Java 21: Virtual Threads

One of the most significant features of Java 21 is Virtual Threads (JEP 444). These lightweight threads reduce the effort needed for writing, maintaining, and observing high-throughput concurrent applications.

As in many of my other articles, let's look first at the status quo before a new feature to understand betters what problem it tries to solve and what the benefits are.

Platform Threads

Before the introduction of Virtual Threads, the Threads we're used to, `java.lang.Thread`, were backed so-called *platform threads*.

These threads are typically mapped 1:1 to kernel threads scheduled by the OS. OS threads are quite "heavy". That makes them suitable for executing all types of tasks.

Depending on the OS and configuration, they consume somewhere between 2 and 10 MB by default. So if you want to utilize a million threads in your heavy-load concurrent application, you better have more than 2 TB of memory to spare!

As you can see, there's an obvious bottleneck restricting the number of threads we can actually have without drawbacks.

One Thread Per Request

This is problematic, as it collides directly with the typical server-application approach of "one thread per request". Using a single thread per request has many advantages, like easier state management and cleanup. But it also creates a scalability limit. An application's "unit of concurrency", in this case, a request, requires a single "platform unit of concurrency". Therefore, Threads are easier to exhaust as raw CPU power or the network.

Even though "one thread per request" has many advantages, sharing heavy-weight Threads utilizes your hardware more evenly, but a quite different approach is needed.

Asynchronous to the Rescue

Instead of running the whole request on a single thread, each part of it uses a Thread from a pool when their task is done, so another task might reuse the same Thread. This allows your code to require fewer Threads, but introduces the burden of *asynchronous* programming.

Coming with it own paradigms, async programming has a certain learning curve, and can make your program harder to understand and follow. Each part of a request might be executed on a different Thread, creating stack traces without sensible context and making debugging something in between quite tricky to almost impossible.

Revisiting the "one thread per request" model, it's clear that a more lightweight approach to Threads solves the bottleneck and would provide a familiar way of doing things.

Lightweight Threads

As the number of platform threads is something that can't be changed without more hardware, another layer of abstraction is needed, severing the dreaded 1:1 mapping that creates the bottleneck in the first place.

Lightweight threads aren't tied to a particular platform thread and don't come with the huge amount memory per-allocated for them. They're scheduled and managed by the runtime, not the underlying OS. That's why a significantly larger number of lightweight threads can be created.

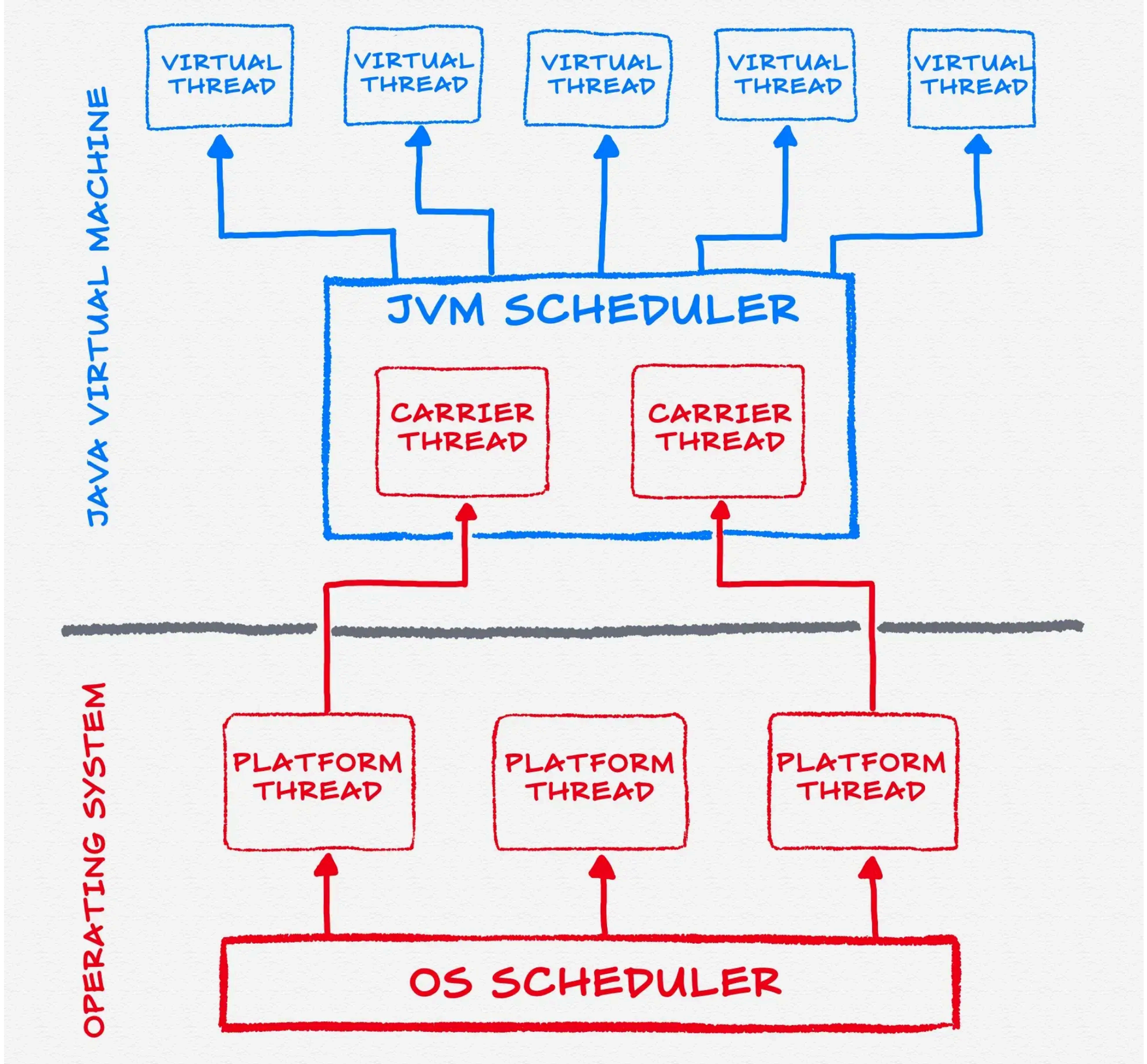
The general concept is nothing new, and many languages have some form of lightweight threads:

