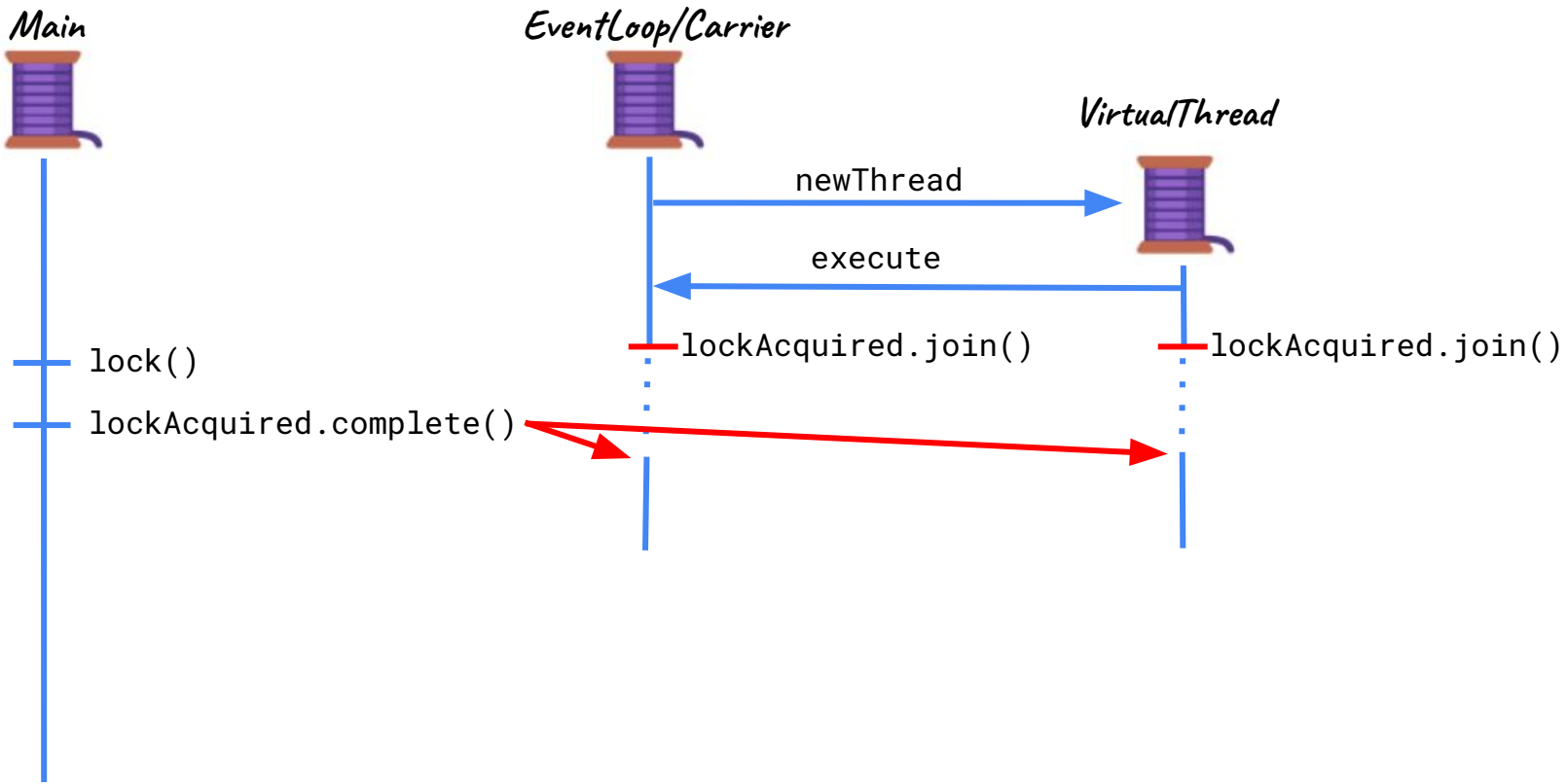
Why not using the event-loop as carrier thread???

