Project Loom - Einfachere Nebenläufigkeit in Java

Ein Thread, ist ein Thread, ist ein (virtueller) Thread.

Welcome

Java 19

Sept 20, 2022

```
sdk install java 19-open
openjdk 19 2022-09-20 build 19+36-2238
```

SDKMAN!

Why ?

Concurrency ?
Parallelism?

A)

B)

C)

D)

E)

F)

There is a lot to do in a Java Program

Image Credit Marcus Biel / cleancodeacademy

There is a lot to do!

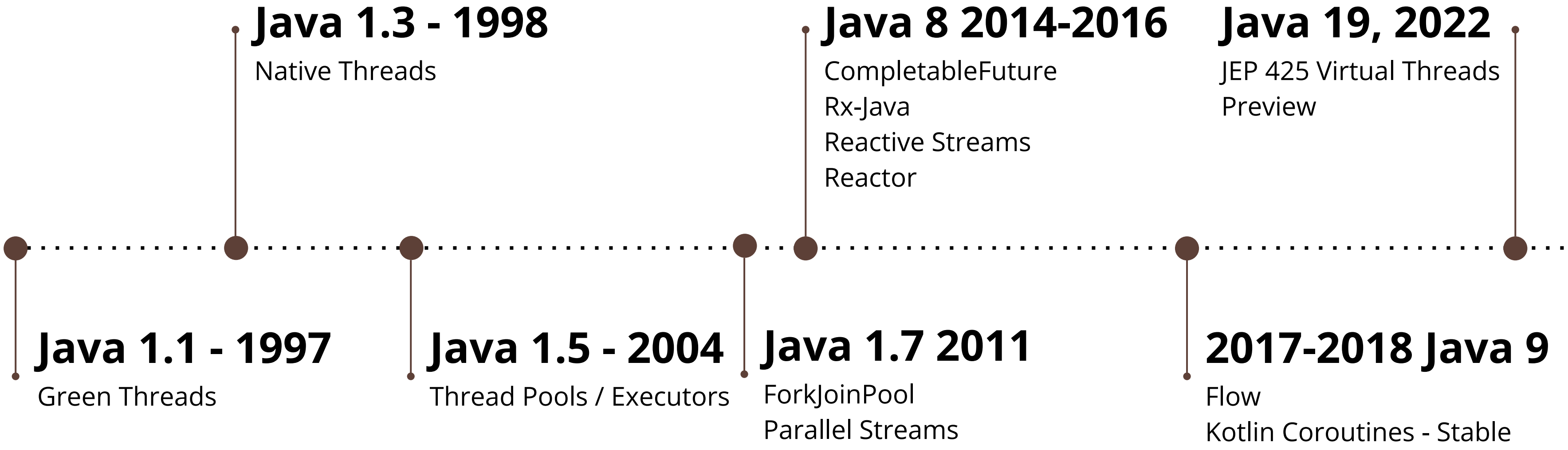
- Process and answer requests
- Compute
- I/O: Read and write files
- Read from databases and network
- Synchronization
- Render
- Logging / Monitoring / Metrics
- Garbage collection

And limited resources

- Developer Brain
- CPU / Hyperthreading
- Memory
- I / O
- Network
- Disk

History in Java

Been There - Seen That



Java 1.1 - 1997

Green Threads

Java 1.3 - 1998

Native Threads

Java 1.5 - 2004

Thread Pools / Executors

Java 1.7 2011

ForkJoinPool
Parallel Streams

Java 8 2014-2016

CompletableFuture
Rx-Java
Reactive Streams
Reactor

2017-2018 Java 9

Flow
Kotlin Coroutines - Stable

Java 19, 2022

JEP 425 Virtual Threads
Preview

What is?

A Thread

A Thread

- Asynchronous **Unit of Execution**
- Multiple Threads running inside a Process
- One Thread at a time on a CPU
- Inside a thread - synchronous execution
- Switching - Switch/Save Registers / Invalidate Caches ...
- Cheaper than process switching, HW support
- green, platform/native, virtual Threads

Green Threads vs. Native Threads

Green Threads

- User Level Threads
- Simulated Multithreading
- Runs on single (LW)Process / CPU
- Scheduled by VM, not OS
- Lots of HW context switching
- "slow"
- Abandoned in Java 1.3
- Management / State overhead

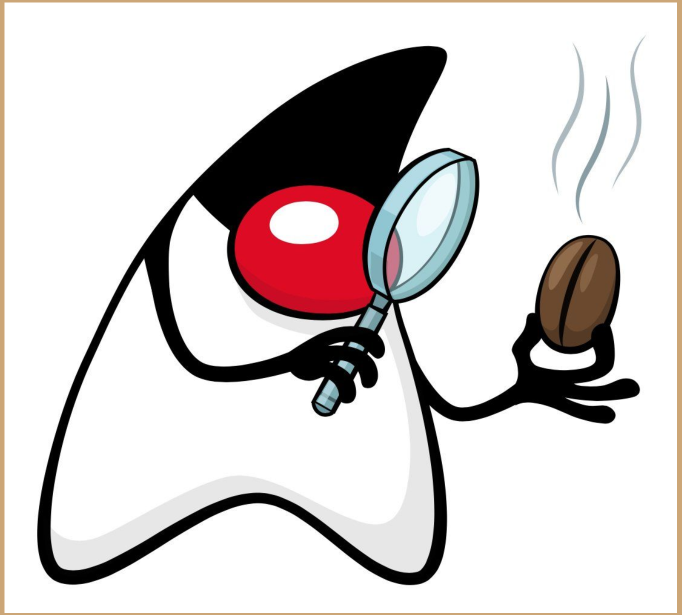
Native Threads

- Mapped to all HW Threads (HT)
- Limited to HW concurrency
- blocking, synchronous execution
- Since Java 1.3 (Solaris - 1.2)
- Better on I/O & context-switching
- Faster than process based concurrency

Java Concurrency looks easier than it is

Devil is in the details

(Brian Goetz - JCP book)



Options Today

Hello Threading World

Example Code & Run

1. new Thread().start()
2. ThreadPool/Executor
3. Reactive Streams/Reactor
4. Kotlin Coroutines
5. Loom - Virtual Threads

Java Threads & Thread Pools

```
var t = new Thread(() -> System.out.println("Hello World"));

t.start();

t.join();

// new! Thread.Builder

var t = Thread.ofPlatform().start(
                  () -> System.out.println("Hello new World"));

t.join();
```

Thread Pools

```
try (var executor = Executors.newFixedThreadPool(5)) {
    IntStream.range(0, 50).forEach(i -> {
        executor.submit(() ->
            System.out.println("Hello Platform Thread "+i+" "+Thread.currentThread())
        );
    });
}  // executor.close()
```

Thread Pools

```
// takes a long time
try (var executor = Executors.newFixedThreadPool(10)) {
    IntStream.range(0, 10_000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));
            return i;
        });
    });
}  // executor.close()
```

Parallel Streams

```
// parallel stream

IntStream.range(1,10).parallel()

 .mapToObj( i -> "Hello World "+i)

   .forEach(System.out::println);
```

Completable Future

```
// Completable Future
var cf = CompletableFuture.completedFuture("complex")
            .thenApplyAsync(String::toUpperCase)
            .thenCombine(
               CompletableFuture.completedFuture("CODE")
               .thenApplyAsync(String::toLowerCase),
                    (s1, s2) -> s1 + s2);
cf.join()
```

Reactive Programming
Reactive Java, Reactor, RxJava, Akka

```
String key = "message";

Mono<String> r = Mono.just("Hello")

    .flatMap(s -> Mono.deferContextual(ctx ->

         Mono.just(s + " " + ctx.get(key))))

    .contextWrite(ctx -> ctx.put(key, "World"));



StepVerifier.create(r).expectNext("Hello World").verifyComplete();
```

