Going inside Java's Project Loom and virtual threads
Introducing Project Loom
Thread builders
Programming with virtual threads
A cautionary tale
When will Project Loom arrive?
Dig deeper

CODING

Going inside Java's Project Loom and virtual threads

See how virtual threads bring back the old days of Java's green thread—that is, Java threads not tied to operating-system threads.

by Ben Evans

January 15, 2021

Let's talk about Project Loom, which is exploring new Java language features, APIs, and runtimes for lightweight concurrency—including new constructs for virtual threads.

Java was the first mainstream programming platform to bake threads into the core language. Before threads, the state of the art was to use multiple processes and various unsatisfactory mechanisms (UNIX shared memory, anyone?) to communicate between them.

At an operating system (OS) level, threads are independently scheduled execution units that belong to a process. Each thread has an execution instruction counter and a call stack but shares a heap with every other thread in the same process.

Not only that, but the Java heap is just a single contiguous subset of the process heap, at least in the HotSpot JVM implementation (other JVMs may differ), so the memory model of threads at an OS level carries over naturally to the Java language domain.

The concept of threads naturally leads to a notion of a lightweight context switch: It is cheaper to switch between two threads in the same process than between threads in a different processes. This is primarily because the mapping tables that convert virtual memory addresses to physical memory addresses are mostly the same for threads in the same process.

By the way, creating a thread is also cheaper than creating a process. Of course, the exact extent to which this is true

depends on the details of the OS in question.

The *Java Language Specification* does not mandate any particular mapping between Java threads and OS threads, assuming that the host OS has a suitable thread concept—which has not always been the case.

In fact, in very early Java versions, the JVM threads were multiplexed onto OS threads (also known as *platform threads*), in what were referred to as *green threads* because those earliest JVM implementations actually used only a single platform thread.

However, this single platform thread practice died away around the Java 1.2 and Java 1.3 era (and slightly earlier on Sun's Solaris OS). Modern Java versions running on mainstream OSs instead implement the rule that one Java thread equals exactly one OS thread.

This means that using `Thread.start()` calls the thread creation system call (such as `clone()` on Linux) and actually creates a new OS thread.

OpenJDK's Project Loom aims, as its primary goal, to revisit this long-standing implementation and instead enable new `Thread` objects that can execute code but do *not* directly correspond to dedicated OS threads.

Or, to put it another way, Project Loom creates an execution model where an object that represents an execution context is not necessarily a thing that needs to be scheduled by the OS. Therefore, in some respects, Project Loom *is* a return to something similar to green threads.

However, the world has changed a lot in the intervening years, and sometimes there are ideas in computing that are ahead of their time.

For example, you could regard EJBs (that is, Jakarta Enterprise Beans, formerly Enterprise JavaBeans) as a form of restricted environment that over-ambitiously tried to virtualize the environment away. Can EJBs perhaps be thought of as a prototypical form of the ideas that would later find favor in modern PaaS systems and, to a lesser extent, in Docker and Kubernetes?

So, if Loom is a (partial) return to the idea of green threads, then one way of approaching it might be via this question: What has changed in the environment that makes it interesting to return to an old idea that was not found to be useful in the past?

To explore this question a little, let's look at an example. Specifically, let's try to crash the JVM by creating too many threads, as follows:

```
//
// Please do not actually run this code... it
//
public class CrashTheVM {
    private static void looper(int count) {
        var tid = Thread.currentThread().getI
        if (count > 500) {
            return;
        }
        try {
            Thread.sleep(10);
            if (count % 100 == 0) {
                System.out.println("Thread id
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        looper(count + 1);
    }

    public static Thread makeThread(Runnable
        return new Thread(r);
    }

    public static void main(String[] args) {
        var threads = new ArrayList<Thread>()
        for (int i = 0; i < 20_000; i = i + 1
            var t = makeThread(() -> looper(1
            t.start();
            threads.add(t);
            if (i % 1_000 == 0) {
                System.out.println(i + " thre
            }
        }
        // Join all the threads
        threads.forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException e)
                e.printStackTrace();
            }
        });
    }
}
```

The code starts up 20,000 threads and does a minimal amount of processing in each one, or at least tries to. In practice, the application will probably die or lock up the machine long before that steady state is reached, though it is possible to get the example to run through to completion if the machine or OS are throttled and can't create threads fast enough to induce the resource starvation.

