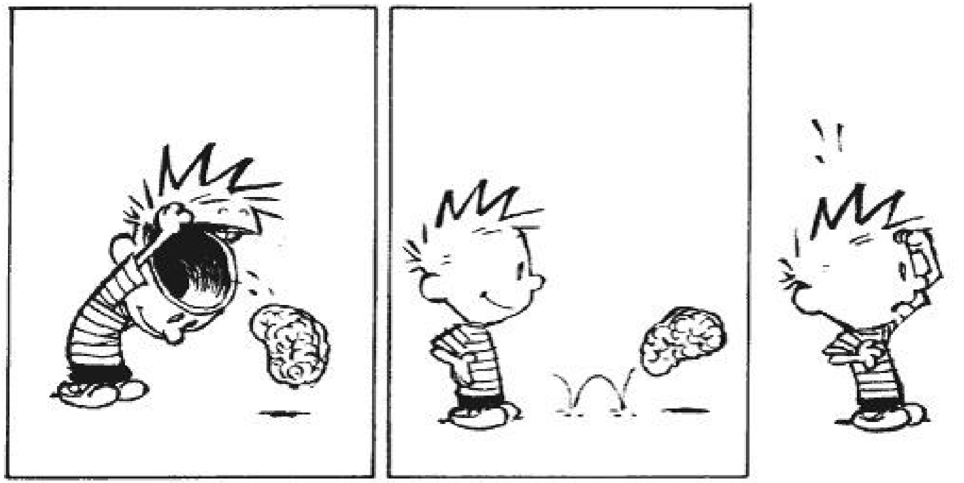
Loom is looming

Rémi Forax / José Paumard



CALVIN & HOBBES © BIL WATTERSON

Don’t believe what we are saying !

What is Loom ?

OpenJDK project started late 2017 by Ron Pressler

Goal: Lowering the cost of concurency

should be integrated soon as preview feature

How many threads can I run ?

DEMO

How many threads can I run ?

```
var barrier = new CyclicBarrier(1_000_001);
for(var i = 0; i < 1_000_000; i++) {
  System.out.println(i);
  new Thread(() -> {
    try {
      barrier.await();
    } catch(InterruptedException | BrokenBarrierException e) {
      throw new AssertionError(e);
    }
  }).start();
}
barrier.await();
```

How many threads can I run ?

On a MacBook Air M1 (16G of RAM)

```
...
4063
4064
4065
4066
[0.373s][warning][os,thread] Failed to start thread "Unknown thread" -
pthread_create failed (EAGAIN) for attributes: stacksize: 2048k,
guardsize: 16k, detached.
[0.373s][warning][os,thread] Failed to start the native thread for
java.lang.Thread "Thread-4066"
Exception in thread "main" java.lang.OutOfMemoryError: unable to create
native thread: possibly out of memory or process/resource limits reached
        at java.base/java.lang.Thread.start0(Native Method)
        at java.base/java.lang.Thread.start(Thread.java:1451)
        at _3_how_many_platform_thread.printHowManyThreads(...java:19)
        at _3_how_many_platform_thread.main(...java:46)
```

What if I’ve more than 4066 clients for my web server ?

Need to change the model

1 request <==> 1 threads

Paradigmatic change
Asynchronous programming

But I loose the stack trace
=> debugging is harder
=> profiling is harder
=> testing is harder
+
colored function problem

Asynchronous Programming
is like an addiction

Once you start using it
It’s hard to go back
… and you loose all your friendly libraries

Solution = coroutine

(not like Kotlin like golang)

In Java,

virtual threads

DEMO

Virtual thread

```
// platform threads
var pthread = new Thread(() -> {
  System.out.println("platform " + Thread.currentThread());
});
pthread.start();
pthread.join();

// virtual threads
var vthread = Thread.startVirtualThread(() -> {
  System.out.println("virtual " + Thread.currentThread());
});
vthread.join();
```

Virtual thread

```
// platform threads
platform Thread[#14,Thread-0,5,main]

// virtual threads
virtual VirtualThread[#15]/runnable@ForkJoinPool-1-worker-1
```

Use a dedicated thread pool internally

Warning! This pool is not the common fork join pool

Or using a *polymorphic* builder

Thread builder

```
// platform threads
var pthread = Thread.ofPlatform()
    .name("platform-", 0)
    .start(() -> {
      System.out.println("platform " + Thread.currentThread());
    });
pthread.join();

// virtual thread
var vthread = Thread.ofVirtual()
    .name("virtual-", 0)
    .start(() -> {
      System.out.println("virtual " + Thread.currentThread());
    });
vthread.join();
```

How many **virtual** threads can I run ?

