Virtual Threads
Strengths and Pitfalls

vJUG

The worlds first virtual
Java User Group

Victor Rentea

👋 I'm **Victor Rentea** 🇷🇴 Java Champion, PhD(CS)

18 years of **Java**, .kt, .js/ts, .scala, .cs, .py ...

10 years of **training** at 150+ companies in 20 countries on:

- Architecture, Refactoring, Unit Testing
- Spring, Hibernate, Performance, Reactive

European Software Crafters **Community** (on Meetup.com)

▶ YouTube.com/vrentea

Life += 👰 + 👧 + 🙇 + 🐈

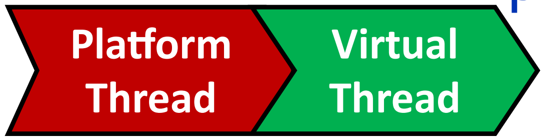
@victorrentea

victorrentea.ro

Before Java 21:

Threads are blocked during I/O

keeping a stack of 0.5 MB RAM for the duration of any (long) call


Platform Thread

Virtual Thread

```
public ABC abc(int id) {
  var a = api.a(id); // REST call
  var b = api.b(a); // REST call
  var c = api.c(a, b); // REST call
  return new ABC(a, b, c);
}
```

What's wrong here?

If using Virtual Threads (Java 21 ⭐star feature):

JVM unmounts the Platform Thread during I/O.

Only a light virtual thread (< 1 KB) blocks in the call.

The OS Platform Thread is used to run other virtual threads.

You keep your code clean and let JVM handle it🍻.

Alternative: non-blocking Concurrency

```
// with CompletableFuture since Java 8
public CompletableFuture<ABC> abc(String id) {
  return supplyAsync(() -> api.a(id), executor)
      .thenCompose(a -> api.b(a)
          .thenCompose(b -> api.c(a, b)
              .thenApply(c -> new ABC(a, b, c))));
}

// with Reactive Programming☠️ in Reactor, rxJava,...
public Mono<ABC> abc(int id) {
   return api.a(id)
        .flatMap(a -> api.b(a).flatMap(
                b -> api.c(a, b).map(
                        c -> new ABC(a, b, c))));
}
```

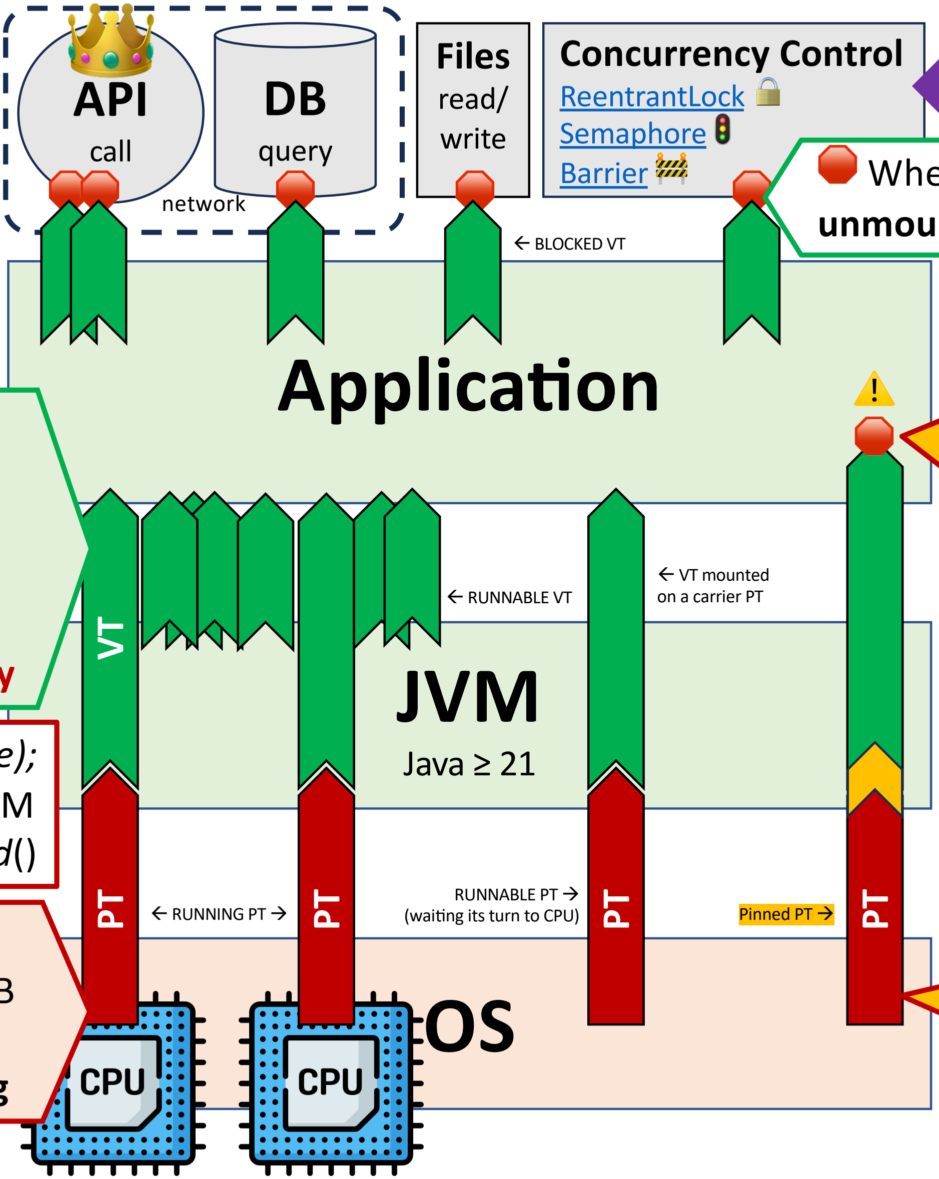
Some libs (eg Jackson) *assume* threads are reused via a pool and **cache data in ThreadLocals**