- Goroutines in Go
- Erlang Processes
- Haskell Threads
- and many more

Java finally introduced its own implementation of lightweight threads in version 21: *Virtual Threads*.

Virtual Threads

Virtual Threads are a new lightweight `java.lang.Thread` variant and part of Project Loom that isn't managed or scheduled by the OS. Instead, the JVM is responsible for scheduling. Of course, any actual work must be run in a platform thread, but the JVM is utilizing so-called *carrier threads*, which are platform threads, to "carry" any *virtual thread* when its time has come to execute.



JVM/OS Thread Scheduler

The required platform threads are managed in a FIFO work-stealing `ForkJoinPool`, which is using all available processors by default, but can be modified to your requirements by tuning the system property `jdk.virtualThreadScheduler.parallelism`. The main difference between a `ForkJoinPool` you're familiar with, the *common pool* that's used by other features like parallel Streams, is that the common pool operates in LIFO mode.

Cheap and Plentiful Threads

Having cheap and lightweight threads, you can use the "one thread per request" model without worrying about how many threads you actually need. If your code calls a blocking I/O operation in a virtual thread, the runtime suspends the virtual thread until it can be resumed later. This way, the hardware is utilized to an almost optimal level, resulting in high levels of concurrency and, therefore, high throughput.

Because they're so cheap, virtual threads aren't reused or need to be pooled. Each task is represented by its own virtual thread.

Setting Boundaries

The scheduler is responsible for managing the carrier threads, so certain boundaries and separations are required to ensure the possible "bajillion" virtual threads run as intended. This is achieved by maintaining no thread affinity between the carrier thread and any virtual thread it might be carrying:

- A virtual thread can't access the carrier, and `Thread.currentThread()` returns the virtual thread itself.
- Stack traces are separate, and any Exception thrown in a virtual thread only includes its own stack frames.
- Thread-local variables of a virtual thread are unavailable to its carrier, and vice-versa.
- From a code perspective, the sharing of a platform thread by the carrier and its virtual thread is invisible.

Let's See The Code

One of the best things about Virtual threads, in my opinion, is that you don't need to learn new paradigms, or a complicated new API, like you would need with asynchronous programming. Instead, you can approach them almost the same as non-virtual threads.

Creating Platform Threads

Creating a platform thread is simple, like creating using a `Runnable`:

```java
Runnable fn = () -> {
  // your code here
};

Thread thread = new Thread(fn).start();
```

As Project Loom streamlines the new approaches to concurrency, it also provides a new way to create platform-backed threads:

```java
Thread thread = Thread.ofPlatform().
                      .start(runnable);
```

Actually, there's a whole fluent API now, as `ofPlatform` returns a `Thread.Builder.OfPlatform` instance:

```java
Thread thread = Thread.ofPlatform().
                       .daemon()
                       .name("my-custom-thread")
                       .unstarted(runnable);
```

But you're not here to learn new ways to create "old" threads, you want the new stuff!

Creating Virtual Threads

For virtual threads, there's also a fluent API in the same vein:

```java
Runnable fn = () -> {
  // your code here
};

Thread thread = Thread.ofVirtual(fn)
                       .start();
```

Alternatively to the builder approach, you can execute a `Runnable` directly with:

```java
Thread thread = Thread.startVirtualThread(() -> {
  // your code here
});
```

As all Virtual Threads are always daemon threads, don't forget to call `join()` if you want to wait on the main thread.

Another way to create a virtual thread is using an Executor:

```java
var executorService = Executors.newVirtualThreadPerTaskExecutor();

executorService.submit(() -> {
  // your code here

});
```

Conclusion

Even though Scoped Values (JEP 446) and Structured Concurrency (JEP 453) are still preview features in Java 21, Virtual Threads arrived a full-fledged feature are production-ready.

They are a versatile and powerful new approach to Java's concurrency and will have a significant impact on our programs going forward. They use the familiar and reliable "one thread per request" approach while utilizing all the available hardware in the most optimal way, without requiring to learn new paradigms or complicated APIs.