Reactor Documentation

Kotlin Coroutines

```
fun main() = runBlocking { // this: CoroutineScope
    launch { doWorld() }
    println("Hello")
}
// suspending function
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

Documentation

Blocking vs. Non-Blocking

Blocking vs. Non-Blocking vs. Continuations

Blocking

- linear program-flow
- execute what you wrote, easy to reason
- blocks on intensive operations
- inefficient use of resources (utilized/blocked)
- classical Java Threading

Non-Blocking

- DSL for describing a processing
- underlying reactive engine
- data flow
- resource efficient
- hard to reason, debug unit-test, profile, maintain
- large complex API
- difficult to correlate operations
- RxJava, Reactive Streams, Akka

Continuation

- ability to capture computation so far and continue later
- explicit continuations (await, async, yield)
- implicit continuations (on entry-points to blocking ops)
- simpler API
- hard work is in the implementation
- Kotlin, JS, Loom

Concurrency & Parallelism help us make more efficient use of existing resources

Concurrency & Parallelism (in Java)

Concurrency vs. Parallelism

Concurrency

- Ability to execute many (different) tasks and make progress on all of them
- Seemingly simultaneously (e.g. on 1 CPU)

Parallel Concurrent Execution

- Multiple tasks are executed concurrently (at the same time) AND
- Multiple CPUs are used to execute tasks in parallel
- **Most common today**

Parallel Execution

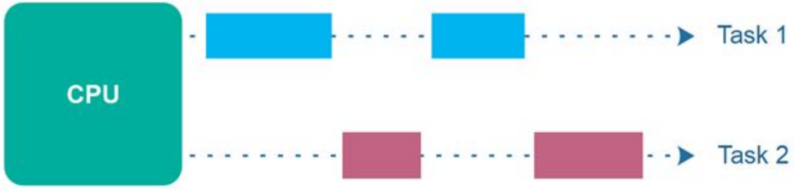
- Utilize more than 1 CPU/Thread to progress multiple tasks simultaneously

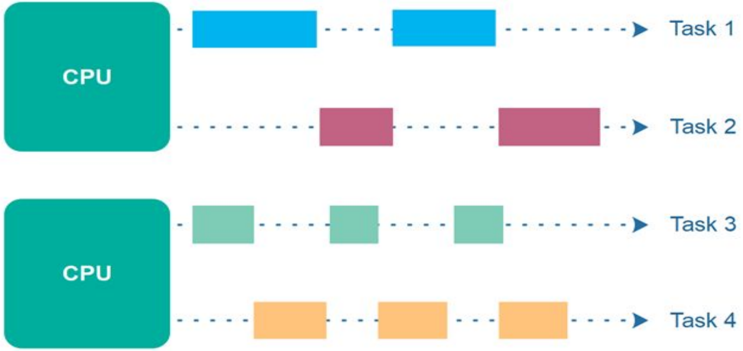
Parallelism

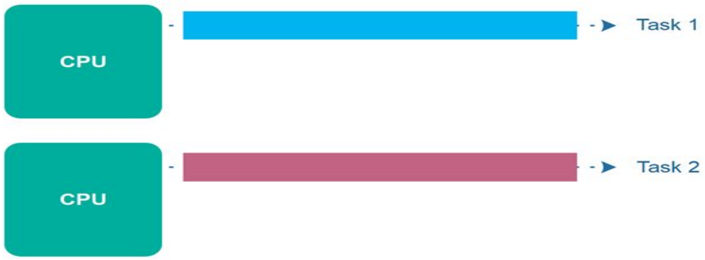
- Ability to divide and conquer a single task into subtasks that can be executed in parallel

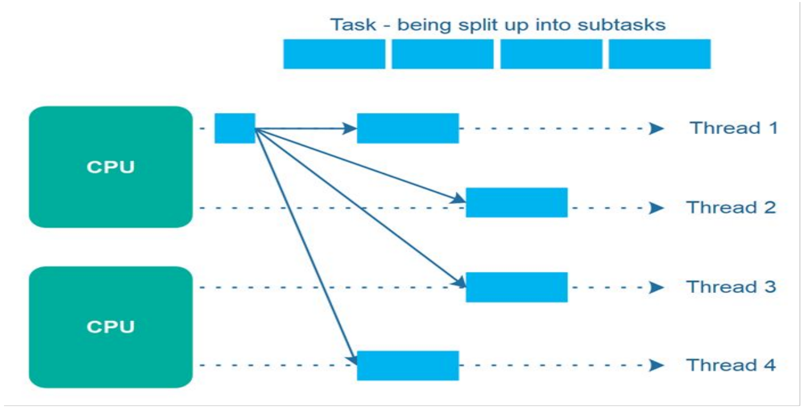
source: jenkov.com

Concurrency vs. Parallelism









Parallel Execution Challenges

- Context Switches / Caches / Branch Predictions
- Race Conditions
- Mutable data / visibility
- Deadlocks
- Starvation
- Resource over- / underutilization
- Immutability / Ownership
- Reasoning / Debugging / Logging
- State-Management
- Execution Depedencies

Project Loom

JEP 425

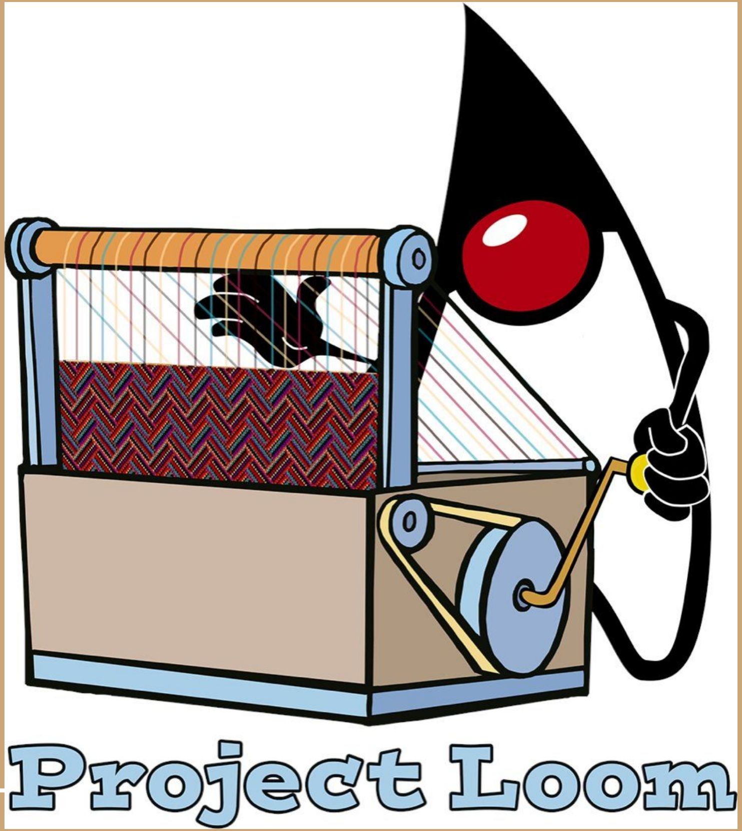
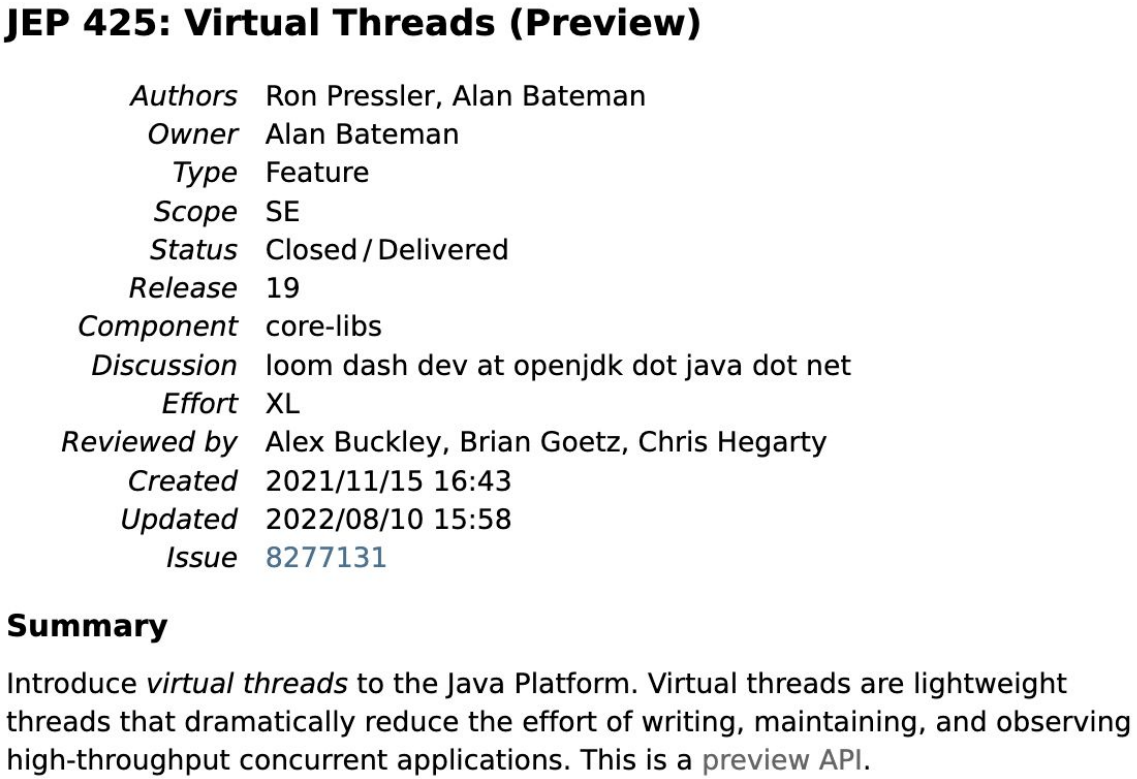


Image OpenJDK / Sharat Chander



JEP 425: Virtual Threads (Preview)

Authors	Ron Pressler, Alan Bateman
Owner	Alan Bateman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	19
Component	core-libs
Discussion	loom dash dev at openjdk dot java dot net
Effort	XL
Reviewed by	Alex Buckley, Brian Goetz, Chris Hegarty
Created	2021/11/15 16:43
Updated	2022/08/10 15:58
Issue	8277131

Summary

Introduce *virtual threads* to the Java Platform. Virtual threads are lightweight threads that dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications. This is a preview API.

JEP 425 - Project Loom - openjdk.org/jeps/425

JEP 425 - Goals

Goals

- Enable server applications written in the **simple thread-per-request** style to **scale with near-optimal** hardware utilization.
- Enable existing code that uses the `java.lang.Thread` API to **adopt virtual threads with minimal change**.
- Enable **easy troubleshooting, debugging, and profiling** of virtual threads with existing JDK tools.

Non-Goals

- It is not a goal to remove the traditional implementation of threads, or to ~~**silently migrate existing applications**~~ to use virtual threads.
- It is not a goal to ~~**change the basic concurrency model**~~ of Java.
- It is not a goal to offer a ~~**new data parallelism construct**~~ in either the Java language or the Java libraries. The Stream API remains the preferred way to process large data sets in parallel.

What’s in the cup?

What’s in the ~~box~~cup?

- Continuations internally in the JVM
- Reimplementation of Networking / IO Code in JVM
- lightweight Virtual Thread, same API as java.util.Thread
- Thread.Builder
- VirtualThreadExecutors
- Auto-Closeable Excecutors
- Structured Concurrency (StructuredTaskScope)

What is?

A virtual Thread

Virtual Thread

- same, stable API as traditional thread (deprecations will be removed)
- handled differently during blocking operations
- lightweight (300 bytes) like a Runnable, JVM can execute millions
- temporarily bound to a platform (carrier) thread
- on each blocking/parking operation -> Continuation yielding
- stack is copied to heap
- on resume, stack copied back and
- execution resumed on different carrier Thread
- uses a separate ForkJoinPool (FJP), to also prevent starving