No need to reuse **VT**s via a Thread Pool – they **are cheap**

Virtual Threads (VT) = cheap
- Java call stack
- **small**, resizable stack: 1-2 KB,
- **many**: 100K .. 1M!
- runs on CPU on a *carrier* **PT**
- scheduled by JVM **collaboratively**

CPU Monopolization eg:*while(true);*
If a CPU-only task never blocks, JVM cannot interrupt its **VT** 👉add *yield*()

Platform Threads (PT)
- **heavy**, fixed-size stack: 0.25-1 MB
- **few**: < 1K .. 10K
- scheduled by OS via **time-sharing**



refactor synchronized to

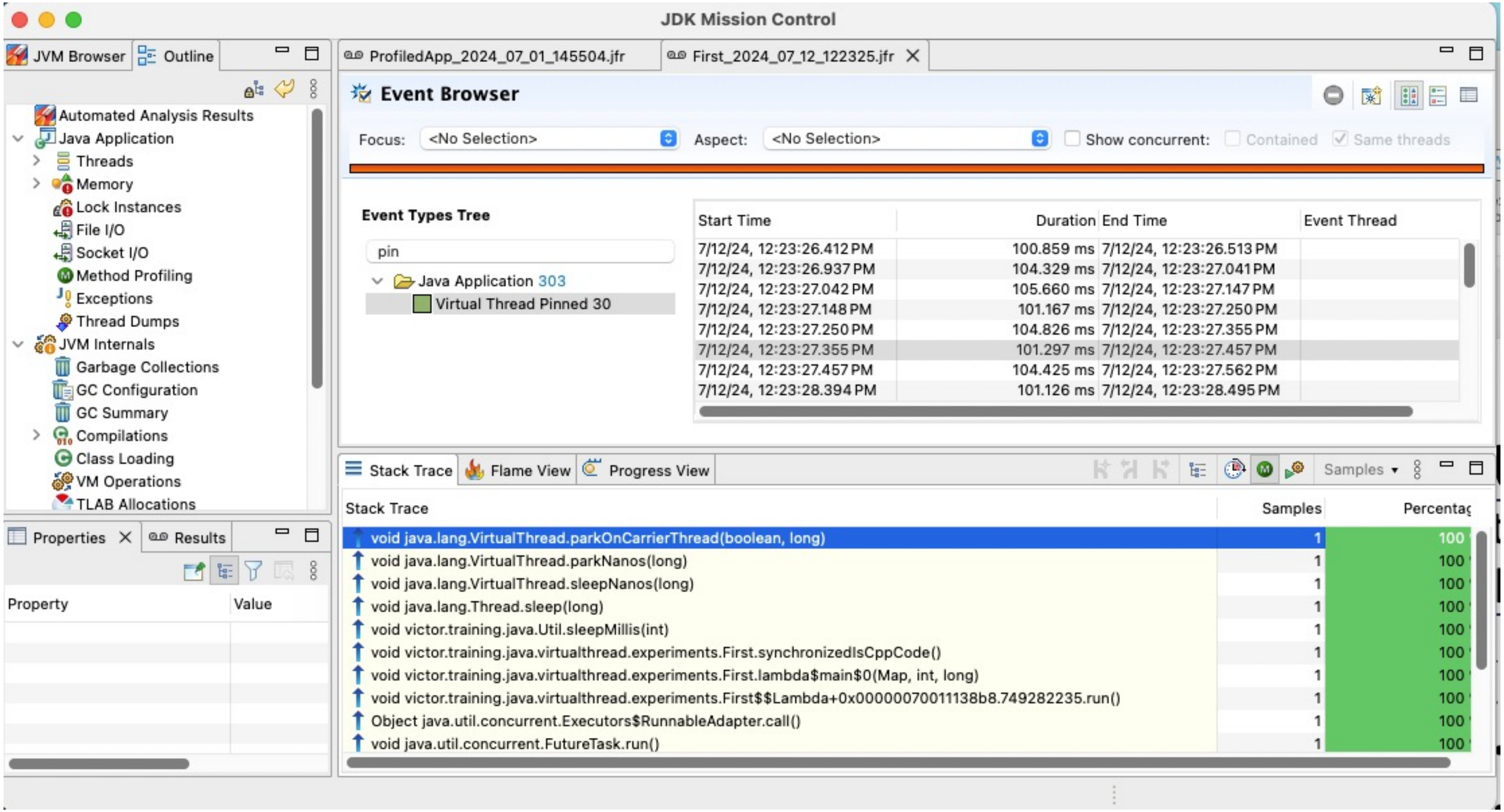
🛑 When a **VT** blocks in **Java code**, JVM **unmounts the PT** to run another **VT**

⚠ **Pinning** of the VT on PT when it blocks in C++ **native** code or **synchronized** block 👉 **refactor>**
trace via: -Djdk.tracePinnedThreads=full

⚠ If JVM runs out of **PTs**, it can:
➔ Let some **VT** starve ➔ not fair☹
➔ Use more **PT** ➔ memory++☹
-Djdk.virtualThreadScheduler.parallelism=2
-Djdk.virtualThreadScheduler.maxPoolSize=2

⚠ **Deadlocks**: VT run on an pool of PTs (potentially exhausted?)