Figure 1 shows an example of what happened on my 2019 MacBook Pro right before the machine became totally unresponsive. This image shows inconsistent statistics, such as the thread count, because the OS is already struggling to keep up.

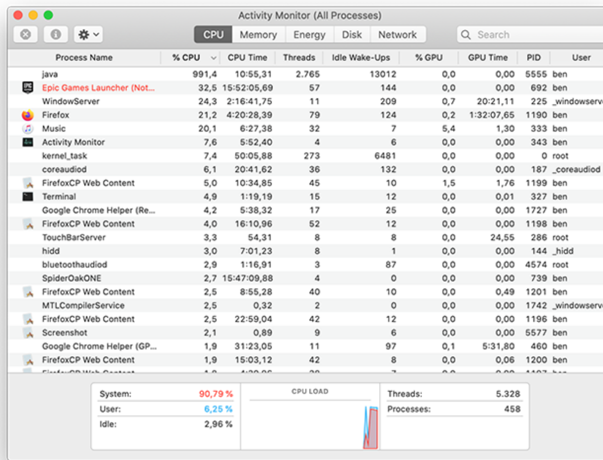

Figure 1. An image showing too many threads: Do not try this at home.

While it is obviously not completely representative of a practical production Java application, this example signposts what will happen to, for example, a web serving environment with one thread per connection. It is entirely reasonable for a modern high-performance web server to be expected to handle tens of thousands (or more) concurrent connections, and yet this example clearly demonstrates the failure of a thread-per-connection architecture for that case.

To put it another way: A modern program may need to keep track of many more executable contexts than it can create threads for.

An alternative takeaway could be that threads are potentially much more expensive than most people think and represent a scaling bottleneck for modern JVM applications. Developers have been trying to solve this problem for years, either by taming the cost of threads or by using a representation of execution contexts that aren't threads.

One way of trying to achieve this was the staged event-driven architecture (SEDA) approach, which first appeared 15 years ago. SEDA can be thought of as a system in which a domain object is moved from A to Z along a multistage pipeline with various different transformations happening along the way. This can be implemented in a distributed system using a messaging system or, in a single process, using blocking queues and a thread pool for each stage.

At each step of the SEDA approach, the processing of the domain object is described by a Java object that contains code to implement the step transformation. For this to work correctly, the code must be guaranteed to terminate; there must be no infinite loops. However, this requirement cannot be enforced by the framework.

There are notable shortcomings to the SEDA approach, not least of which is the discipline required by programmers to use the

architecture. Let's look for a better alternative, and that's Project Loom.

Introducing Project Loom

Project Loom is an OpenJDK project that aims to enable "easy-to-use, high-throughput lightweight concurrency and new programming models on the Java platform." The project aims to accomplish this by adding these new constructs:

- Virtual threads
- Delimited continuations
- Tail-call elimination

The key to all of this is *virtual threads*, which are designed to look to the programmer just like ordinary, familiar threads. However, virtual threads are managed by the Java runtime and are *not* thin, one-to-one wrappers over OS threads. Instead, virtual threads are implemented in user space by the Java runtime. (This article will not cover delimited continuations or tail-call elimination, but you can read about them here.)

The major advantages that virtual threads are intended to bring include

- Creating and blocking them is cheap.
- Java execution schedulers (thread pools) can be used.
- There are no OS-level data structures for the stack.

The removal of the involvement of the OS in the lifecycle of a virtual thread is what removes the scalability bottleneck. Large-scale JVM applications can cope with having millions or even billions of objects, so why should they be restricted to just a few thousand OS-schedulable objects (which is one way to think about what a thread is)?