  ```
  -Djdk.defaultScheduler.parallelism=N
  ```

Java Virtual Threads

```
var threads =

IntStream.range(0,10).mapToObj(i ->
     Thread.ofVirtual().start(() -> { // or Thread.startVirtualThread(runnable)

          System.out.println("Hello Virtual Thread "+i+" "+Thread.currentThread());

     })).toList();

for (Thread t : threads) {

  t.join();

}
```

```
Hello Virtual Thread 3 VirtualThread[#10079]/runnable@ForkJoinPool-1-worker-18
Hello Virtual Thread 6 VirtualThread[#10082]/runnable@ForkJoinPool-1-worker-17
Hello Virtual Thread 1 VirtualThread[#10077]/runnable@ForkJoinPool-1-worker-16
Hello Virtual Thread 5 VirtualThread[#10081]/runnable@ForkJoinPool-1-worker-15
Hello Virtual Thread 7 VirtualThread[#10083]/runnable@ForkJoinPool-1-worker-17
Hello Virtual Thread 2 VirtualThread[#10078]/runnable@ForkJoinPool-1-worker-17
Hello Virtual Thread 4 VirtualThread[#10080]/runnable@ForkJoinPool-1-worker-17
Hello Virtual Thread 8 VirtualThread[#10084]/runnable@ForkJoinPool-1-worker-19
Hello Virtual Thread 9 VirtualThread[#10085]/runnable@ForkJoinPool-1-worker-18
```

Java Virtual Threads - Executor

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {

    IntStream.range(0, 1_000_000).forEach(i -> {

        executor.submit(() -> {

            Thread.sleep(java.time.Duration.ofSeconds(1));

            return i; // callable, throws Exception

        });

    });

}  // executor.close() is called implicitly, and waits
```

Under the Hood

If in doubt

Start a new virtual Thread

Example: Echo Web-Server

```
// java --enable-preview --source 19 LoomServer.java
// echo -n 'Hello Loom' | nc -n 127.0.0.1 2000
import java.io.*;
import java.net.*;
import java.util.concurrent.*;

public class LoomServer {
    public static void main(String...args) throws IOException {
        try (var ss = new ServerSocket(2000);
             var pool = Executors.newVirtualThreadPerTaskExecutor()) {
             while (true) {
                var socket = ss.accept();
                pool.execute(() -> {
                    try (var s = socket;
                         var in = s.getInputStream();
                         var out = s.getOutputStream()) {
                            byte b = -1;
                            while ((b = (byte)in.read()) != -1) {
                                out.write(b+1);
                            }
                        } catch(IOException ioe) {}
                });
             }
```

Under The Hood

- Virtual Threads run on (different) Platform Threads
- They use a separate Fork Join Pool
- Instead of blocking (IO, networking, sleep, synchronization) they are yielding control
- Blocking Code in the JVM refactored to use Continuations
- Continuations move stack from Platform Thread to Heap
- Can pick up later on another thread
- Except when using addresses or native code

Example Thread.sleep()

- Thread.sleep()
  - VThread.sleepNanos()
    - VThread.doSleepNanos()
      - VTread.tryYield()
        - VThread.yieldContinuation()
          - unmount()
          - Continuation.yield()

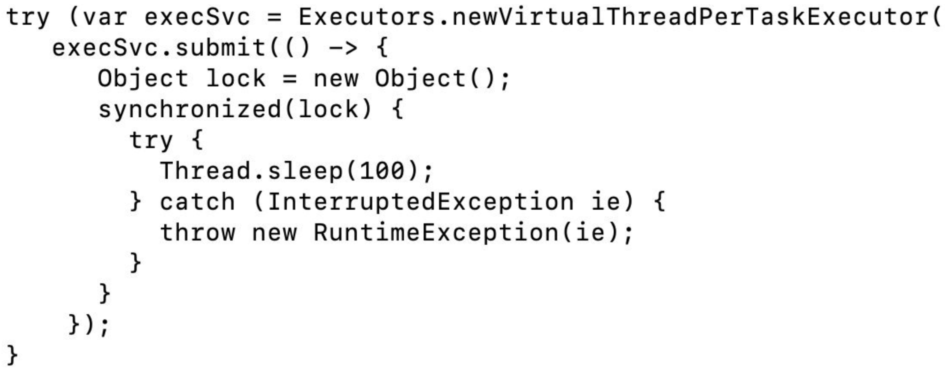
```
private void unmount() {
    // set Thread.currentThread() to return the platform thread
    Thread carrier = this.carrierThread;
    carrier.setCurrentThread(carrier);
    // break connection to carrier thread, synchronized with interrupt
    synchronized (interruptLock) {
        setCarrierThread(null);
    }
    carrier.clearInterrupt();
}
```

```
static final ContinuationScope VTHREAD_SCOPE =
            new ContinuationScope("VirtualThreads");

@ChangesCurrentThread
private boolean yieldContinuation() {
    boolean notifyJvmti = notifyJvmtiEvents;


    // unmount
    if (notifyJvmti) notifyJvmtiUnmountBegin(false);
    unmount();
    try {
        return Continuation.yield(VTHREAD_SCOPE);
    } finally {
        // re-mount
        mount();
        if (notifyJvmti) notifyJvmtiMountEnd(false);
    }
}
```

Caveats or when does it not work?

