State of Loom

Ron Pressler, May 2020

Project Loom aims to drastically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications that make the best use of available hardware.

Work on Project Loom started in late 2017. This document explains the motivations for the project and the approaches taken, and summarizes our work so far. Like all OpenJDK projects, it will be delivered in stages, with different components arriving in GA (General Availability) at different times, likely taking advantage of the Preview mechanism, first.

You can find more material about Project Loom on its wiki, and try most of what's described below in the Loom EA binaries (Early Access). Feedback to the loom-dev mailing list *reporting on your experience using Loom* will be much appreciated.

```
Thread.startVirtualThread(() -> {
    System.out.println("Hello, Loom!");
});
```

Key Takeaways

- A virtual thread is a `Thread` — in code, at runtime, in the debugger and in the profiler.
- A virtual thread is not a wrapper around an OS thread, but a Java entity.
- Creating a virtual thread is cheap — have millions, and don't pool them!
- Blocking a virtual thread is cheap — be synchronous!
- No language changes are needed.
- Pluggable schedulers offer the flexibility of asynchronous programming.

Contents

Why

Threads Are What It's All About

Java is used to write some of the biggest, most scalable applications in the world. Scalability is the ability of a program to gracefully handle growing workloads. One way in which Java programs scale is *parallelism*: we want to process a chunk of data that could grow quite large, so we describe its transformation as a pipeline of lambdas on a stream, and by setting it to `parallel` we ask multiple processing cores to sink their teeth into the task as one, like a swarm of piranhas devouring a large fish; one piranha could get the job done — it's just faster this way. That mechanism was delivered in Java 8. But there's a different, harder and more prevalent, kind of scaling, which is about handling relatively independent tasks demanded of the application at the same time. That they must be served simultaneously is not an implementation choice but a requirement. We call that *concurrency*, it's the bread-and-butter of contemporary software, and that's what Loom is about.

Consider the web server. Each of the requests it serves is largely independent of the others. For each, we do some parsing, query a database or issue a request to a service and wait for the result, do some more processing and send a response. Not only does this process not cooperate with other simultaneous HTTP requests on completing some job, most of the time it doesn't care at all about what other requests are doing, yet it still competes with them for processing and I/O resources. Not piranhas, but taxis, each with its own route and destination, it travels and makes its stops. The presence of other taxis traveling on the same road system does not make any one cab reach its destination any sooner — if anything, it might slow it down — but if only a single cab could be on the city roads at any one time it wouldn't just be a slow transit system, it would be a dysfunctional one. The more taxis that can share the roads without gridlocking downtown, the better the system. From its early days Java supported this kind of job. Servlets allow us to write code that looks straightforward on the screen. It's a simple sequence — parsing, database query, processing, response — that doesn't worry if the server is now handling just this one request or a thousand others.

Every concurrent application has some units of concurrency natural to its domain, some work that is done independently of other work and at the same time. For a web server, this could be the HTTP request or the user session; for a database server this could be the transaction. Concurrency has a long and rich history that predates Java's, but the idea, as far as how Java was designed, is simple: represent this domain unit of concurrency with a software unit of concurrency that runs sequentially, like a taxi going on its simple route without caring about any others. This software construct is the thread. It virtualizes resources, from processors to I/O devices, and schedules their use — exploiting the fact that each process might employ different hardware units at different times — exposing it as a sequential process. The defining feature of threads is that they sequence not only processing operations but also *blocking* — waiting for some external occurrence, be it I/O or some event or triggered by another thread, continuing execution only after that occurrence. The question of how threads should best communicate with one another — what the appropriate mix of shared data structures and message passing should be — is not essential to the concept of threads, and it's possible that whatever the current mix is in Java applications, it will shift with new features.

Whether you use them directly or inside, say, a JAX-RS framework, concurrency in Java means threads. In fact, the entire Java platform — from the VM, through the language and libraries, to the debuggers and profilers — is built around the thread as the core organizing component of a running program:

- I/O APIs are *synchronous* and describe initiating I/O operations and waiting for their result as a sequential ordering of statements by *blocking* the thread;
- Memory side-effects (if organized to be race-free) are sequentially ordered by the actions of the thread, as if no other thread is competing to use that memory;
- exceptions provide useful information by placing the failing operation in the context of the current thread's call-stack;
- single-stepping in a debugger follows execution in order, whether it entails some processing or I/O because single-stepping is associated with a thread;
- application profiles organize work by threads when showing how much time is spent processing or waiting for I/O or synchronization.