Shattering this limitation and unlocking new concurrent programming styles is the main aim of Project Loom.

Let's see virtual threads in action. Download a Project Loom beta build and spin up `jshell`, as shown in the following example:

```
$ jshell
|  Welcome to JShell -- Version 16-loom
|  For an introduction type: /help intro

jshell> Thread.startVirtualThread(() -> {
   ...>     System.out.println("Hello World")
   ...> });
Hello World
$1 ==> VirtualThread[<unnamed>,<no carrier th

jshell>
```

You can straightaway see the virtual thread construct in the output. The code is also using a new static method, `startVirtualThread()`, to start the lambda in a new execution context, which is a virtual thread. It's that simple!

Virtual threads must be opted-in: Existing codebases must continue to run in *exactly* the way they ran before the advent of Project Loom. Nothing can break, and everyone must make the conservative assumption that all existing Java code genuinely needs the standing "lightweight wrapper over OS" thread architecture that has been, until now, the only game in town.

So, what's the benefit? Well, the arrival of virtual threads opens up new horizons in other ways. Until now, the Java language has offered two primary ways of creating new threads:

- Subclass `java.lang.Thread` and call the inherited `start()` method.
- Create an instance of `Runnable` and pass it to a `Thread` constructor; then start the resulting object.

Since the concept of threads is changing, it makes sense to re-examine the methods you use to create threads as well. You have already met the new static factory method for fire-and-forget virtual threads, but the existing thread API needs to be improved in a few other ways as well.

Thread builders

One important new notion is the `Thread.Builder` class, which has been added as an inner class of `Thread`. See it in action by replacing the `makeThread()` method in the previous example with this code:

```
public static Thread makeThread(Runnable r) {
        return Thread.builder().virtual().tas
    }
```

This code calls the `virtual()` method on the builder to explicitly create a virtual thread that will execute the `Runnable`. You could, of course, have omitted the call to `virtual()` and that would have created a traditional, OS-schedulable thread object. But where's the fun in that?

If you substitute the virtual version of `makeThread()` and recompile the example with a version of Java that supports Loom, you can execute the resulting binary.

This time, the program runs to completion without an issue, and the overall load profile looks like **Figure 2**.

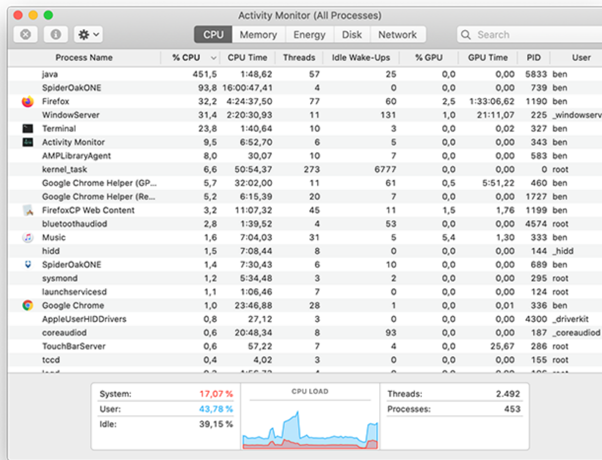

Figure 2. Creating lots of virtual threads instead of traditional Java threads

This is just one example of the Project Loom philosophy in action, which is to localize the changes you need to make to your Java applications to *only* the code locations that create threads.

One way in which the new thread library encourages developers to move on from older paradigms is that subclasses of `Thread` cannot be virtual. Therefore, code that subclasses `Thread` will continue to be created using traditional OS threads. The intention is to protect existing code that uses subclasses of `Thread` and follow the principle of least surprise.

Over time, as virtual threads become more common and developers stop caring about the difference between virtual and OS threads, this should discourage the use of the subclassing mechanism because it will always create an OS-schedulable thread.