Detecting Thread Pinning using JFR Events



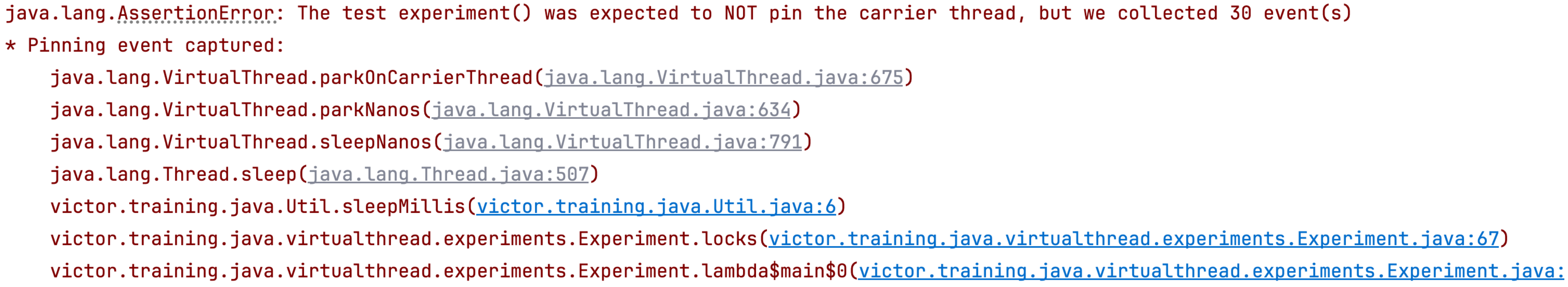
Start Time	Duration	End Time	Event Thread
7/12/24, 12:23:26.412 PM	100.859 ms	7/12/24, 12:23:26.513 PM	
7/12/24, 12:23:26.937 PM	104.329 ms	7/12/24, 12:23:27.041 PM	
7/12/24, 12:23:27.042 PM	105.660 ms	7/12/24, 12:23:27.147 PM	
7/12/24, 12:23:27.148 PM	101.167 ms	7/12/24, 12:23:27.250 PM	
7/12/24, 12:23:27.250 PM	104.826 ms	7/12/24, 12:23:27.355 PM	
7/12/24, 12:23:27.355 PM	101.297 ms	7/12/24, 12:23:27.457 PM	
7/12/24, 12:23:27.457 PM	104.425 ms	7/12/24, 12:23:27.562 PM	
7/12/24, 12:23:28.394 PM	101.126 ms	7/12/24, 12:23:28.495 PM	

Stack Trace	Samples	Percentag
void java.lang.VirtualThread.parkOnCarrierThread(boolean, long)	1	100
void java.lang.VirtualThread.parkNanos(long)	1	100
void java.lang.VirtualThread.sleepNanos(long)	1	100
void java.lang.Thread.sleep(long)	1	100
void victor.training.java.Util.sleepMillis(int)	1	100
void victor.training.java.virtualthread.experiments.First.synchronizedIsCppCode()	1	100
void victor.training.java.virtualthread.experiments.First.lambda$main$0(Map, int, long)	1	100
void victor.training.java.virtualthread.experiments.First$$Lambda+0x00000070011138b8.749282235.run()	1	100
Object java.util.concurrent.Executors$RunnableAdapter.call()	1	100
void java.util.concurrent.FutureTask.run()	1	100

Detecting Thread Pinning via Tests

```java
@ExtendWith(LoomUnitExtension.class)
public class ExperimentTest {
  @Test
  @ShouldNotPin
  void experiment() throws Exception {
    experiment.execute();
  }
}
```

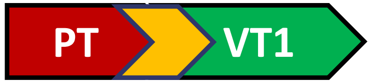
```xml
<dependency>
  <groupId>me.escoffier.loom</groupId>
  <artifactId>loom-unit</artifactId>
  <version>0.3.0</version>
  <scope>test</scope>
</dependency>
```



```
java.lang.AssertionError: The test experiment() was expected to NOT pin the carrier thread, but we collected 30 event(s)
* Pinning event captured:
    java.lang.VirtualThread.parkOnCarrierThread(java.lang.VirtualThread.java:675)
    java.lang.VirtualThread.parkNanos(java.lang.VirtualThread.java:634)
    java.lang.VirtualThread.sleepNanos(java.lang.VirtualThread.java:791)
    java.lang.Thread.sleep(java.lang.Thread.java:507)
    victor.training.java.Util.sleepMillis(victor.training.java.Util.java:6)
    victor.training.java.virtualthread.experiments.Experiment.locks(victor.training.java.virtualthread.experiments.Experiment.java:67)
    victor.training.java.virtualthread.experiments.Experiment.lambda$main$0(victor.training.java.virtualthread.experiments.Experiment.java:
```

https://quarkus.io/guides/virtual-threads#testing-virtual-thread-applications

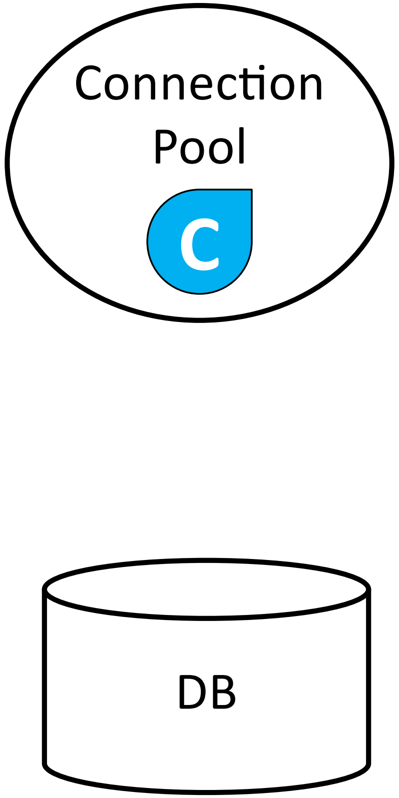
Virtual Threads + Resource Pool = Deadlock💞