The problem is that the thread, the software unit of concurrency, cannot match the scale of the application domain's natural units of concurrency — a session, an HTTP request, or a single database operation — nor can it match the scale of concurrency that modern hardware can support. A server can handle upward of a million concurrent open sockets, yet the operating system cannot efficiently handle more than a few thousand active (non-idle) threads. As the workload on the servlet container increases and more requests are in flight, its ability to scale is hampered by the relatively small number of threads the operating system can support because Little's law tells us that the average duration of servicing a request is directly proportional to the number of requests we can service concurrently. So if we represent a domain unit of concurrency with a thread, the scarcity of threads becomes our scalability bottleneck long before the hardware does.[1] Servlets read nicely but scale poorly.

This is not a fundamental limitation of the concept of threads, but an accidental feature of their implementation in the JDK as trivial wrappers around operating system threads. OS threads have a high footprint, creating them requires allocating OS resources, and scheduling them — i.e. assigning hardware resources to them — is suboptimal. They're not taxis so much as trains.

This creates a large mismatch between what threads were *meant* to do — abstract the scheduling of computational resources as a straightforward construct — and what they effectively *can* do. A mismatch in several orders of magnitude can have a big impact.

Propping Threads Up by Missing Their Point

And a big impact it's had. Ironically, the threads invented to virtualize scarce computational resources for the purpose of transparently sharing them, have themselves become scarce resources, and so we've had to erect complex scaffolding to share *them*.

Because threads are costly to create, we pool them. The cost of creating a new thread is so high that to reuse them we happily pay the price of leaking thread-locals and a complex cancellation protocol.

But pooling alone offers a thread-sharing mechanism that is too coarse-grained. There just aren't enough threads in a thread pool to represent all the concurrent tasks running even *at a single point in time*. Borrowing a thread from the pool for the entire duration of a task holds on to the thread even while it is waiting for some external event, such as a response from a database or a service, or any other activity that would block it. OS threads are just too precious to hang on to when the task is just waiting. To share threads more finely and efficiently, we could return the thread to the pool every time the task has to wait for some result. This means that the task is no longer bound to a single thread for its entire execution. It also means we must avoid blocking the thread because a blocked thread is unavailable for any other work.

The result is the proliferation of *asynchronous* APIs, from asynchronous NIO in the JDK, through asynchronous servlets, to the many so-called "reactive" libraries that do just that — return the thread to the pool while the task is waiting, and go to great lengths to not block threads. Chopping down tasks to pieces and letting the asynchronous construct put them together results in intrusive, all-encompassing and constraining frameworks. Even basic control flow, like loops and try/catch, need to be reconstructed in "reactive" DSLs, some sporting classes with hundreds of methods.

Because, as mentioned above, much of the Java platform assumes that execution context is embodied in a thread, all that context is lost once we dissociate tasks from threads: Exception stack traces no longer provide a useful context, when single-stepping in the debugger we find ourselves in scheduler code, jumping from one task to another, and the profile of an application under I/O load might show us idle thread pools because tasks waiting for I/O do not keep holding their thread by blocking and, instead, return it to the pool.

This style is now used by some not because it is easier to understand — many programmers report that for them it is harder; not because it is easier to debug or profile — it's *much* harder; not because it fits well with the rest of the language or integrates well with existing code or can be hidden away in "experts-only code"— quite the opposite, it is virally-intrusive and makes clean integration with ordinary synchronous code virtually impossible, but just because the implementation of threads in Java is inadequate in both footprint and performance. The asynchronous programming style fights the design of the Java platform at every turn and pays a high price in maintainability and observability. But it does so *for a good reason*: to meet the scalability and throughput demands and make good use of costly hardware resources.