Note that other parts of the thread library need to be upgraded to better support Project Loom. For example, `ThreadBuilder` can also build `ThreadFactory` instances that can be passed to various `Executors`, as shown here:

```
jshell> var tb = Thread.builder();
tb ==> java.lang.Thread$BuilderImpl@2e0fa5d3

jshell> var tf = tb.factory();
tf ==> java.lang.Thread$KernelThreadFactory@2

jshell> var vtf = tb.virtual().factory();
vtf ==> java.lang.Thread$VirtualThreadFactory
```

Obviously, at some point, virtual threads must be attached to an actual OS thread to execute. These OS threads upon which a virtual thread executes are called *carrier threads*. Over its lifetime, a single virtual thread may run on several different carrier threads. This is somewhat reminiscent of the way that

regular threads will execute on different physical CPU cores over time—both are examples of execution scheduling.

You have already seen carrier threads in some `jshell` output in one of the earlier examples.

Programming with virtual threads

The arrival of virtual threads brings with it a change of mindset. Programmers who have written concurrent applications in Java as it exists today are used to having to deal (consciously or unconsciously) with the inherent scaling limitations of threads.

Java developers are used to creating task objects, often based on `Runnable` or `Callable`, and handing them off to executors, backed by thread pools that exist to conserve precious thread resources. What if all of that was suddenly different?

Project Loom tries to solve the scaling limitation of threads by introducing a new concept of a thread that is cheaper than existing notions and that doesn't directly map to an OS thread. This new capability still looks and behaves like today's threads that Java programmers already understand.

This means that rather than needing to learn a completely new programming style (such as the continuation-passing style or the promise/future approach or callbacks), the Project Loom runtime keeps the same programming model you know from today's threads for virtual threads. In other words, virtual threads are simply threads, at least as far as the programmer is concerned.

Virtual threads are *preemptive* because the user code does not need to explicitly yield control of the CPU. Scheduling points are up to the virtual scheduler and the JDK. Developers must make no assumptions on when yields happen, because that is purely an implementation detail.

However, to appreciate how virtual threads differ, it is worth understanding the basics of the OS theory that underlies scheduling.

When the OS schedules platform threads, it allocates a *time slice* of CPU time to a thread. When the time slice is up, a hardware interrupt is generated and the kernel is able to resume control, remove the executing platform (user) thread, and replace it with another.

This mechanism is how UNIX (and assorted other OSs) has been able to implement time-sharing of the processor among different tasks, even decades ago in the era when computers had only a single processing core.

Virtual threads, however, are handled differently than platform threads. None of the existing schedulers for virtual threads uses time slices to preempt virtual threads.

Using time slices for preemption of virtual threads would be possible, and the JVM is already capable of taking control of executing Java threads. It does so at JVM *safepoints*, for example.

Instead, virtual threads automatically give up (or *yield*) their carrier thread when a blocking call (such as I/O) is made. This is handled by the library and runtime and is not under the explicit control of the programmer.

Thus, rather than forcing programmers to explicitly manage yielding, or relying upon the complexities of nonblocking or callback-based operations, Project Loom allows Java programmers to write code in traditional thread-sequential style. This has additional benefits such as allowing debuggers and profilers to work in the usual way.

Toolmakers and runtime engineers need to do a bit of extra work to support virtual threads, but that's better than forcing additional cognitive burden onto everyday Java developers.

The designers of Loom expect that because virtual threads need never be pooled, they *should* never be pooled. Instead, the model is the unconstrained creation of virtual threads. For this purpose, an *unbounded executor* has been added. It can be accessed via a new factory method `Executors.newVirtualThreadExecutor()`.

The default scheduler for virtual threads is the work-stealing scheduler introduced in `ForkJoinPool`. (It is interesting how the work-stealing aspect of fork/join has become far more important than the recursive decomposition of tasks.)

The design of Project Loom is predicated on developers understanding the computational overhead that will be present on the different threads in their applications.