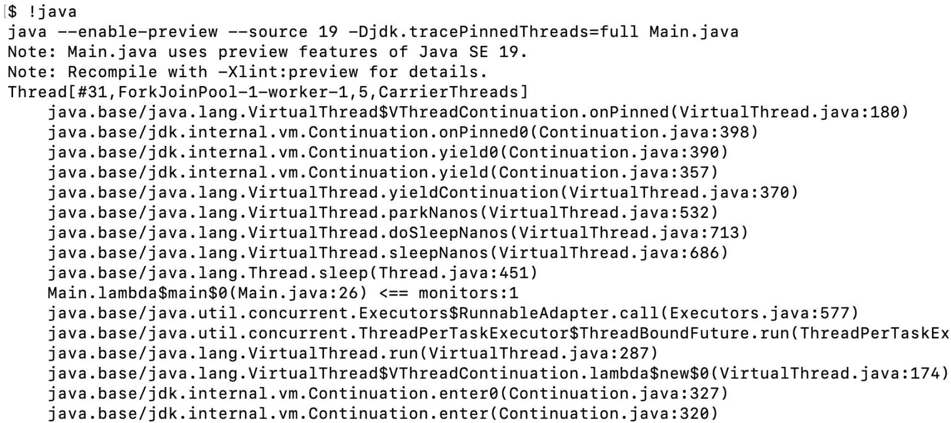
- when stack cannot be moved to heap
- if it contains memory addresses (synchronized) -> use ReentrantLock!
- calls c-code
- File I/O
- DNS (Windows)

- then the task stays pinned to a platform thread
- to avoic exhaustion the FJP creates new temporary platform threads
- can track with `-Djdk.tracePinnedThreads=full`



```
try (var execSvc = Executors.newVirtualThreadPerTaskExecutor()) {
   execSvc.submit(() -> {
      Object lock = new Object();
      synchronized(lock) {
        try {
          Thread.sleep(100);
        } catch (InterruptedException ie) {
          throw new RuntimeException(ie);
        }
      }
    });
}
```

example by **A. Sundararajan**



```
$ !java
java --enable-preview --source 19 -Djdk.tracePinnedThreads=full Main.java
Note: Main.java uses preview features of Java SE 19.
Note: Recompile with -Xlint:preview for details.
Thread[#31,ForkJoinPool-1-worker-1,5,CarrierThreads]
    java.base/java.lang.VirtualThread$VThreadContinuation.onPinned(VirtualThread.java:180)
    java.base/jdk.internal.vm.Continuation.onPinned0(Continuation.java:398)
    java.base/jdk.internal.vm.Continuation.yield0(Continuation.java:390)
    java.base/jdk.internal.vm.Continuation.yield(Continuation.java:357)
    java.base/java.lang.VirtualThread.yieldContinuation(VirtualThread.java:370)
    java.base/java.lang.VirtualThread.parkNanos(VirtualThread.java:532)
    java.base/java.lang.VirtualThread.doSleepNanos(VirtualThread.java:713)
    java.base/java.lang.VirtualThread.sleepNanos(VirtualThread.java:686)
    java.base/java.lang.Thread.sleep(Thread.java:451)
    Main.lambda$main$0(Main.java:26) <== monitors:1
    java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:577)
    java.base/java.util.concurrent.ThreadPerTaskExecutor$ThreadBoundFuture.run(ThreadPerTaskEx
    java.base/java.lang.VirtualThread.run(VirtualThread.java:287)
    java.base/java.lang.VirtualThread$VThreadContinuation.lambda$new$0(VirtualThread.java:174)
    java.base/jdk.internal.vm.Continuation.enter0(Continuation.java:327)
    java.base/jdk.internal.vm.Continuation.enter(Continuation.java:320)
```

JDK Changes

- Large pull request (#8166) touching 1333 files
- Thread.Builder, virtualThreadPerTaskExecutor
- refactor all blocking/parking code to use Continuations for Virtual Threads
  - Network I/O,
  - Locks
  - Thread.sleep
- replace c-code where possible with Java code (e.g. in Method -> MethodHandles)
- not (yet):
  - File I/O (waiting for io_uring)
  - synchronized, due to address usage

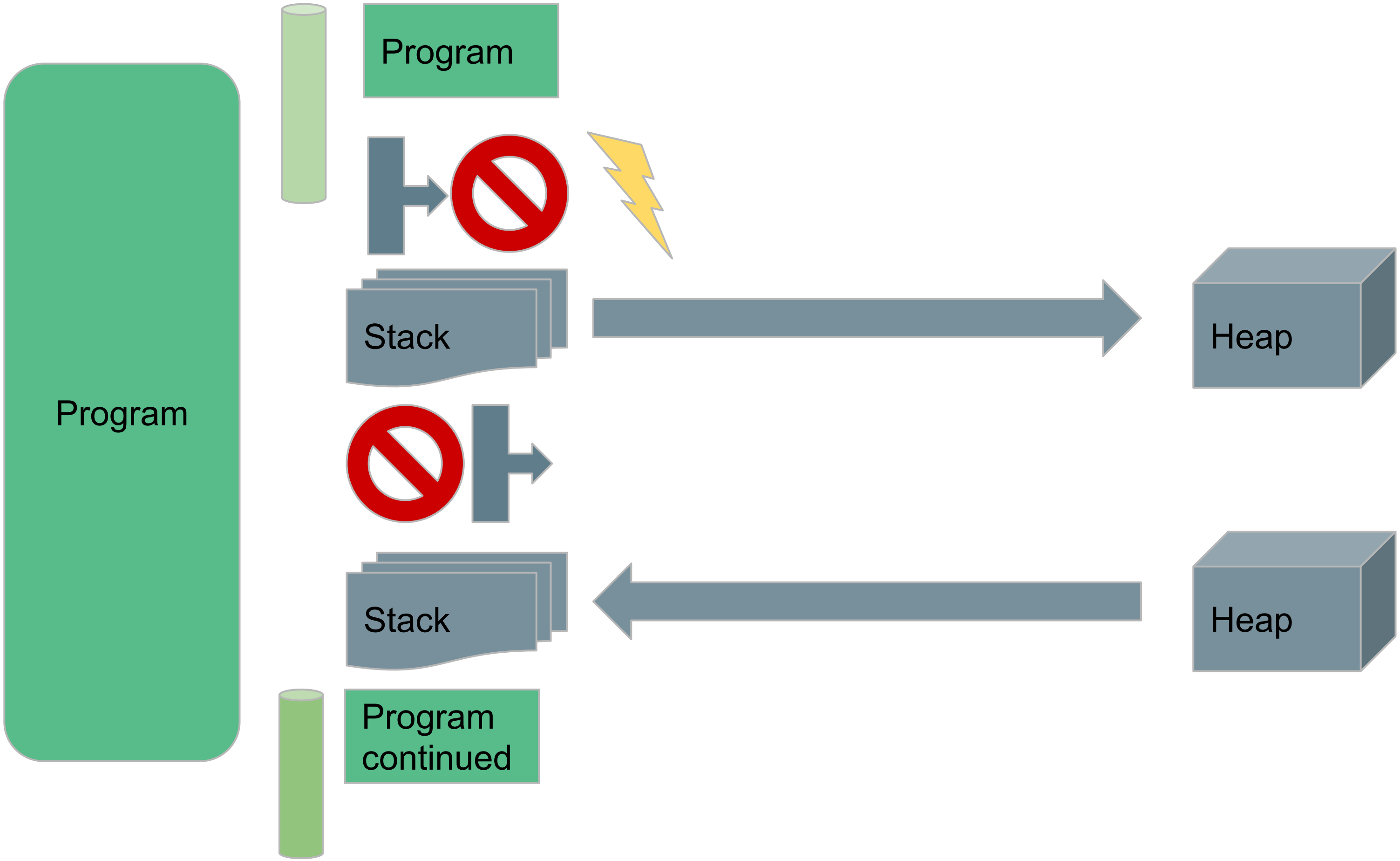
Continuations on the long way to Loom

What is?