DEMO

How many virtual threads can I run ?

```
var counter = new AtomicInteger();
var threads = IntStream.range(0, 1_000_000)
    .mapToObj(i -> Thread.ofVirtual().unstarted(() -> {
      try {
        Thread.sleep(1_000);
      } catch (InterruptedException e) {
        throw new AssertionError(e);
      }
      counter.incrementAndGet();
    }))
    .toList();

for (var thread : threads) { thread.start(); }
for (var thread : threads) { thread.join(); }
System.out.println(counter);  // 1_000_000
```

Running a thread

Platform (native) thread (starts in ms)

- Creates a 2M stack
- System call to ask the OS to schedule the thread

Virtual thread (starts in µs)

- Growing stack using stack banging
- Use a specific fork-join pool of pre-created OS threads
  - One OS thread per core

Concurrency for Loom

Two strategies for concurrency

- Competitive: all threads compete for the CPUs/cores
- Cooperative: each thread hand of the CPUs to the next

Loom does both, carrier threads compete and virtual threads cooperate

How it works under the hood ?

Uses jdk.internal.vm.Continuation

```
var continuation = new Continuation(() -> {
  System.out.println("C1");
  Continuation.yield();
  System.out.println("C2");
  Continuation.yield();
  System.out.println("C3");
});
```

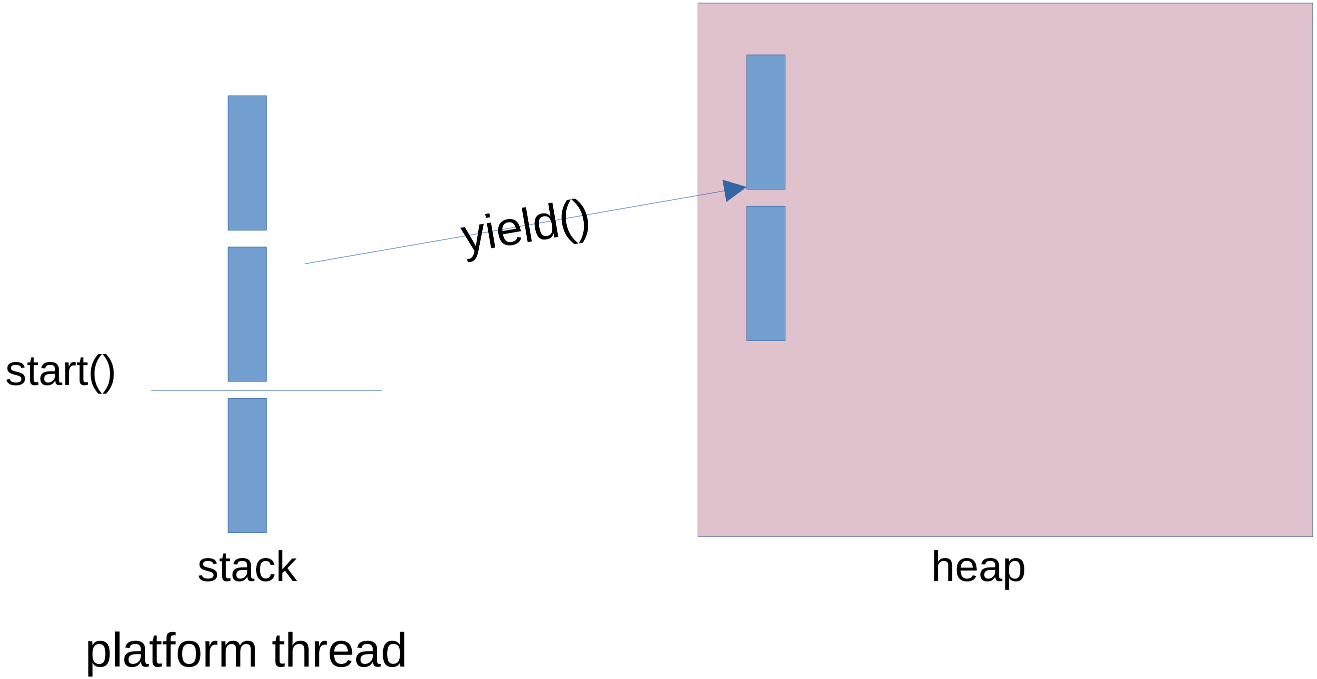
```
System.out.println("start");
continuation.start();
System.out.println("came back");
continuation.start();
System.out.println("back again");
continuation.start();
System.out.println("back again again");
```