Some programming languages tried to address the problem of thorny asynchronous code by building a *new* concept on top of threads: async/await.[2] It works similarly to threads but cooperative scheduling points are marked explicitly with `await`. This makes it possible to write scalable synchronous code and solves the context issue by introducing a new kind of context that is a thread in all but name but is *incompatible with threads*. If synchronous and asynchronous code cannot normally be mixed — one blocks and the other returns some sort of `Future` or `Flow` — async/await creates two differently "colored" worlds that cannot be mixed even though *both* of them are synchronous, and, to make matters more confusing, calls synchronous code asynchronous to indicate that, despite being synchronous, no thread is blocked. As a result, C# requires *two* different APIs to suspend execution of the currently executing code for some predetermined duration, and Kotlin does too, one to suspend the thread and one to suspend the new construct that is *like* a thread, but isn't. The same goes for all synchronous APIs, from synchronization to I/O, that are duplicated. Not only is there no single abstraction of two implementations of the same concept, but the two worlds are syntactically disjoint, requiring the programmer to mark her code units as suitable to run in one mode or the other, but not both.

Moreover, explicit cooperative scheduling points provide little benefit on the Java platform. The VM is optimized for peak performance, not deterministic worst-case latency like a realtime OS, and so it might nondeterministically introduce various pauses at arbitrary points in the program, for GC, for deoptimization, not to mention arbitrary, nondeterministic and indefinite preemption by the OS. The duration of a blocking operation can range from several orders of magnitude longer than those nondeterministic pauses to several orders of magnitude *shorter*, and so explicitly marking them is of little help. A better way to control latency, and at a more appropriate granularity, is deadlines.

The mechanisms built to manage threads as a scarce resource are an unfortunate case of a good abstraction abandoned in favor of another, worse in most respects, merely because of the runtime performance characteristics of the implementation. This state of affairs has had a large deleterious effect on the Java ecosystem.

Programmers are forced to choose between modeling a unit of domain concurrency directly as a thread and wasting considerable throughput that their hardware can support, or using other ways to implement concurrency on a very fine-grained level but relinquishing the strengths of the Java platform. Both choices have a considerable financial cost, either in hardware or in development and maintenance effort.

We can do better.

Project Loom intends to eliminate the frustrating tradeoff between efficiently running concurrent programs and efficiently writing, maintaining and observing them. It leans into the strengths of the platform rather than fight them, and also into the strengths of the efficient components of asynchronous programming. It lets you write programs in a familiar style, using familiar APIs, and in harmony with the platform and its tools — but also with the hardware — to reach a balance of write-time and runtime costs that, we hope, will be widely appealing. It does so without changing the language, and with only minor changes to the core library APIs. A simple, synchronous web server will be able to handle many more requests without requiring more hardware.

Right-Sizing Threads

If we could make threads lighter, we could have more of them. If we have more of them, they could be used as intended: to directly represent domain units of concurrency by virtualizing scarce computational resources and hiding the complexity of managing those resources. This is not a new idea, and is perhaps most familiar as the approach taken in Erlang and Go.

Our foundation is **virtual threads**. Virtual threads are just threads, but creating and blocking them is cheap. They are managed by the Java runtime and, unlike the existing platform threads, are not one-to-one wrappers of OS threads, rather, they are implemented in userspace in the JDK.

OS threads are heavyweight because they must support all languages and all workloads. A thread requires the ability to suspend and resume the execution of a computation. This requires preserving its state, which includes the instruction pointer, or program counter, that contains the index of the current instruction, as well as all of the local computation data, which is stored on the stack. Because the OS does not know how a language manages its stack, it must allocate one that is large enough. Then we must *schedule* executions when they become *runnable* — started or unparked — by assigning them to some free CPU core. Because the OS kernel must schedule all manner of threads that behave very differently from one another in their blend of processing and blocking — some serving HTTP requests, others playing videos — its scheduler must be an adequate all-around compromise.

Loom adds the ability to control execution, suspending and resuming it, by reifying its state not as an OS resource, but as a Java object known to the VM, and under the direct control of the Java runtime. Java objects securely and efficiently model all sorts of state machines and data structures, and so are well suited to model execution, too. The Java runtime knows how Java code makes use of the stack, so it can represent execution state more compactly. Direct control over execution also lets us pick schedulers — ordinary Java schedulers — that are better-tailored to our workload; in fact, we can use pluggable custom schedulers. Thus, the Java runtime's superior insight into Java code allows us to shrink the cost of threads.