Simply put, if there are a vast number of threads that all need a lot of CPU time constantly, your application has a resource crunch that clever scheduling can't help. On the other hand, if there are only a few threads that are expected to become CPU-bound, these should be placed into a separate pool and provisioned with platform threads.

Virtual threads are also intended to work well in the case where there are many threads that are CPU-bound only occasionally. The intent is that the work-stealing scheduler will smooth out the CPU utilization and real-world code will eventually call an operation that passes a yield point (such as blocking I/O).

A cautionary tale

Here's an example of how Project Loom's design can cause some unexpected behavior when custom scheduling of virtual threads is used:

```
public final class TangledLoom {
    public static void main(String[] args) {
        var scheduler = Executors.newFixedThr
        Runnable r = () -> {
            System.out.println(Thread.current
            while (true) {
                int total = 0;
                for (int i = 0; i < 10; i++)
                    total = total + hashing(i
                }
                System.out.println(Thread.cur
            }
        };
        var tA = Thread.builder().virtual(sch
        var tB = Thread.builder().virtual(sch
        var tC = Thread.builder().virtual(sch
        tA.start();
        tB.start();
        tC.start();
        try {
            tA.join();
            tB.join();
            tC.join();
        } catch (Throwable tx) {
            tx.printStackTrace();
        }
    }

    private static int hashing(int length, ch
        final StringBuilder sb = new StringBu
        for (int j = 0; j < length * 1_000_00
            sb.append(c);
        }
        final String s = sb.toString();
        return s.hashCode();
    }
}
```

When you run this code, you should see the following behavior:

```
$ java TangledLoom
B starting
A starting
B : -1830232064
C starting
C : -1830232064
B : -1830232064
C : -1830232064
B : -1830232064
C : -1830232064
B : -1830232064
C : -1830232064
```

This is an example of *thread starvation*; the sad thread `A` never seems to make progress.

Over time, as Project Loom becomes more familiar to Java developers, a common set of patterns will emerge as best practices. But for now, everyone is still in the early days of

learning how to use the new technology effectively, and you should be cautious, as this example shows.

When will Project Loom arrive?

Loom development is taking place in a separate repository; it's not on the JDK mainline. This means it is much too early to talk about when the changes might arrive in an official release of Java.

Early access binaries are available, but these still have some rough edges. Crashes are still not uncommon. The basic API is taking shape, but it is almost certainly not completely finalized yet. There is still a lot of work to be done on the APIs that are being built on top of virtual threads, such as structured concurrency and other more advanced features.

A key question that developers always have is about performance. That's difficult to answer during the early stages of development of a new technology. Things simply are not yet at the point where meaningful comparisons can be made, and the current performance is not thought to be really indicative of the final version.

As with other long-range projects within OpenJDK, the real answer is that it will be ready when it's ready. For now, there is enough of a prototype to start experimenting with Project Loom, so you can get a first taste of what future threading development in Java might look like.

Dig deeper

- Project Loom homepage
- Project Loom: Fibers and continuations for the Java Virtual Machine
- Early access snapshot
- The role of preview features in Java 14, Java 15, Java 16, and beyond
- Webcast: Project Loom: Modern Scalable Concurrency for the Java Platform
- Project Loom on GitHub

Ben Evans

Ben Evans (@kittylyst) is a Java Champion and Principal Engineer at New Relic. He has written five books on programming, including *Optimizing Java* (O'Reilly). Previously he was a founder of jClarity (acquired by Microsoft) and a member of the Java SE/EE Executive Committee.

Share this Page

Contact

US Sales: +1.800.633.0738
Global Contacts
Support Directory
Subscribe to Emails

About Us

Careers
Communities
Company Information
Social Responsibility Emails

Downloads and Trials

Java for Developers
Java Runtime Download
Software Downloads
Try Oracle Cloud

News and Events

Acquisitions
Blogs
Events
Newsroom

ORACLE | Integrated Cloud
Applications & Platform Services

© Oracle | Site Map | Terms of Use & Privacy | Cookie Preferences | Ad Choices