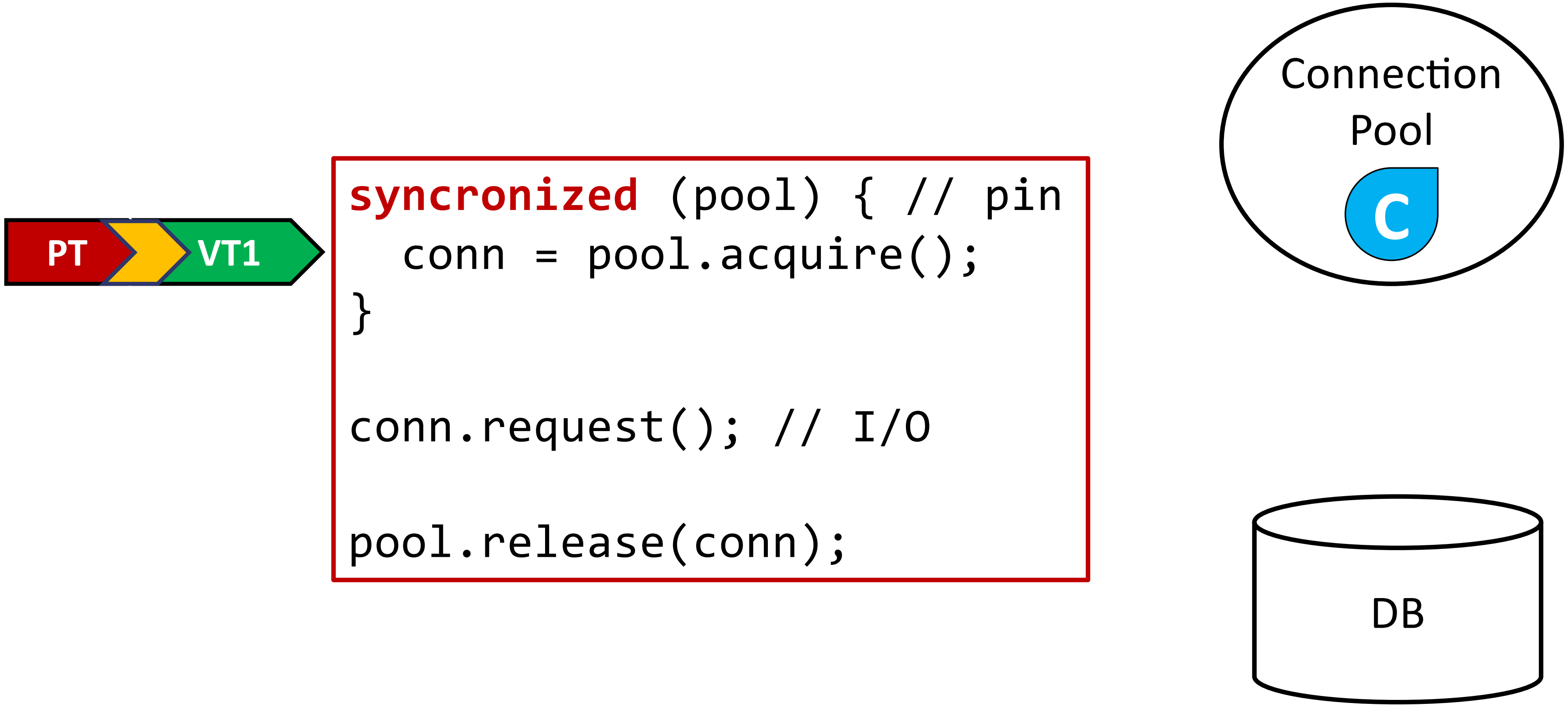
```
syncronized (pool) { // pin
  conn = pool.acquire();
}

conn.request(); // I/O

pool.release(conn);
```



Virtual Threads + Resource Pool = Deadlock💞

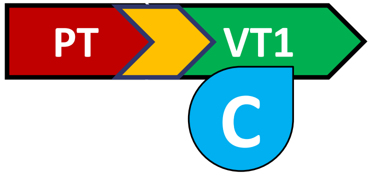


```
syncronized (pool) { // pin
  conn = pool.acquire();
}

conn.request(); // I/O

pool.release(conn);
```

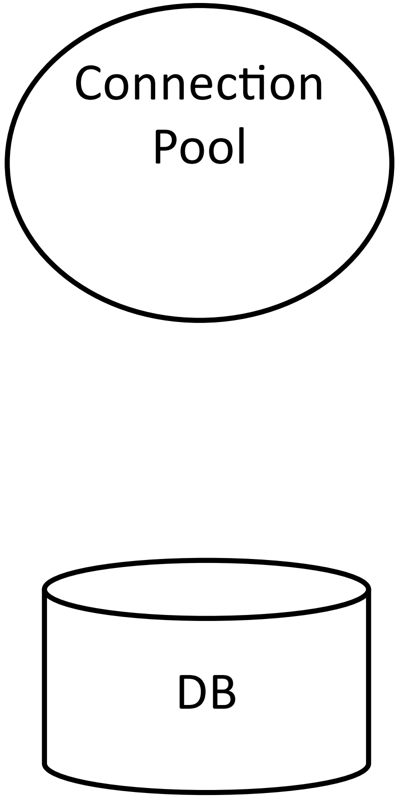
Virtual Threads + Resource Pool = Deadlock💞