A Continuation?

A continuation

"A continuation is a callback function k that represents the current state of the program's execution. More precisely, the continuation k is a function of one argument, namely the value that has been computed so far, that returns the final value of the computation after the rest of the program has run to completion."



Continuation in Pictures

```
import jdk.internal.vm.Continuation;
import jdk.internal.vm.ContinuationScope;

var scope = new ContinuationScope("scope");

var c = new Continuation(scope, () -> {
    System.out.println("Started");
    Continuation.yield(scope);
    System.out.println("Running");
    Continuation.yield(scope);
    System.out.println("Still running");
});

System.out.println("Start");
int i=0;
while (!c.isDone()) {
    c.run();
    System.out.println("Running "+i+" result "+c.isDone());
    i++;
}
System.out.println("End");
```

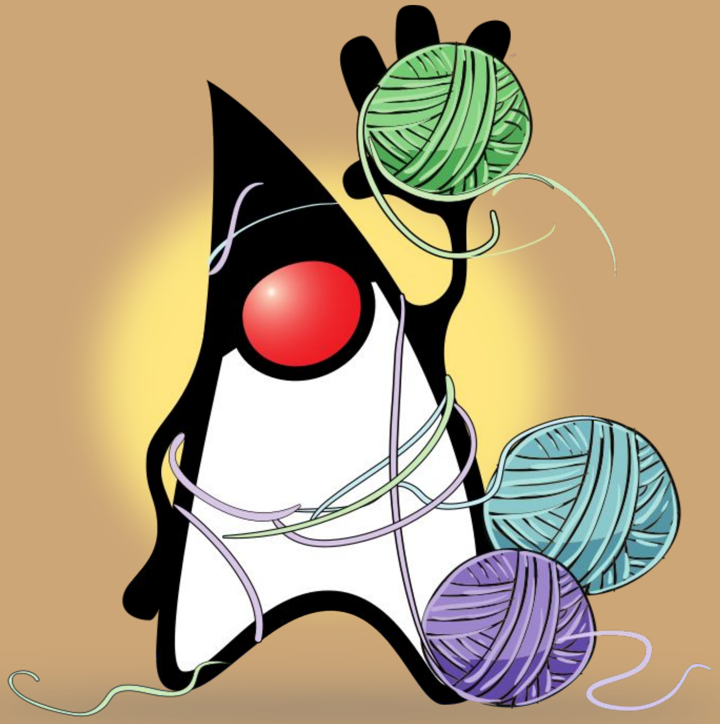
```
Start
Started
Running 0 result false
Running
Running 1 result false
Still running
Running 2 result true
End
```

Continuations in the JDK

What is?

Structured Concurrency

JEP 428



Launching Millions of Threads is no better than GOTO

Nathaniel Smith

Structured Concurrency (JEP 428)

Goals

- Improve the maintainability, reliability, and observability of multithreaded code.
- Promote a style of concurrent programming which can eliminate common risks arising from cancellation and shutdown, such as thread leaks and cancellation delays.

Non-Goals

- It is not a goal to replace any of the concurrency constructs in the `java.util.concurrent` package, such as `ExecutorService` and `Future`.
- It is not a goal to define the definitive structured concurrency API for Java. Other structured concurrency constructs can be defined by third-party libraries or in future JDK releases.
- It is not a goal to define a means of sharing streams of data among threads (i.e., *channels*). We might propose to do so in the future.
- It is not a goal to replace the existing thread interruption mechanism with a new thread cancellation mechanism. We might propose to do so in the future.

Structured Concurrency (JEP 428) - TODO

- with so many threads running you need management and control structures
- since JDK 5 no direct interaction with Threads but submit to ExecutorService -> Future
- ExecutorService is still unstructured
- Multi-Thread Control structures are missing in Java (like Erlang/Akka)
    - even if they exist in the business process
    - no task->subtask relationships between threads
    - every thread can read from a future or submit to an executor
- Loom keeps this model -> Structure is missing
- what happens when a (child or parent) thread fails?

Scope - virtual thread launcher

- specialized, auto-closeable, short-lived execution-scope
- like an executor, but uses virtual Threads and FJP
- submit tasks to it (**`fork()`**)
- `StructuredTaskScope<T>()`
- `Future<T> future = scope.fork(task);`
- `scope.join()` -> returns when all tasks are complete
- switch on `future.state() (FAILED, RUNNING, SUCCESS, CANCELLED)`
    - `future.resultNow() / future.exceptionNow()`
- better with specialized implementations (first-one-wins or fail-fast)

StructuredTaskScope Example

```
try ( var scope = new StructuredTaskScope<String>() ) {
    var future1 = scope.fork(task1);
    var future2 = scope.fork(task2);
    scope.join();
    return switch (future1.state()) {
        case Future.SUCCESS -> future1.resultNow();
        case Future.FAILED -> future1.exceptionNow();
    }
}
```

API StructuredTaskScope

StructuredTaskScope.ShutdownOnSuccess

```
import jdk.incubator.concurrent.*;
try ( var scope = new StructuredTaskScope.ShutdownOnSuccess<String>() ) {
    IntStream.range(0,10).forEach(i ->
                                  scope.fork(() -> String.valueOf(i)));
    scope.join();
    // first returning wins, exception if none did
    System.out.println(scope.result());
}
```

- ShutdownOnFailure - same just for fail-fast

API StructuredTaskScope

Your own Scope

- to implement your own "**structured business logic**"
- Subclass `StructuredTaskScope`
- override `handleComplete(Future<T>)`
- depending on future-state, do what you need to do
- is called concurrently, needs to use thread safe instance state
- custom result method, reduce state to a result (or Exception)
- **Concern**: Still a lot of multi-threaded technical infrastructure complexity

JEP Café #13

Loom Debugging - Just regular!?

- Stacktrace in IDE-Debuggers (as expected)
- Patches in Java Debug Wire Protocol (JWDP) & Java Debugger Interface (JDI)
- Challenge - Display Millions of Threads
- Structured concurrency can help here too (Tree-Display)
- JFR should work - match allocations, method calls, etc. to virtual Threads
- Gaps in Thread API:
  - list all running threads
  - carrier <-> virtual thread