Execution:

```
start
C1
came back
C2
back again
C3
back again again
```

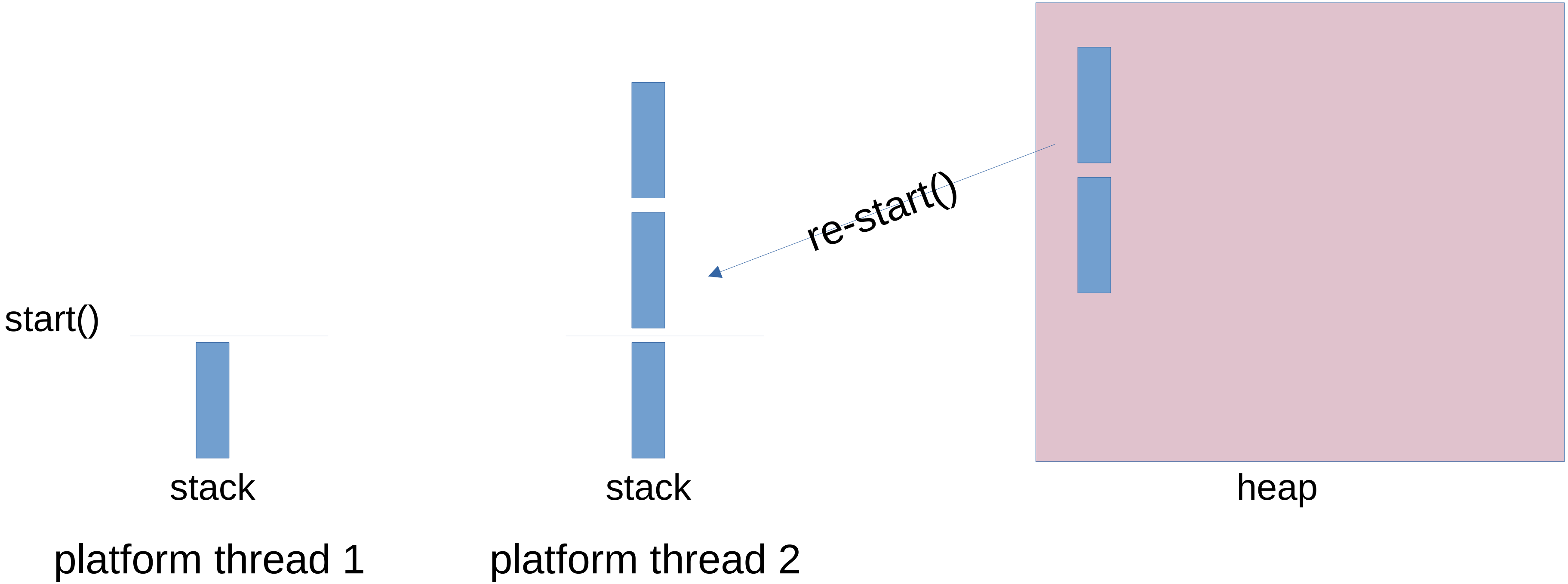
Continuation.yield()

yield() copy the stack to the heap



Continuation.start()

start() copy from the heap to another stack



And in the JDK

All blocking codes are changed to

- Check if current thread is a virtual thread
- If it is, Continuation.yield() instead of blocking
- Register a handler that will be called when the OS is ready
  - When the handler is called, find a carrier thread and called Continuation.start()