Whereas the OS can support up to a few thousand active threads, the Java runtime can support millions of virtual threads. Every unit of concurrency in the application domain can be represented by its own thread, making programming concurrent applications easier. Forget about thread-pools, just spawn a new thread, one per task. You've already spawned a new virtual thread to handle an incoming HTTP request, but now, in the course of handling the request, you want to simultaneously query a database and issue outgoing requests to three other services? No problem — spawn *more* threads. You need to wait for something to happen without wasting precious resources? Forget about callbacks or reactive stream chaining — just block. Write straightforward, boring code. All the benefits threads give us — control flow, exception context, debugging flow, profiling organization — are preserved by virtual threads; only the runtime cost in footprint and performance is gone. There is no loss in flexibility compared to asynchronous programming because, as we'll see, we have not ceded fine-grained control over scheduling.

Migration: From Threads to (Virtual) Threads

With new capabilities on hand, we knew how to *implement* virtual threads; how to represent those threads to programmers was less clear.

Every new Java feature creates a tension between conservation and innovation. Forward compatibility lets existing code enjoy the new feature (a great example of that is how old code using single-abstract-method types works with lambdas). But we also wish to correct past design mistakes and begin afresh.

The `java.lang.Thread` class dates back to Java 1.0, and over the years accumulated both methods and internal fields. It contains methods such as `suspend`, `resume`, `stop` and `countStackFrames`, that have been deprecated for over twenty years, methods like `getAllStackTraces` that assume that the number of threads is small, antiquated concepts such as the context-classloader, added to support certain application container usages, and even older ones, such as `ThreadGroup`, whose original purpose seems to have been lost to history yet still permeates a lot of internal code and tools that deal with threads, including outdated, unused methods such as `Thread.enumerate`.

Indeed, earlier prototypes of Loom represented our user-mode threads in a new `Fiber` class that helped us examine the dependencies of existing code on the thread API. Several observations made during that experiment helped shape our position:

- *Some* parts of the thread API are very pervasively used, in particular, **`Thread.currentThread()`** and **`ThreadLocal`**. Without them very little existing code can run. We tried making **`ThreadLocal`** mean thread-or-fiber-local and had **`Thread.currentThread()`** return some **`Thread`** view of the **`Fiber`**, but these were added complications.
- Other parts of the **`Thread`** API are not only seldom used, but are hardly exposed to programmers at all. Since Java 5, programmers have been encouraged to create and start threads indirectly through **`ExecutorService`**s, so that the clutter in the **`Thread`** class is not terribly harmful; new Java developers need not be exposed to most of it and not at all to its antiquated vestiges. So the pedagogical cost of keeping the **`Thread`** API is small.
- We could reduce the footprint of the metadata in the **`Thread`** class by moving it to a "sidecar" object to only be allocated on demand.
- The new deprecation and removal policy will gradually allow us to clean up the **`Thread`** API.
- We couldn't come up with anything sufficiently better than **`Thread`** that would justify a completely new API.

There are still some inconveniences like unfortunate return types and the interruption mechanism, but what we learned during that experiment — that we can keep parts of the Thread API and de-emphasize others — moved the needle in favor of keeping the existing API, and representing our user-mode threads with the **`Thread`** class. And here we are: virtual threads are just **`Thread`**s, and any library that knows **`Thread`** already knows virtual threads. Debuggers and profilers work with them as with today's threads. Unlike async/await, they introduce no "semantic gap": the behavior of the code as it appears to the programmer on the screen is preserved at runtime, and appears the same to all tools.

(You Already Know) How to Program with Virtual Threads

Creating and starting a virtual thread can be done like so:

```
Thread t = Thread.startVirtualThread(() -> { ... });
```

For more flexibility, there's the new `Thread.Builder`, that can do the same thing as above:

```
Thread t = Thread.builder().virtual().task(() -> { ... }).start();
```

or create an unstarted virtual thread:

```
Thread t = Thread.builder().virtual().task(() -> ...).build();
```

There is no public or protected **`Thread`** constructor to create a virtual thread, which means that subclasses of **`Thread`** cannot be virtual. Because subclassing platform classes constrains our ability to evolve them, it's something we want to discourage.