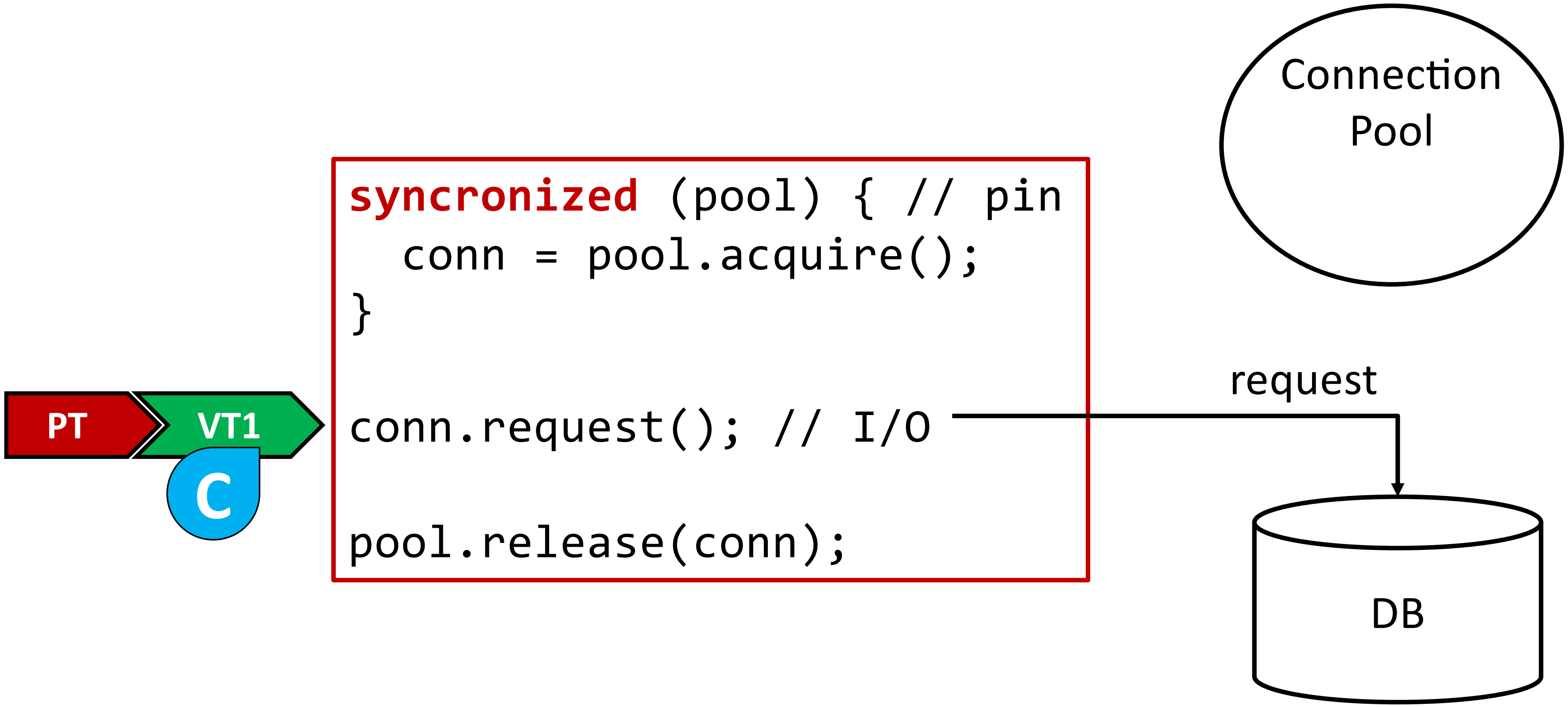
```
syncronized (pool) { // pin
  conn = pool.acquire();
}

conn.request(); // I/O

pool.release(conn);
```



Virtual Threads + Resource Pool = Deadlock💞

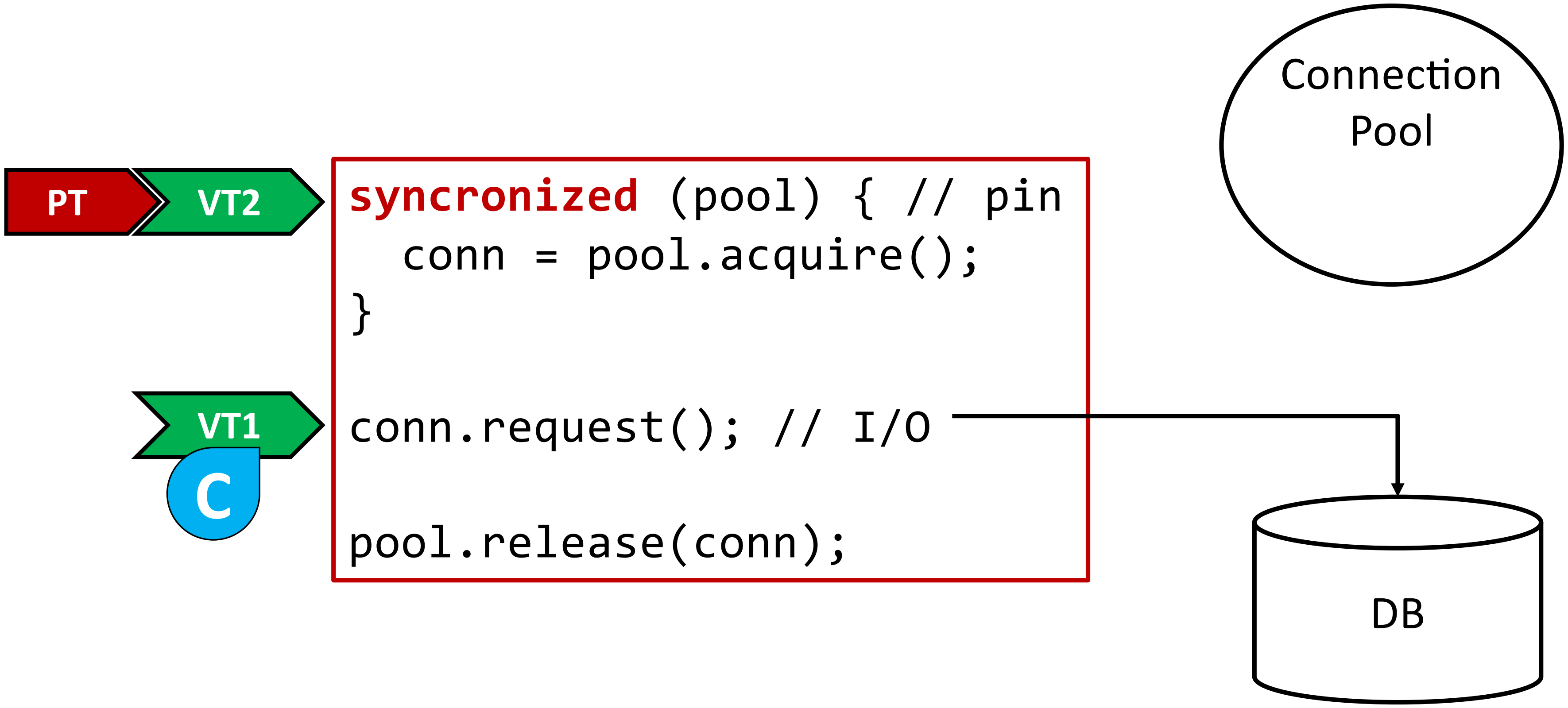


```
syncronized (pool) { // pin
  conn = pool.acquire();
}

conn.request(); // I/O

pool.release(conn);
```