Example with Thread.sleep()

```
private static void sleepMillis(long millis) throws InterruptedException {
  Thread thread = currentThread();
  if (thread instanceof VirtualThread vthread) {
    long nanos = NANOSECONDS.convert(millis, MILLISECONDS);
    vthread.sleepNanos(nanos);
  } else {
    sleep0(millis);
  }
}
```

```
void sleepNanos(long nanos) throws ...
  long remainingNanos = ...;
  while (remainingNanos > 0) {
    parkNanos(remainingNanos);
    ...
  }
}
```

```
void parkNanos(long nanos) {
  long startTime = System.nanoTime();
  boolean yielded;
  Future<?> unparker = scheduleUnpark(nanos);
  setState(PARKING);
  try {
    yielded = yieldContinuation();
  } finally {
    cancel(unparker);
  }

  // park on the carrier thread for remaining time when pinned
  if (!yielded) {
    parkOnCarrierThread(true, deadline - System.nanoTime());
  }
}
```

yield() can fail !

Synchronized block are written in assembly and uses an address on stack

=> the stack frames can not be copied

Native code that does an upcall to Java may use an address on stack

=> the stack frames can not be copied

Stealth rewrite of the JDK for Loom

- Java 13
  - JEP 353 Reimplement the Legacy Socket API
- Java 15
  - JEP 373 Reimplement the Legacy DatagramSocket API
  - JEP 374 Deprecate and Disable Biased Locking
- Java 18
  - JEP 416 Reimplement Core Reflection with Method Handles

Loom: under the Hood

VM creates as many virtual threads as the user want

- It mounts a virtual thread to an available carrier thread when starting
- if blocking, unmount the current virtual thread and mount another virtual thread

There are still some issues

Synchronized blocks

=> use ReentrantLock instead

Native code that does an upcall

=> no such call in the JDK anymore

Problems with some libraries using native code, Hadoop, Spark, ...

Thread Local issue

1_000_000 threads => 1_000_000 thread locals ??

ThreadLocal issue

ThreadLocal implementation store the values in a Map inside java.lang.Thread

- Does not scale well !

=> provide a more lightweight implementation

jdk.incubator.concurrent.ScopeLocal

DEMO

Thread Local

```
private static final ThreadLocal<String> USER = new ThreadLocal<>();

private static void sayHello() {
  System.out.println("Hello " + USER.get());
}

public static void main(String[] args) throws InterruptedException {
  var vthread = Thread.ofVirtual()
      .allowSetThreadLocals(true)
      .start(() -> {
    USER.set("Bob");
    try {
      sayHello();
    } finally {
      USER.remove();
    }
  });

  vthread.join();
}
```

Can be used to disallow thread locals
throw an ISE when calling ThreadLocal.set()

Scope Local

```
private static final ScopeLocal<String> USER = ScopeLocal.newInstance();

private static void sayHello() {
  System.out.println("Hello " + USER.get());
}

public static void main(String[] args) throws InterruptedException {
  var vthread = Thread.ofVirtual()
      .allowSetThreadLocals(false)
      .start(() -> {
  ScopeLocal.where(USER, "Bob", () -> {
      sayHello();
    });
  });

  vthread.join();
}
```

Assign the value for the scope

ScopeLocal

WARNING API in progress

- Replacement for ThreadLocal
- Stores the value inside the stack, not inside java.lang.Thread
  - => faster (if not too many locals)
  - => use far less memory
- API amenable to JITs

Executor and structured concurrency

Executors

An executor recycle the threads

- Do we need an executor if creating a virtual thread does not cost much ?

An executor as another role

- Manage all the submitted task
  - But cancellation/exception management is wrong !