The builder can also create a **`ThreadFactory`**,

```
ThreadFactory tf = Thread.builder().virtual().factory();
```

that can be passed to `java.util.concurrent.Executors` to create **`ExecutorService`**s that employ virtual threads and used as usual. But since we don't need and don't want and to pool virtual threads, we've added a new method, **`newUnboundedExecutor`** to **`Executors`**. It constructs an **`ExecutorService`** that creates and starts a new thread for every submitted task *without pooling* — when a task terminates, its thread terminates:

```
ThreadFactory tf = Thread.builder().virtual().factory();
ExecutorService e = Executors.newUnboundedExecutor(tf);
Future<Result> f = e.submit(() -> { ... return result; }); // spawns a new virtual
        thread
...
Result y = f.get(); // joins the virtual thread
```

The baggage of the **`Thread`** API doesn't bother us, as we don't use it directly.

Other than constructing the **`Thread`** object, everything works as usual, except that the vestigial **`ThreadGroup`** of all virtual threads is fixed and cannot enumerate its members. **`ThreadLocal`**s work for virtual threads as they do for the platform threads, but as they might drastically increase memory footprint merely because there can be a great many virtual threads, **`Thread.Builder`** allows the creator of a thread to forbid their use in that thread. We're exploring an alternative to **`ThreadLocal`**, described in the Scope Variables section.

The introduction of virtual threads does not remove the existing thread implementation, supported by the OS. Virtual threads are just a new implementation of `Thread` that differs in footprint and scheduling. Both kinds can lock on the same locks, exchange data over the same `BlockingQueue` etc. A new method, `Thread.isVirtual`, can be used to distinguish between the two implementations, but only low-level synchronization or I/O code might care about that distinction.

However, the existence of threads that are so lightweight compared to the threads we're used to does require some mental adjustment. First, we no longer need to avoid blocking, because blocking a (virtual) thread is not costly. We can use all the familiar synchronous APIs without paying a high price in throughput. Second, creating these threads is cheap. Every task, within reason, can have its own thread entirely to itself; there is never a need to pool them. If we don't pool them, how do we limit concurrent access to some service? Instead of breaking the task down and running the service-call subtask in a separate, constrained pool, we just let the entire task run start-to-finish, in its own thread, and use a semaphore in the service-call code to limit concurrency — this is how it *should* be done.

Using virtual threads well does not require learning new concepts so much as it demands we *unlearn* old habits developed over the years to cope with the high cost of threads and that we've come to automatically associate with threads merely because we've only had the one implementation.

In the remainder of this document, we will discuss how virtual threads extend beyond the behavior of classical threads, pointing out a few new API points and interesting use-cases, and observing some of the implementation challenges. But all you need to use virtual threads successfully has already been explained.

Scheduling

Unlike the kernel scheduler that must be very general, virtual thread schedulers can be tailored for the task at hand. The same kind of flexible scheduling offered by asynchronous programming can be employed by virtual threads, but because the result is a thread and the details of scheduling well-hidden, you don't need to understand how it works any more than you study the kernel scheduler, unless you intend to use or write a custom scheduler yourself. Otherwise, this section is completely optional.

Outside the kernel we don't have direct access to CPU cores, so we use kernel threads as an approximation: our scheduler will schedule virtual threads' computations onto "physical" platform worker threads. We call the scheduler's workers *carrier* threads, as they carry the virtual threads on their backs. Like asynchronous frameworks, we end up scheduling kernel threads, except we abstract the result as a thread rather than let the details of scheduling leak into application code.

When a virtual thread becomes runnable the scheduler will (eventually) *mount* it on one of its worker platform threads, which will become the virtual thread's carrier for a time and will run it until it is descheduled — usually when it blocks. The scheduler will then *unmount* that virtual thread from its carrier, and pick another to mount (if there are any runnable ones). Code that runs on a virtual thread cannot observe its carrier; `Thread.currentThread` will always return the current (virtual) thread.

By default, virtual threads are scheduled by a global scheduler with as many workers as there are CPU cores (or as explicitly set with `-Djdk.defaultScheduler.parallelism=N`). A great many virtual threads are, then, scheduled onto a small number of platform threads. This is known as M:N scheduling (M user-mode threads are scheduled onto N kernel threads, where M >> N). Early versions of the JDK, also implemented `Thread`s in userspace as *green threads*; they, however, employed M:1 scheduling, using just one kernel thread.[3]

Work-stealing schedulers work well for threads involved in transaction processing and message passing, that normally process in short bursts and block often, of the kind we're likely to find in Java server applications. So initially, the default global scheduler is the work-stealing `ForkJoinPool`.