Virtual Threads + Resource Pool = Deadlock💞

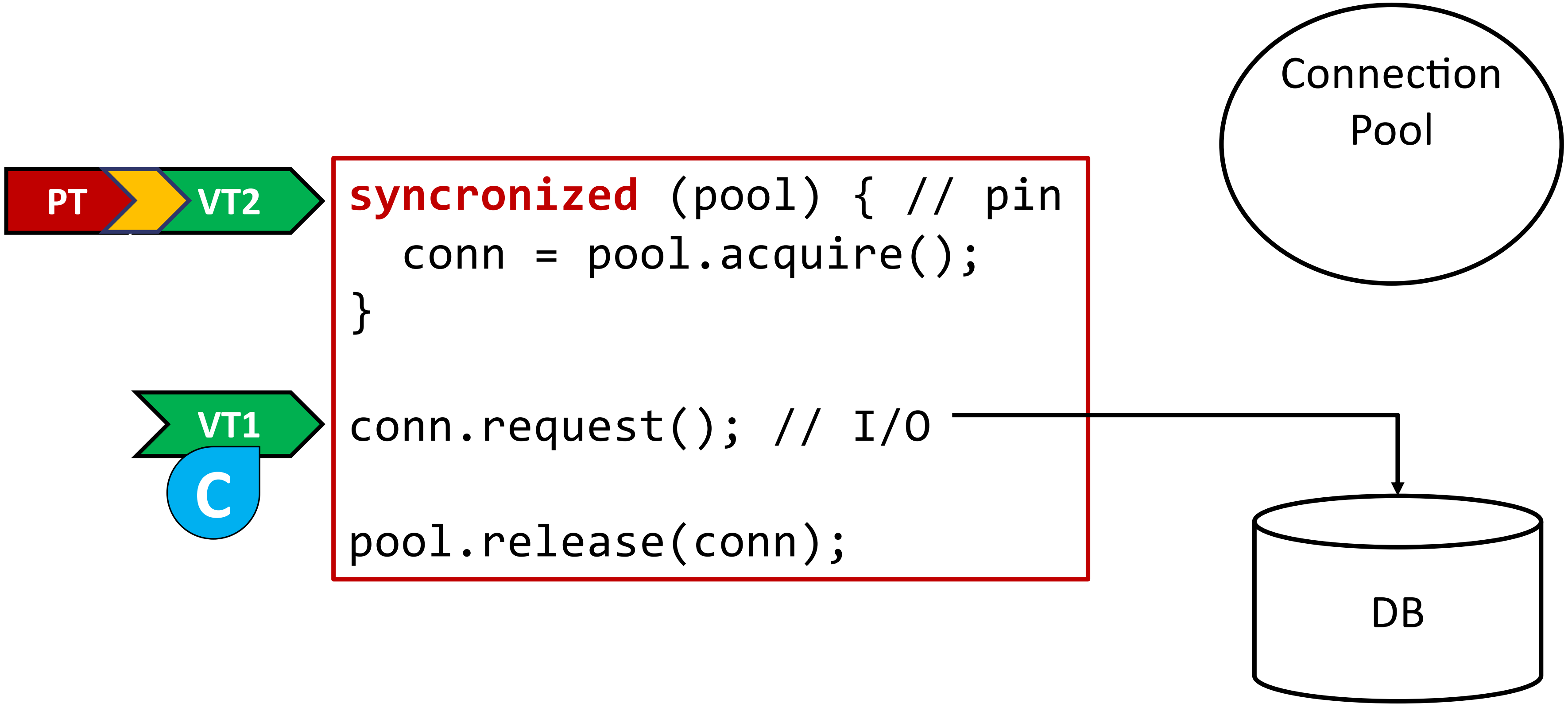


```
syncronized (pool) { // pin
  conn = pool.acquire();
}

conn.request(); // I/O

pool.release(conn);
```