WARNING API in progress

Structured Concurrency

Use syntactic constructions to represent the dependency tree of the tasks

DEMO

VirtualThreadPerTaskExecutor

```java
var executor = Executors.newCachedThreadPool();
//var executor = Executors.newVirtualThreadPerTaskExecutor();

var future1 = executor.submit(() -> {
  Thread.sleep(10);
  return 42;
});
var future2 = executor.submit(() -> {
  Thread.sleep(1_000);
  return 100;
});

executor.shutdown();

var result = future1.get() + future2.get();
System.out.println(result);

// everything is fine here, right !
```

Special executor using virtual threads
Warning! the carrier threads are *deamon*

Running task (1) with an Executor

```
var executor = Executors.newCachedThreadPool();

var future1 = executor.submit(() -> {
  Thread.sleep(10);
  return 42;
});

var future2 = executor.submit(() -> {
  Thread.sleep(1_000);
  System.out.println("end");
  return 100;
});

executor.shutdown();

//var result = future1.get() + future2.get();
var result = future1.get();
System.out.println(result);

// future2 still running here !
```

Oops !

Running task (2) with an Executor

```
var executor = Executors.newCachedThreadPool();

var future1 = executor.<Integer>submit(() -> {
  throw new AssertionError("oops");
});

var future2 = executor.submit(() -> {
  Thread.sleep(1_000);
  System.out.println("end");
  return 100;
});

executor.shutdown();

Try {
  var result = future1.get() + future2.get();
  System.out.println(result);
} catch(ExecutionException e) {
  throw new AssertionError(e.getCause());
}

// future2 still running here !
```

Oops !

jdk.incubator.concurrent.StructuredTaskScope

```
try (var scope = new StructuredTaskScope<>()) {
  var start = System.currentTimeMillis();

  var future1 = scope.fork(() -> {
    Thread.sleep(1_000);
    return 42;
  });

  var future2 = scope.fork(() -> {
    Thread.sleep(1_000);
    return 100;
  });

  scope.join();

  var end = System.currentTimeMillis();
  System.out.println("elapsed " + (end – start));
  var result = future1.resultNow() + future2.resultNow();
  System.out.println(result);
} // call close() !
```

Wait for all computations

Throw an exception in case of dangling tasks

Future state()

```
try (var scope = new StructuredTaskScope<>()) {
  var future = scope.fork(() -> {
    Thread.sleep(1_000);
    return 42;
  });

  System.out.println(future.state());  // RUNNING

  //scope.shutdown();

  scope.join();

  System.out.println(future.state());  // SUCCESS
}
```

Future state() with a shutdown()

```
try (var scope = new StructuredTaskScope<>()) {
  var future = scope.fork(() -> {
    Thread.sleep(1_000);
    return 42;
  });

  System.out.println(future.state());  // RUNNING

  scope.shutdown();

  scope.join();

  System.out.println(future.state());  // CANCEL
}
```

StructuredTaskScope.shutdown()

If called by the main thread

- Shutdown all the tasks

If called by one task

- Shutdown all the tasks but the caller task

Shutdown() on failure

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
  var start = System.currentTimeMillis();

  var future1 = scope.<Integer>fork(() -> {
    throw new AssertionError("oops");
  });

  var future2 = scope.fork(() -> {
    Thread.sleep(1_000);
    System.out.println("end");
    return 42;
  });

  scope.join();

  var end = System.currentTimeMillis();
  System.out.println("elapsed " + (end – start));
  var result = future1.resultNow() + future2.resultNow();
  System.out.println(result);

}  // future and future2 are not running here !
```

This task fails

Shutdown() on success

```
try (var scope = new StructuredTaskScope.ShutdownOnSuccess<Integer>()) {
  var start = System.currentTimeMillis();

  var future1 = scope.fork(() -> {
    Thread.sleep(1_000);
    return 42;
  });

  var future2 = scope.fork(() -> {
    Thread.sleep(42);
    return 100;
  });

  scope.join();

  var end = System.currentTimeMillis();
  System.out.println("elapsed " + (end – start));
  //System.out.println(future1.resultNow());
  //System.out.println(future2.resultNow());
  System.out.println(scope.result());
}
```

This task completes first

The result is stored in the scope

Summary

Loom

Will be integrated soon Java 19 (or 20)

Introduce virtual threads (better for latency)

- Synchronized() block and native code with an upcall make the code slower

More APIs to come (jdk.incubator.concurrent)

- ScopeLocal
- StructuredTaskScope

Questions ?

https://github.com/forax/loom-fiber