Virtual threads are preemptive, not cooperative — they do not have an explicit `await` operation at scheduling (task-switching) points. Rather, they are preempted when they block on I/O or synchronization. Platform threads are sometimes forcefully preempted by the kernel if they occupy the CPU for a duration that exceeds some allotted time-slice. Time-sharing works well as a scheduling policy when active threads don't outnumber cores by much and only very few threads are processing-heavy. If a thread is hogging the CPU for too long, it's preempted to make other threads responsive, and then it's scheduled again for another time-slice. When we have millions of threads, this policy is less effective: if many of them are so CPU-hungry that they require time-sharing, then we're under-provisioned by orders of magnitude and no scheduling policy could save us. In all other circumstances, either a work-stealing scheduler would automatically smooth over sporadic CPU-hogging or we could run problematic threads as platform threads and rely on the kernel scheduler. For this reason, none of the schedulers in the JDK currently employs time-slice-based preemption of virtual threads, but that is not to say it won't in the future — see Forced Preemption.

You must not make any assumptions about where the scheduling points are any more than you would for today's threads. Even without forced preemption, any JDK or library method you call could introduce blocking, and so a task-switching point.

Virtual threads can use an arbitrary, pluggable scheduler. A custom scheduler can be set on a per-thread basis, like so:

```
Thread t = Thread.builder().virtual(scheduler).build();
```

or per-factory, like so:

```
ThreadFactory tf = Thread.builder().virtual(scheduler).factory();
```

The thread is assigned to the scheduler from birth till death.

Custom schedulers can use various scheduling algorithms, and can even choose to schedule their virtual threads onto a particular single carrier thread or a set of them (although, if a scheduler only employs one worker it is more vulnerable to pinning).

The custom scheduler doesn't need to know that it is used to schedule virtual threads. It can be of any type implementing `java.util.concurrent.Executor`, and only needs to implement a single method: `execute`. This method will be called when the thread is made runnable, meaning, requesting to be scheduled, when started or unparked. But what is the `Runnable` instance passed to `execute`? It is a `Thread.VirtualThreadTask` that allows the scheduler to query the virtual thread's identity, and it wraps the internal preserved state of the virtual thread's execution. When the scheduler assigns this `Runnable` to some worker thread that then calls the `run` method, the method will *mount* the virtual thread to become its *carrier*, and the virtual thread's suspended execution will be magically restored and its execution resumed as if on top of the carrier. To the scheduler, the `run` method appears to behave like any other — it executes seemingly in the same thread (in fact, it does run in the same kernel thread) and ostensibly returns when the task terminates, but the code "inside" `run` will observe that it is running in a virtual thread, and `run` will return to the scheduler when the virtual thread blocks, putting the `VirtualThreadTask` in a suspended state. You can think of `VirtualThreadTask` as a `Runnable` embodying the resumption of the virtual thread's execution. This is where the magic happens. This process will be explained in more detail in a separate document about this new VM capability.

The scheduler must never execute the `VirtualThreadTask` concurrently on multiple carriers. In fact, the return from `run` must *happen-before* another call to `run` on the same `VirtualThreadTask`.

Regardless of scheduler, virtual threads exhibit the same memory consistency — specified by the Java Memory Model (JMM)[4] — as platform `Thread`s, but custom schedulers could choose to provide stronger guarantees. For example, a scheduler with a single worker platform thread would make all memory operations totally ordered, not require the use of locks, and would allow using, say, `HashMap` instead of a `ConcurrentHashMap`. However, while threads that are race-free according to the JMM will be race-free on any scheduler, relying on the guarantees of a specific scheduler could result in threads that are race-free in that scheduler but not in others.

Performance and Footprint

Both the task-switching cost of virtual threads as well as their memory footprint will improve with time, before and after the first release.

Performance is determined by the algorithm the VM uses to mount and unmount a virtual thread as well as the behavior of the scheduler. For those that wish to experiment with performance, the VM option `-XX:[-/+]UseContinuationChunks` can be used to select between two underlying algorithms. In addition, the default scheduler, a `ForkJoinPool`, is not optimized for situations where it is under-utilized (there are fewer submitted tasks, i.e. runnable virtual threads, than workers) and performs sub-optimally in such cases, so you may want to experiment with the size of the default scheduler's worker pool (`-Djdk.defaultScheduler.parallelism=N`).

