Java Magazine

Written by the Java community for Java and JVM developers

Java 19, JVM Internals

Coming to Java 19: Virtual threads and platform threads

May 20, 2022 | 8 minute read

Nicolai Parlog
Developer Advocate, Oracle

Operating systems can’t increase the efficiency of platform threads, but the JDK will make better use of them by severing the one-to-one relationship between its threads and OS threads.

Now that Project Loom’s JEP 425 officially previews *virtual threads* for Java 19, it’s time to take a close look at them: scheduling and memory management; mounting, unmounting, capturing, and pinning; observability; and what you can do for optimal scalability.

Before I go into virtual threads, I need to revisit classic threads or, what I will call them from here on out, *platform threads*.

The JDK implements platform threads as thin wrappers around operating system (OS) threads, which are costly, so you cannot have too many of them. In fact, the number of threads often becomes the limiting factor long before other resources, such as CPU or network connections, are exhausted.

In other words, platform threads often cap an application’s throughput to a level well below what the hardware could support hardware could support.

That's where virtual threads come in.

Virtual threads

Operating systems can't increase the efficiency of the platform threads, but the JDK can make better use of them by severing the one-to-one relationship between its threads and OS threads.

A virtual thread is an instance of `java.lang.Thread` that requires an OS thread to do CPU work—but doesn't hold the OS thread while waiting for other resources. You see, when code running in a virtual thread calls a blocking I/O operation in the JDK API, the runtime performs a nonblocking OS call and automatically suspends the virtual thread until the operation finishes.

During that time, other virtual threads can perform calculations on that OS thread, so they're effectively sharing it.

Critically, Java's virtual threads incur minimal overhead, so there can be many, many, many of them.

So just as operating systems give the illusion of plentiful memory by mapping a large virtual address space to a limited amount of physical RAM, the JDK gives the illusion of plentiful threads by mapping many virtual threads to a small number of OS threads.

And just as programs barely ever care about virtual versus physical memory, rarely does concurrent Java code have to care whether it runs in a virtual thread or a platform thread.

You can focus on writing straightforward, potentially blocking code—the runtime takes care of sharing the available OS threads to reduce the cost of blocking to near zero.

Virtual threads support thread-local variables, synchronized blocks, and thread interruption; therefore, code working with `Thread` and `currentThread` won't have to change. Indeed, this means that existing Java code will easily run in a virtual thread without any changes or even recompilation!

Once server frameworks offer the option to start a new virtual thread for every incoming request, all you need to do is update the framework and JDK and flip the switch.

Speed, scale, and structure

It's important to understand what virtual threads are for—and what they are not for.

Never forget that virtual threads aren't *faster* threads. Virtual threads don't magically execute more instructions per second than platform threads do.

What virtual threads are really good for is *waiting*.

Because virtual threads don't require or block an OS thread, potentially millions of virtual threads can wait patiently for requests to the file system, databases, or web services to finish.

By maximizing the utilization of external resources, virtual threads provide larger scale, not more speed. In other words, they improve *throughput*.

Beyond hard numbers, virtual threads can also improve code quality

Beyond hard numbers, virtual threads can also improve code quality.

Their cheapness opens the door to a fairly new concurrent programming paradigm called *structured concurrency*, which I covered in *Inside Java Newscast #17*.

It’s now time to explain how virtual threads work.

Scheduling and memory

The operating system schedules OS threads, and thus platform threads, but virtual threads are scheduled by the JDK. The JDK does so indirectly by assigning virtual threads to platform threads in a process called *mounting*. The JDK unassigns the platform threads later; this is called *unmounting*.

The platform thread running a virtual thread is called its *carrier thread* and from the perspective of Java code, the fact that a virtual and its carrier temporarily share an OS thread is invisible. For example, stack traces and thread-local variables are fully separated.

Carrier threads are then left to the OS to schedule as usual; as far as the OS is concerned, the carrier thread is simply a platform thread.

To implement this process, the JDK uses a dedicated `ForkJoinPool` in first-in-first-out (FIFO) mode as a virtual thread scheduler. (Note: This is distinct from the common pool used by parallel streams.)