Virtual Threads + Resource Pool = Deadlock💞

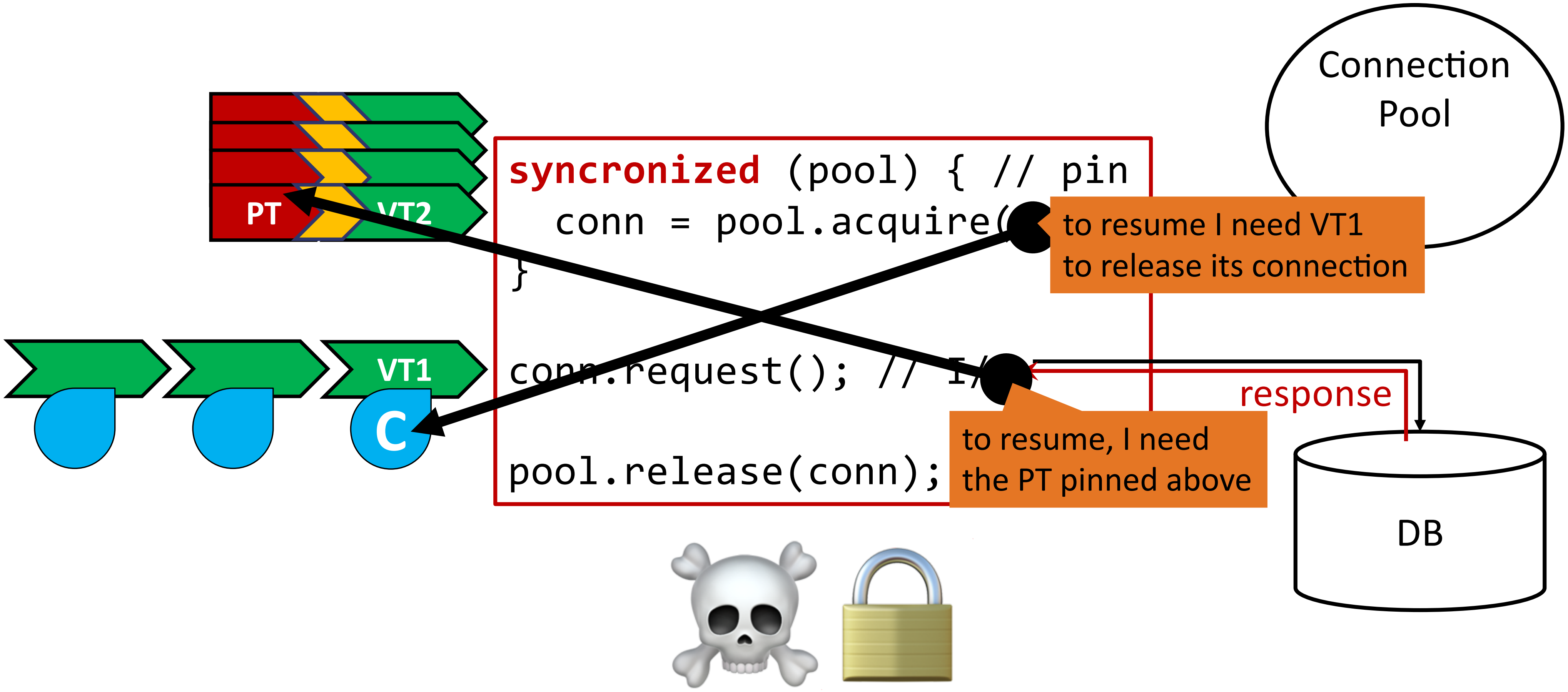


```
syncronized (pool) { // pin
  conn = pool.acquire();
}

conn.request(); // I/O

pool.release(conn);
```

Virtual Threads + Resource Pool = Deadlock💞



```
syncronized (pool) { // pin
   conn = pool.acquire(
}

conn.request(); // I/

pool.release(conn);
```

to resume I need VT1 to release its connection

to resume, I need the PT pinned above

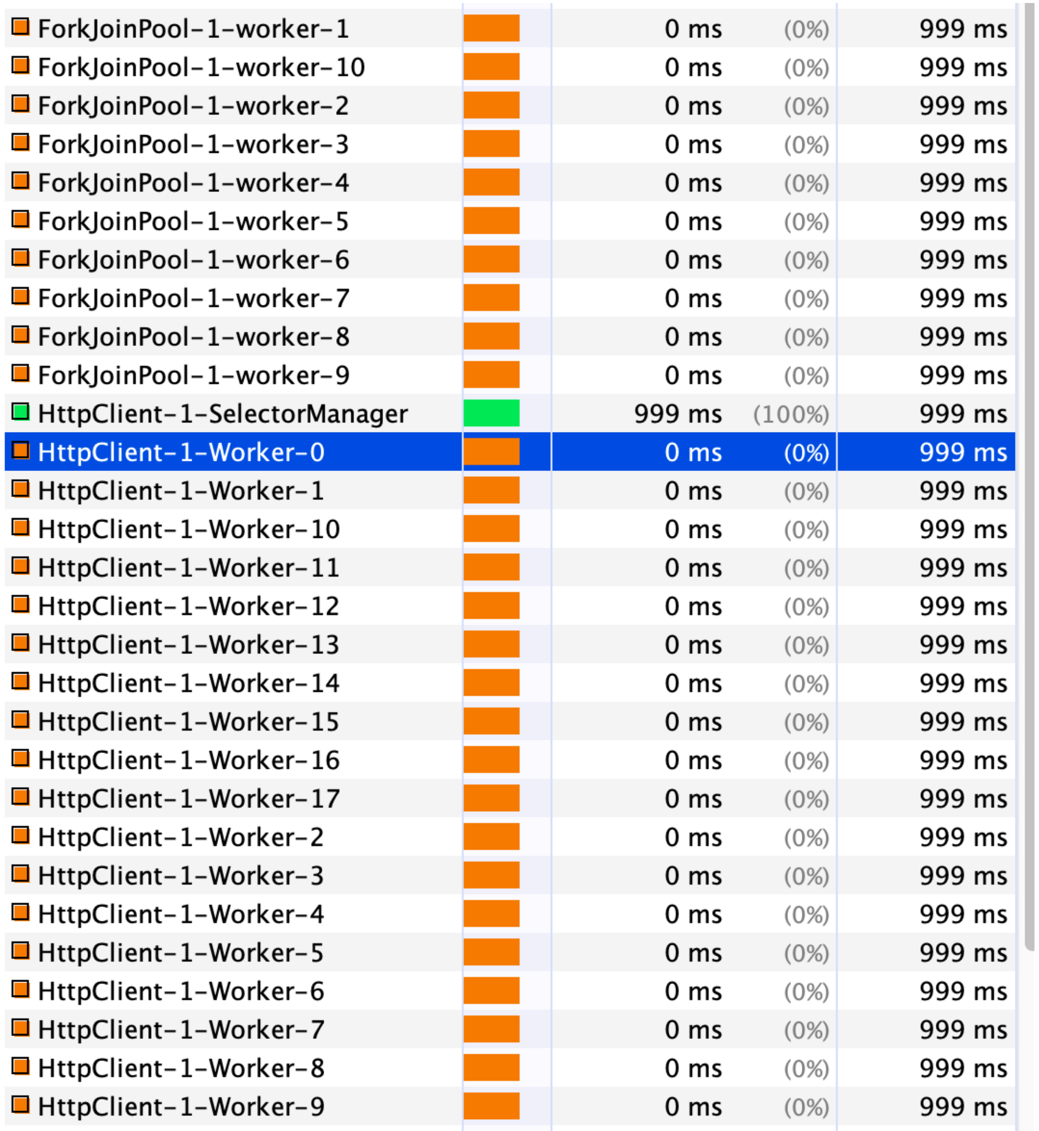
Virtual Threads + Resource Pool = Deadlock💞