Footprint is determined mostly by the internal VM representation of the virtual thread's state — which, while much better than a platform thread, is still not optimal — as well as the use of thread-locals.

Discussions over the runtime characteristics of virtual threads should be brought to the loom-dev mailing list.

Pinning

We say that a virtual thread is *pinned* to its carrier if it is mounted but is in a state in which it cannot be unmounted. If a virtual thread blocks while pinned, it blocks its carrier. This behavior is still correct, but it holds on to a worker thread for the duration that the virtual thread is blocked, making it unavailable for other virtual threads.

Occasional pinning is not harmful if the scheduler has multiple workers and can make good use of the other workers while some are pinned by a virtual thread. Very frequent pinning, however, will harm throughput.

In the current Loom implementation, a virtual thread can be pinned in two situations: when there is a native frame on the stack — when Java code calls into native code (JNI) that then calls back into Java — and when inside a `synchronized` block or method. In those cases, blocking the virtual thread will block the physical thread that carries it. Once the native call completes or the monitor released (the `synchronized` block/method is exited) the thread is unpinned.

If you have a common I/O operation guarded by a `synchronized`, replace the monitor with a `ReentrantLock` to let your application benefit fully from Loom's scalability boost even before we fix pinning by monitors (or, better yet, use the higher-performance `StampedLock` if you can).

Two commonly used methods in the JDK introduced a native frame that would pin a virtual thread: `AccessController.doPrivileged` and `Method.invoke` (+ its constructor counterpart, `Constructor.newInstance`). `doPrivileged` was rewritten in pure Java. `Method.invoke` uses a native call for some iterations, and after warmup generates Java bytecode; in the Loom prototype, we've reimplemented it in Java using `MethodHandle`s. Static class initializers are also called by native code, but they run very infrequently so we don't worry about them.

In addition, blocking in native code or attempting to obtain an unavailable monitor when entering `synchronized` or calling `Object.wait`, will also block the native carrier thread.

The limitations of `synchronized` will eventually go away, but native frame pinning is here to stay. We do not expect it to have any significant adverse impact because such situations very rarely arise in Java, but Loom will add some diagnostics to detect pinned threads.

All Your Blocking Are Belong to Us

Representing threads as "pure" Java objects is the first step. The second is getting all of your code and libraries to use the new mechanism; otherwise they will block OS threads instead of virtual threads. Luckily, we don't need to change all libraries and all applications. Whenever you run a blocking operation in, say, Spring or Hibernate, it ultimately makes use of some core-library API in the JDK — the `java.*` packages. The JDK controls all the interaction points between the application and the OS or the outside world[5], so all we need to do is to adapt them to work with virtual threads. Everything built on top of the JDK will now work with virtual threads. Specifically, we need to adapt all points in the JDK where we block; these come in two flavors: synchronization (think locks or blocking queues) and I/O. In particular, when a synchronous I/O operation is called on a virtual thread, we want to block the virtual thread, perform a non-blocking I/O operation under the covers, and set it so that when the operation completes it will unblock the virtual thread.

Synchronization

- See limitations of `synchronized`/`Object.wait` in pinning.
- All other forms of synchronization, usually in the `java.util.concurrent` package and libraries that use it, block and unblock threads using the `LockSupport.park/unpark` methods. We've adapted those, so that `java.util.concurrent` is virtual-thread friendly.
- Further tuning to policies in `java.util.concurrent` is still necessary to get the best performance with virtual threads.

I/O

- The `java.nio.channels` classes — `SocketChannel`, `ServerSocketChannel` and `DatagramChannel` — were retrofitted to become virtual-thread-friendly. When their synchronous operations, such as `read` and `write`, are performed on a virtual thread, only non-blocking I/O is used under the covers.
- "Old" I/O networking — `java.net.Socket`, `ServerSocket` and `DatagramSocket` — has been reimplemented in Java on top of NIO, so it immediately benefits from NIO's virtual-thread-friendliness.
- DNS lookups by the `getHostName`, `getCanonicalHostName`, `getByName` methods of `java.net.InetAddress` (and other classes that use them) are still delegated to the operating system, which only provides a OS-thread-blocking API. Alternatives are being explored.