- No (more) API to do that in Loom - API removed in fall 2021
- It lead to…. **deadlocks!**

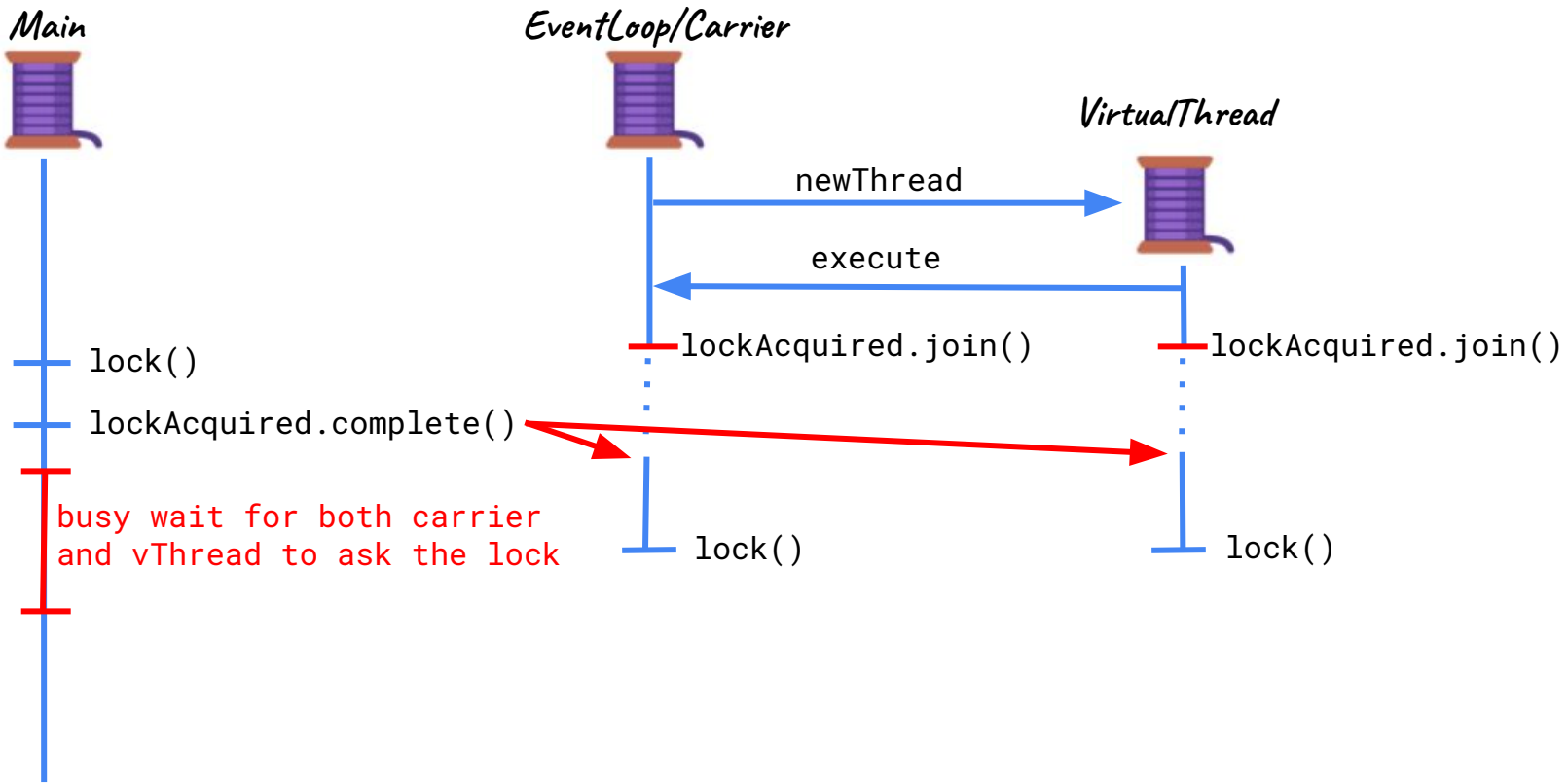
What’s going on ??? 🤔



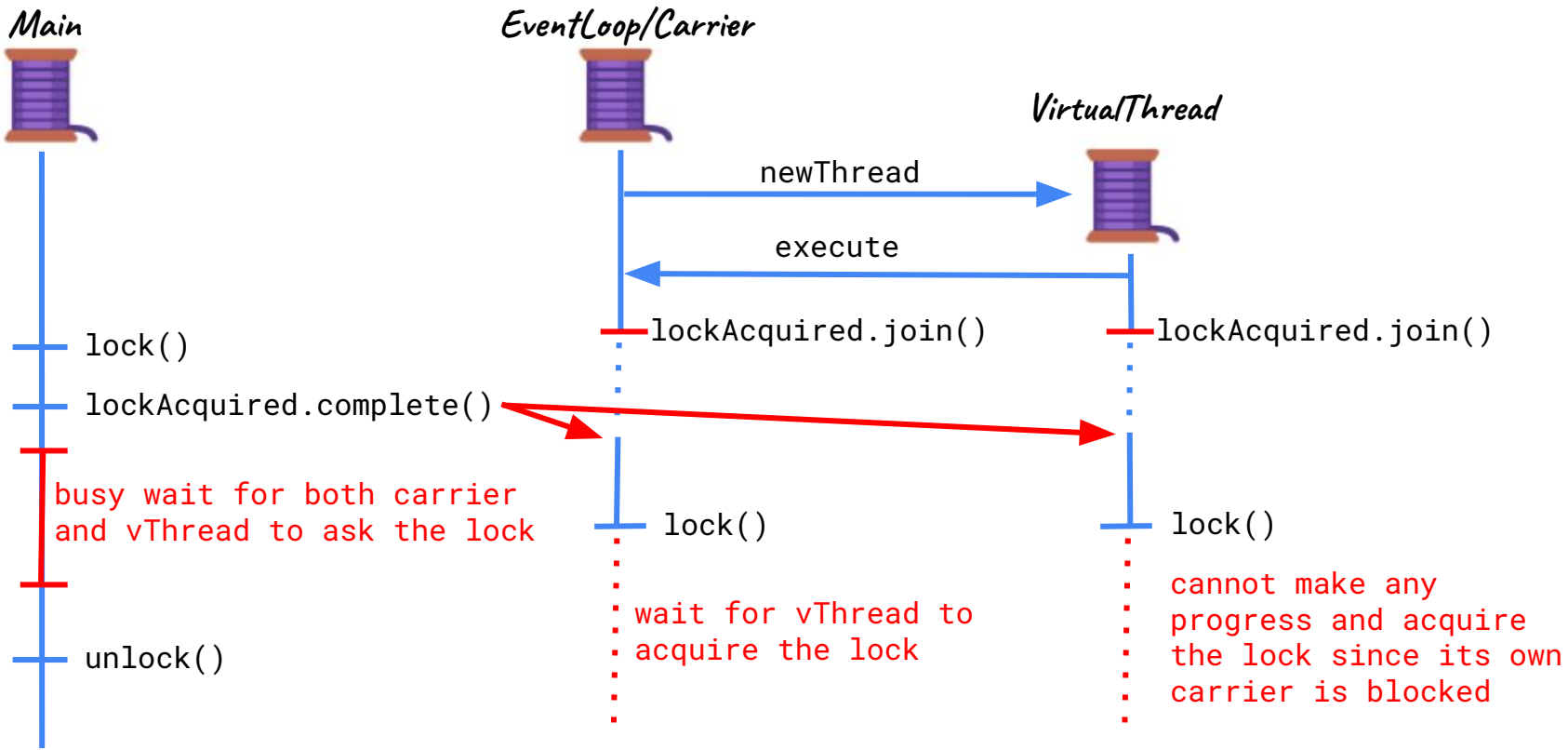
What’s going on ??? 🤔



What's going on ??? 🤔



What's going on ??? 🤔



Conclusions

- Virtual threads are **not faster** threads ⇒ they don't magically execute more instructions per second than native ones do.
- Virtual threads are **not drop in replacement** for native threads ⇒ they are a different (new) tool and you have to **know their features and characteristics** in order to use them appropriately.
- Virtual threads are generally **not a good fit** for CPU bound tasks ⇒ **lack of fairness** and the performance **cost of coroutines** can be problematic.
- What virtual threads are really good for is **waiting** ⇒ their goal is **maximizing the utilization of external resources** and then **improving throughput**, without affecting readability.

Learn More

Quarkus Virtual Threads

- REST / HTTP
- Hibernate, Bean Validation, Transactions
- Kafka, AMQP, JMS
- gRPC
- Scheduled tasks
- Event Bus
- ...