wiki.openjdk.org/display/loom/Debugger+Support

Loom Demos & Comparisions

github.com/ebarlas

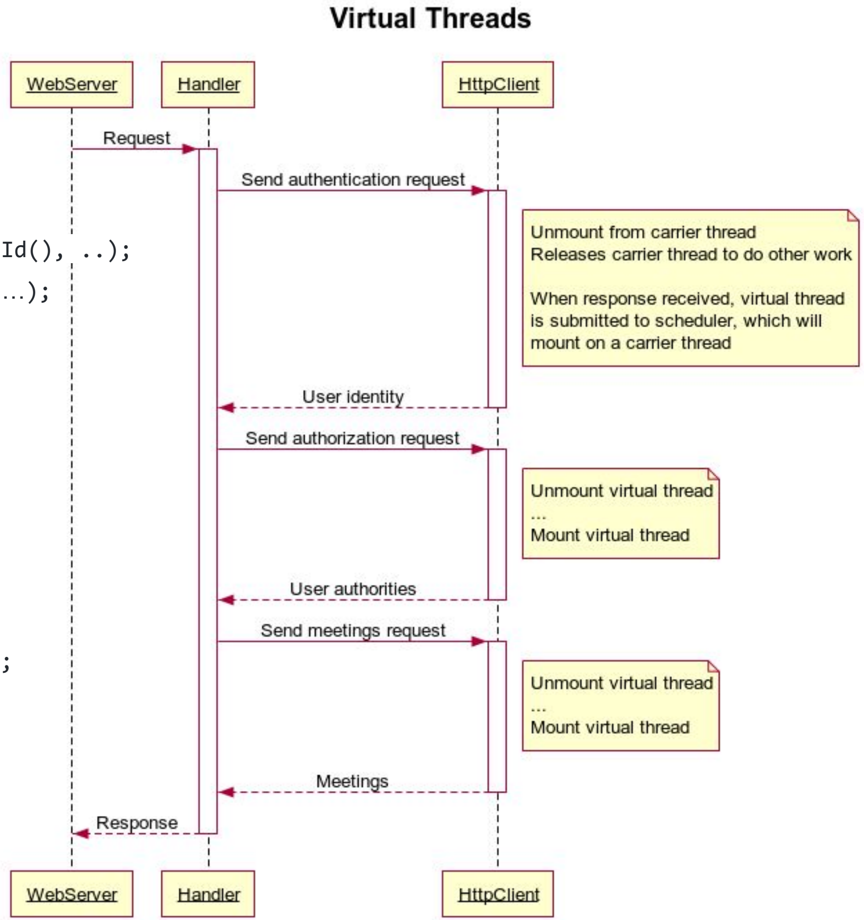
Web-Backend
Game of Life
5M persistent connections

Elliot Barlas - MicroHttp - LogMeIn

Performance Comparison
Web-Server calling Backend

github.com/ebarlas/project-loom-comparison

```
public void handle(Request request, Consumer<Response> callback) {
    executorService.execute(() -> callback.accept(doHandle(request)));
}
Response doHandle(Request request) {
  var token = request.header("Authorization");
  var authentication = sendRequestFor("/authenticate?token=" + token, …);
  var authorization = sendRequestFor("/authorize?id=" + authentication.userId(), ..);
  var meetings = sendRequestFor("/meetings?id=" + authentication.userId(), …);
  var headers = List.of(new Header("Content-Type", "application/json"));
  return new Response(200, "OK", headers, Json.toJson(meetings));
}
<T> T sendRequestFor(String endpoint, Class<T> type)
    throws IOException, InterruptedException {
    URI uri = URI.create("http://%s%s".formatted(backend, endpoint));
    var request = HttpRequest.newBuilder().uri(uri).GET().build();
    HttpResponse<String> response = httpClient.send(request,
                                      HttpResponse.BodyHandlers.ofString());
    if (response.statusCode() != 200) {
        throw new RuntimeException("error occurred contacting "+endpoint);
    }
    return Json.fromJson(response.body(), type);
}
```

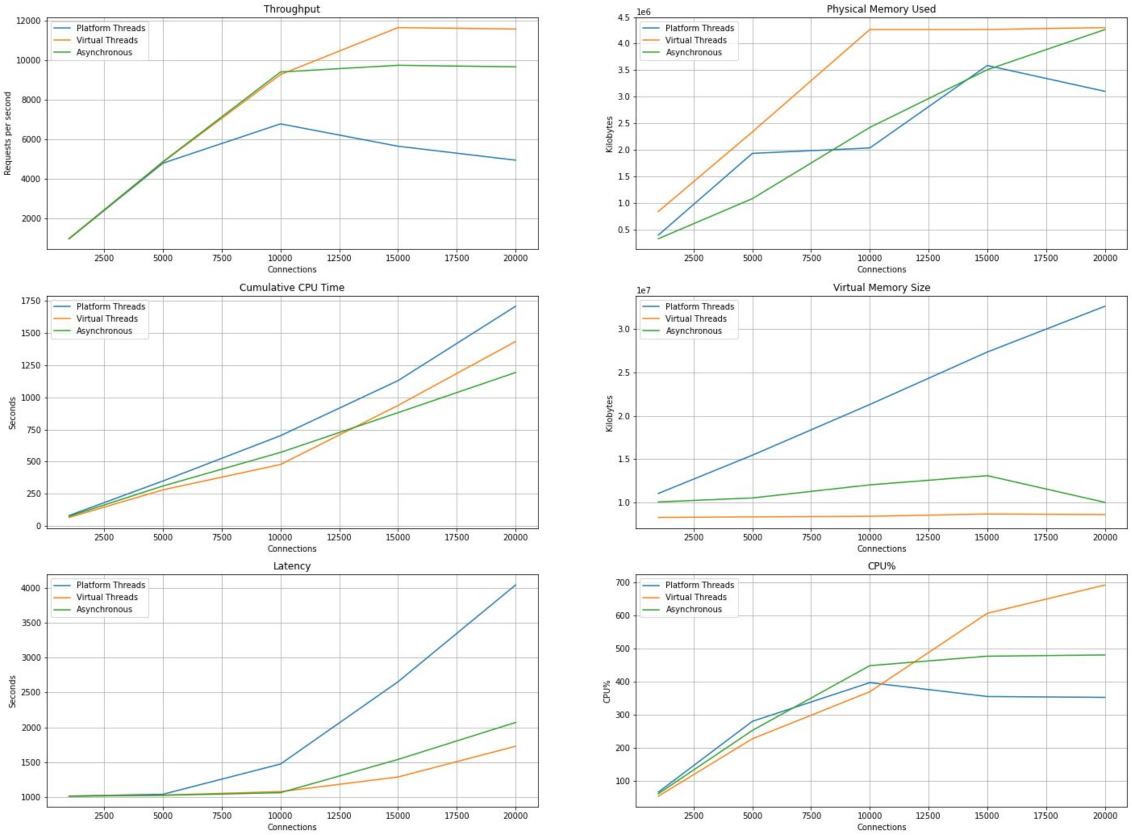


www.websequencediagrams.com

Same old Executor (for both Platform & Virtual Threads)

WebServer

- Browser (ab)
- Web-Server calling
- 3x Backend (0,3s latency)
    - Authentication
    - Authorization
    - Database
- Platform Threads
- Asynchronous
- Virtual Threads