- Process pipes will similarly be made virtual-thread-friendly, except maybe on Windows, where this requires a greater effort.
- Console I/O has also been retrofitted.
- `Http(s)URLConnection` and the implementation of TLS/SSL were changed to rely on `j.u.c` locks and avoid pinning.
- File I/O is problematic. Internally, the JDK uses buffered I/O for files, which always reports available bytes even when a read will block. On Linux, we plan to use io_uring for asynchronous file I/O, and in the meantime we're using the `ForkJoinPool.ManagedBlocker` mechanism to smooth over blocking file I/O operations by adding more OS threads to the worker pool when a worker is blocked.

As a result, libraries that use the JDK's networking primitives — whether in the JDK core library or outside it — will also automatically become non-(OS-thread-)blocking; this includes JDBC drivers, and HTTP clients and servers.

Debugging and Profiling

Serviceability and observability have always been high-priority concerns for the Java platform, and are among its distinguishing features. It was important for us to have a good debugging and profiling experience for virtual threads on day one, especially as these are aspects where virtual threads can offer significant benefits over asynchronous programming, whose particularly *bad* debugging and profiling experience is its own distinguishing feature.

The debugger agent that powers the Java Debugger Wire Protocol (JDWP) and the Java Debugger Interface (JDI) used by Java debuggers and supports ordinary debugging operations such as breakpoints, single stepping, variable inspection etc., works for virtual threads as it does for classical threads. Stepping over a blocking operation behaves as you would expect, and single stepping doesn't jump from one task to another, or to scheduler code, as happens when debugging asynchronous code. This has been facilitated by changes to support virtual threads at the JVM TI level. We've also engaged the IntelliJ IDEA and NetBeans debugger teams to test debugging virtual threads in those IDEs.

In the current EA, not *all* debugger operations are supported for virtual threads. Some operations pose special challenges. For example, debuggers often list all active threads. If you have a million threads, this is both slow and unhelpful. In fact, we do not offer any mechanism to enumerate all virtual threads. Some ideas are being explored, like listing only virtual threads on which some debugger event, such as hitting a breakpoint, has been encountered during the debugging session.

One of the biggest problems with asynchronous code is that it is nearly impossible to profile well. There is no good general way for profilers to group asynchronous operations by context, collating all subtasks in a synchronous pipeline processing an incoming request. As a result, when you try to profile asynchronous code, you often see idle thread pools even when the application is under load, as there is no way to track the operations waiting for asynchronous I/O.

Virtual threads solve this, as synchronous operations are associated with the threads they block (even when employing non-blocking I/O under the covers). We've modified the JDK Flight Recorder (JFR) — the foundation of profiling and structured logging in the JDK — to support virtual threads. Blocked virtual threads can be shown in the profiler and time spent on I/O measured and accounted for.

On the other hand, virtual threads introduce some challenges for observability. For example, how do you make sense of a one-million-thread thread-dump? We believe structured concurrency can help with that.

Sidebar: Why "Virtual"?

In previous iterations of the project, we called our lightweight, user-mode threads "fibers", but found ourselves repeatedly explaining that they are not a new concept but a different implementation of a familiar one — the thread. Also, that term is already used for constructs that are similar yet different enough to cause confusion. "Green threads" is similarly tainted by other implementations. We considered the non-specific "lightweight threads," but "lightweight" is relative and we pictured future JDKs having "micro-threads," and so we settled on Brian Goetz's suggestion to call them "virtual threads," which also tested well in conferences. The name is supposed to evoke a connection with virtual memory: we get more of something (address space, threads) by mapping the virtual construct on top of the concrete one (physical memory, OS threads).

To Part 2: Further Work

Reviewed by Alan Bateman, Chris Plummer, Duncan MacGregor, Andrew Haley, Doug Lea, John Rose, Mark Reinhold, Alex Buckley, and Brian Goetz.

---

1. Unless the application is CPU-bound, but most server applications are not.↩
2. Many of those languages had good reasons to do that, but they don't apply to Java. This discussion is beyond the scope of this document, and will be made in another.↩
3. If virtual threads are threads, can they serve as carriers for other virtual threads? In principle yes, but we explicitly forbid such usage, as it is not useful and would usually indicate a mistake.↩
4. See *JSR 133: Java Memory Model and Thread Specification Revision* and *JSR 133 (Java Memory Model) FAQ*.↩
5. Unless the application or its libraries use JNI to access native code directly.↩