tl;dr – the long story

- Resource = HTTP/JDBC/Redis... Connection
- Acquire is done via a synchronized block
  - Any incoming VT blocks pinned on the carrier PT
- Using the resource, Java runs an I/O blocking call
  - Example: network read => PT is released
- Last available PT brings a new VT to *acquire* => pin
- No PT is available to resume the VT work after network read

Old Habbits
Die Hard

An old library
(Java's HttpClient)
creating a new PT
for each request
😱



🟧 ForkJoinPool-1-worker-1	🟧	0 ms	(0%)	999 ms
🟧 ForkJoinPool-1-worker-10	🟧	0 ms	(0%)	999 ms
🟧 ForkJoinPool-1-worker-2	🟧	0 ms	(0%)	999 ms
🟧 ForkJoinPool-1-worker-3	🟧	0 ms	(0%)	999 ms
🟧 ForkJoinPool-1-worker-4	🟧	0 ms	(0%)	999 ms
🟧 ForkJoinPool-1-worker-5	🟧	0 ms	(0%)	999 ms
🟧 ForkJoinPool-1-worker-6	🟧	0 ms	(0%)	999 ms
🟧 ForkJoinPool-1-worker-7	🟧	0 ms	(0%)	999 ms
🟧 ForkJoinPool-1-worker-8	🟧	0 ms	(0%)	999 ms
🟧 ForkJoinPool-1-worker-9	🟧	0 ms	(0%)	999 ms
🟩 HttpClient-1-SelectorManager	🟩	999 ms	(100%)	999 ms
🟧 HttpClient-1-Worker-0	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-1	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-10	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-11	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-12	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-13	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-14	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-15	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-16	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-17	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-2	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-3	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-4	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-5	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-6	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-7	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-8	🟧	0 ms	(0%)	999 ms
🟧 HttpClient-1-Worker-9	🟧	0 ms	(0%)	999 ms

Old Habbits Die Hard

Long I/O in synchronized block

```
synchronized void bad() {
  // a network call
}
```

Virtual Threads – When / When NOT

- To **accept more parallel requests** with less instances (3, not 7)
  - Especially when calling other APIs a lot
  - Why: less memory blocked / call, VT context switch >>>> faster than PT
- **Won't help CPU-bound systems** (CPU 90-100%)
  - Add occasional Thread.*yield*() for fairness (avoid PT monopolization)
- **Won't help if current bottleneck is outside Java**
  - eg: increasing load on the single shared Database/MQ might be worse

Coding with Virtual Threads

- Avoid **synchronized** {io|locks} 👉 use ReentrantLock
- Reduce heap memory used / request = next bottleneck
- Protect remote systems 👉 Semaphore (not ThreadPools)
- Reduce the size of ThreadLocal data to keep VTs light
- Mind your libraries
  - @ShouldNotPin
  - -Djdk.tracePinnedThreads=short (or full)
  - 'Virtual Thread Pinned' event in JFR recording in [pre]prod

Legacy Code / Libraries can:

- ... pin VT to PT in **synchronized** blocks, leading to:
  - **Bottlenecks** starving carrier PTs when I/O in synchronized
  - **Deadlocks** with connection pools (JDBC, Redis..)
- ... create **new Platform Threads** (Java HttpClient)
- ... limit **max connections** (Apache HttpClient, JDBC)

Only way to see Virtual Threads stacks (as of Oct 2024): jcmd <pid> Thread.dump_to_file <file>

Dig More:

- Understand Virtual Threads
  - Intro - https://blog.rockthejvm.com/ultimate-guide-to-java-virtual-threads
  - Virtual Threads design explained by Lead of Project Loom@Oracle - https://youtu.be/EO9oMiL1fFo
  - Virtual Threads vs (Kotlin) coroutines by Venkat at jPrime'23 - https://youtu.be/uoTyIFvckXA
- Industry War Stories
  - medium.com/@phil_3582/java-virtual-threads-some-early-gotchas-to-look-out-for-f65df1bad0db
  - blog.ydb.tech/how-we-switched-to-java-21-virtual-threads-and-got-deadlock-in-tpc-c-for-postgresql-cca2fe08d70b
  - blog.ycrash.io/pitfalls-to-avoid-when-switching-to-virtual-threads/
  - https://www.infoq.com/news/2024/08/netflix-performance-case-study/
- Library support for Virtual Threads:
  - HikariCP: https://github.com/brettwooldridge/HikariCP/pull/2055
  - Jackson ✅: https://github.com/FasterXML/jackson-core/issues/919
  - Spring Boot ✅: https://spring.io/blog/2022/10/11/embracing-virtual-threads
  - Postgres JDBC ✅: https://jdbc.postgresql.org/changelogs/2023-03-17-42.6.0-release/