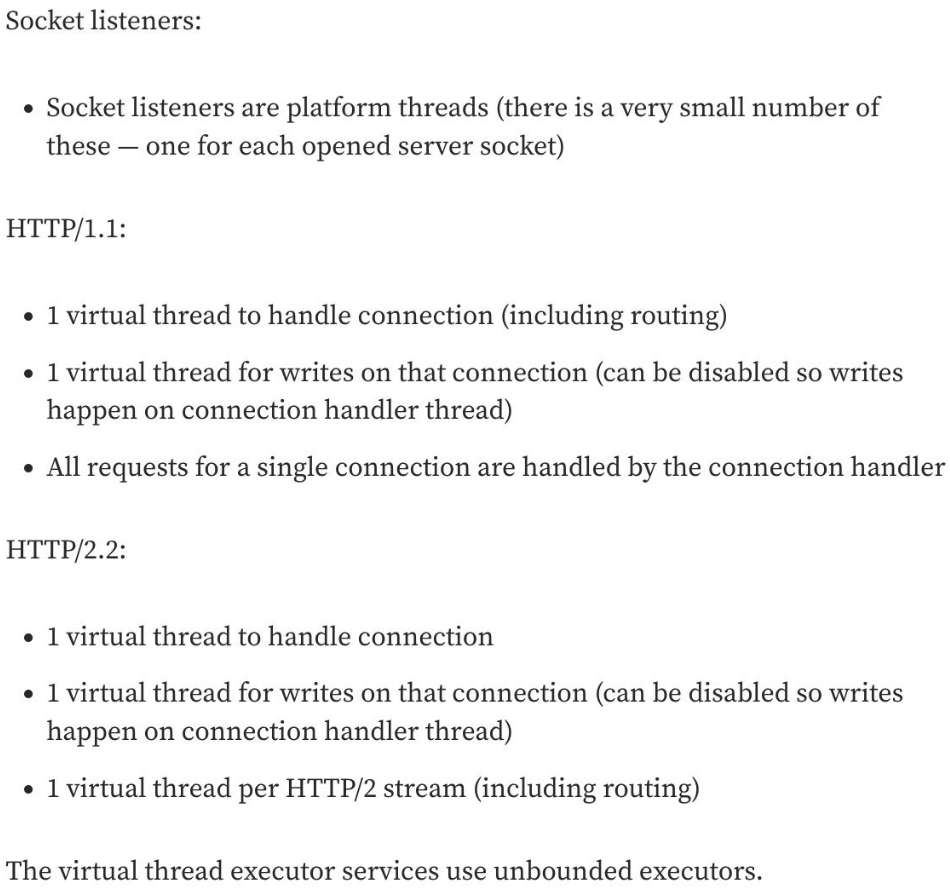
HELIDON NÍMA

Nima Test

```
java --enable-preview -jar nima/target/example-nima-blocking.jar

2022.09.22 02:36:05.204 Logging at initialization configured using classpath: /logging.properties
2022.09.22 02:36:05.394 [0x6e82e640] http://127.0.0.1:8080 bound for socket '@default'
2022.09.22 02:36:05.396 [0x6e82e640] async writes, queue length: 32
2022.09.22 02:36:05.403 Níma server started all channels in 8 milliseconds. 244 milliseconds since JVM startup. Java 19+36-2238

ab -n 10000 -c 8 http://127.0.0.1:8080/one
Concurrency Level:      8
Time taken for tests:   0.848 seconds
Complete requests:      10000
Requests per second:    11797.14 [#/sec] (mean)
Time per request:       0.678 [ms] (mean)
Time per request:       0.085 [ms] (mean, across all concurrent requests)
Transfer rate:          0.00 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:     0    0   0.0      0       0
Waiting:        0    0   0.0      0       0
Total:          0    0   0.0      0       1
```

Helidon Nima

Note: In either case, you should not "obstruct" the thread. Obstruction is a long-term, full utilization of the thread. In a reactive framework this would consume one of the event loop threads effectively stopping the server. In blocking (Níma) this may cause an issue with the "pinned thread". In both cases this can be resolved by off-loading the heavy load to a dedicated executor service using platform threads.



Socket listeners:

- Socket listeners are platform threads (there is a very small number of these — one for each opened server socket)

HTTP/1.1:

- 1 virtual thread to handle connection (including routing)
- 1 virtual thread for writes on that connection (can be disabled so writes happen on connection handler thread)
- All requests for a single connection are handled by the connection handler

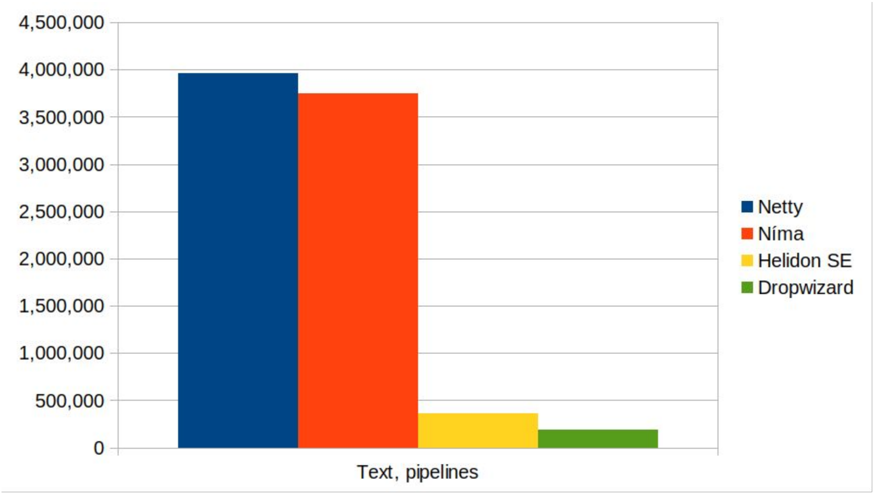
HTTP/2.2:

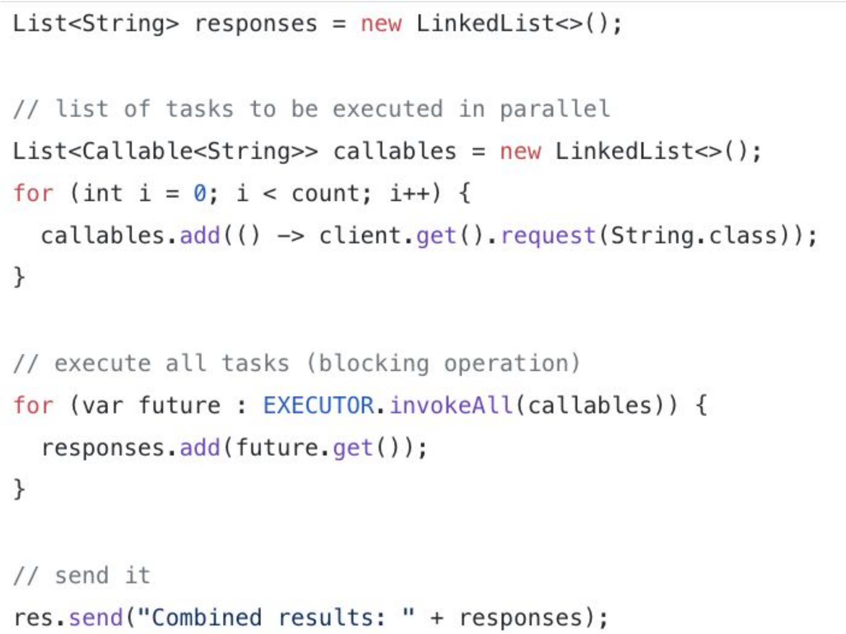
- 1 virtual thread to handle connection
- 1 virtual thread for writes on that connection (can be disabled so writes happen on connection handler thread)
- 1 virtual thread per HTTP/2 stream (including routing)

The virtual thread executor services use unbounded executors.

Performance comparable to Netty pipelined

Note: What we can see from these numbers (and what is our goal with Níma) is that we can achieve performance comparable to a minimalist Netty server, while maintaining a simple, easy to use programming model.





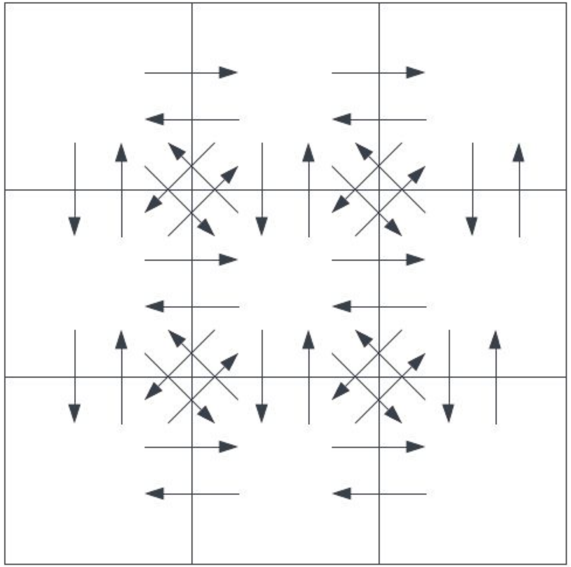
```
List<String> responses = new LinkedList<>();

// list of tasks to be executed in parallel
List<Callable<String>> callables = new LinkedList<>();
for (int i = 0; i < count; i++) {
  callables.add(() -> client.get().request(String.class));
}

// execute all tasks (blocking operation)
for (var future : EXECUTOR.invokeAll(callables)) {
  responses.add(future.get());
}

// send it
res.send("Combined results: " + responses);
```

Game of Life
CSP - Communicating Sequential Processes

github.com/ebarlas/game-of-life-csp

CSP - Not possible in Java before

- Channel (BlockingQueue<Boolean>) to exchange information
- Grid - Channel<boolean[][]>
- Each Cell has
    - **a virtual thread**
    - channels for ticks and results (width x height x 2)
    - one channel per neighbour (~ width x height x 8)
    - aka a LOT of channels/queues

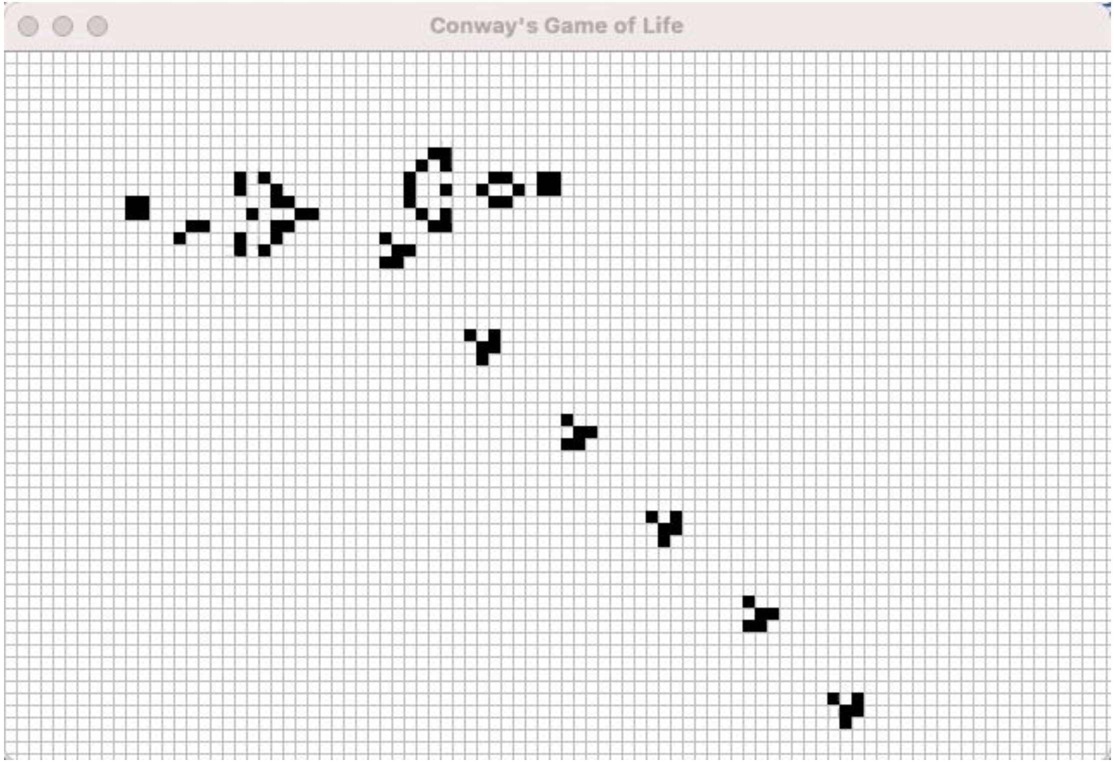


Wikipedia CSP

Cell's biological Clock - "Life"

```
private void run() {
    while (true) {
        tickChannel.take(); // wait for tick stimulus

        // announce liveness to neighbors
        outChannels.forEach(ch -> ch.put(alive));

        // receive liveness from neighbors
        int neighbors = inChannels.stream()
            .map(Channel::take)
            .mapToInt(b -> b ? 1 : 0).sum();

        // calculate next state based on game of life rules
        alive = alive && neighbors == 2 || neighbors == 3;

        // announce resulting next state
        resultChannel.put(alive);
    }
}
```

- `take / put` -> block -> suspend
- exchange via *`channel/queue`*
- write only to local **`state`**



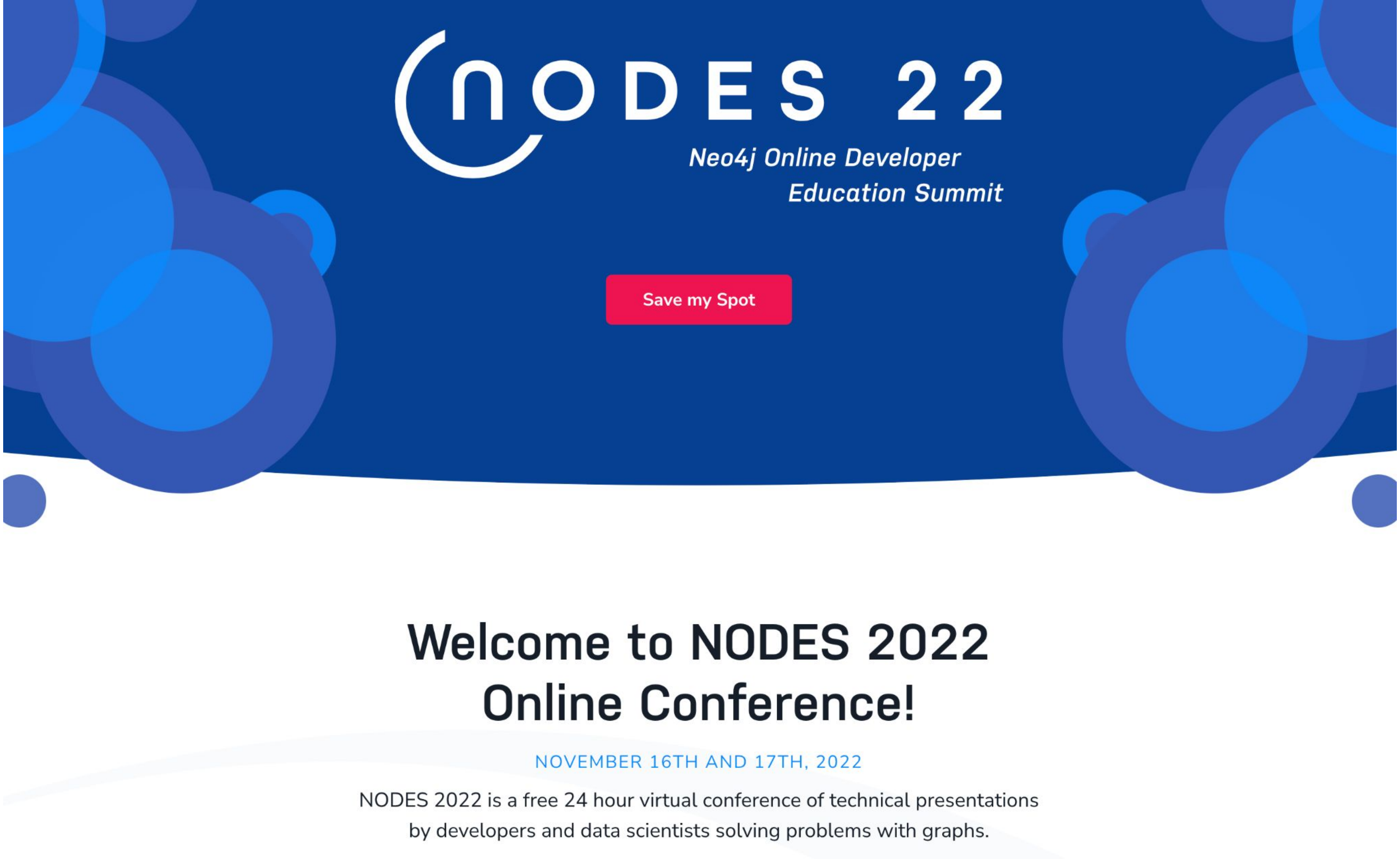
Conway's Game of Life

Resources

(there is a lot)

Resources

- JEP 425 - Virtual Threads
- JEP 428 - Structured Concurrency
- JEP Café #12 - 10M Threads
- JEP Café #13 - Loom Tutorial
- Heinz Kabutz Loom Video
- JavaSpektrum Loom (me)
- Loom Lab (Nicolai Parlog)
- Million Virtual Threads
- News Grab Bag Java 19
- Virtual Threads PR
- Loom Wiki
- Inside Java Loom
- Loom Networking under the Hood
- InfoQ Interview Ron Pressler
- State Of Loom Part 1
- State Of Loom Part 2
- Helidon Nima
- Loom and Thread Fairness (Morling)
- Loom Comparison
- Structured Concurrency



NODES 22

Neo4j Online Developer Education Summit

Welcome to NODES 2022 Online Conference!

NOVEMBER 16TH AND 17TH, 2022

NODES 2022 is a free 24 hour virtual conference of technical presentations by developers and data scientists solving problems with graphs.

What can you do?

Test! Provide feedback!

Don’t use in production!

Thank You!

... I'd love to take questions!