By default, the JDK’s scheduler uses as many platform threads as there are available processor cores, but that behavior can be tuned with a system property.

Where do the stack frames of unmounted virtual threads go? They are stored on the heap as *stack chunk objects*.

Some virtual threads will have deep call stacks (such as a request handler called from a web framework), but those spawned by them will usually be much shallower (such as a method that reads from a file).

The JDK could mount a virtual thread by copying all its frames from heap to stack. When the virtual thread is unmounted, most frames are left on the heap and copied lazily as needed.

Thus, stacks grow and shrink as the application runs. This is crucial to making virtual threads cheap enough to have so many and to frequently switch between them. And there’s a good chance that future work can further reduce memory requirements.

Blocking and unmounting

Typically, a virtual thread will unmount when it blocks on I/O (for example, when it reads from a socket) or when it calls other blocking operations in the JDK (such as `take` on a `BlockingQueue`).

When the blocking operation is ready to finish (the socket received the bytes or the queue can hand out an element), the operation submits the virtual thread back to the scheduler, which will, in FIFO order, eventually mount it to resume execution.

However, despite prior work in JDK Enhancement Proposals such as JEP 353 (Reimplement the legacy `Socket` API) and JEP 373 (Reimplement the legacy `DatagramSocket` API), not *all* blocking operations in the JDK unmount the virtual thread. Instead, some *capture* the carrier thread and the underlying OS platform thread, thus blocking both.

This unfortunate behavior can be caused by limitations at the OS level (which affects many file system operations) or at the JDK level (such as with the `Object.wait()` call).

The capture of an OS thread is compensated by temporarily adding a platform thread to the scheduler, which can hence occasionally exceed the number of available processors; a maximum can be specified with a system property.

Unfortunately, there's one more imperfection in the initial virtual thread proposal: When a virtual thread executes a native method or a foreign function or it executes code inside a synchronized block or method, the virtual thread will be *pinned* to its carrier thread. A pinned thread will not unmount in situations where it otherwise would.

No platform thread is added to the scheduler in this situation, though, because there are a few things you can do to minimize the impact of pinning—more on that in a minute.

That means capturing operations and pinned threads will reintroduce platform threads that are waiting for something to finish. This doesn't make an application incorrect, but it might hinder its scalability.

Fortunately, future work may make synchronization nonpinning. And, refactoring internals of the `java.io` package and implementing OS-level APIs such as `io_uring` on Linux may reduce the number of capturing operations.

Virtual thread observability

Virtual threads are fully integrated with existing tools used to observe, analyze, troubleshoot, and optimize Java applications. For example, the Java Flight Recorder (JFR) can emit events when a virtual thread starts or ends, didn't start for some reason, or blocks while being pinned.

To see the latter situation more prominently, you can configure the runtime, via a system property, to print a stack trace when a thread blocks while pinned. The stack trace highlights stack frames that cause the pinning.

And because virtual threads are simply threads, debuggers can step through them just as they can through platform threads. Of course, those debugger user interfaces might need to be updated to deal with millions of threads, or you'll get some very tiny scroll bars!

Virtual threads naturally organize themselves in a hierarchy. That behavior and their sheer number make the flat format of traditional thread dumps unsuitable, though, so they will stick to just dumping platform threads. A new kind of thread dump in `jcmd` will present virtual threads alongside platform threads, all grouped in a meaningful way, in both plain text and JSON.

Three pieces of practical advice

The first item on my list: *Don't pool virtual threads!*

Pooling makes sense only for expensive resources and virtual threads aren't expensive. Instead, create new virtual threads whenever you need to do stuff concurrently. You might use thread pools to limit access to certain resources, such as requests to a database. Don't. Instead, use semaphores to make sure access to certain resources, such as requests to a database. Don't. Instead, use semaphores to make sure only a specified number of threads are accessing that resource, as follows:

```
// WITH THREAD POOL
private static final ExecutorService
  DB_POOL = Executors.newFixedThreadPool(16);

public <T> Future<T> queryDatabase(
    Callable<T> query) {
  // pool limits to 16 concurrent queries
  return DB_POOL.submit(query);
}


// WITH SEMAPHORE
private static final Semaphore
  DB_SEMAPHORE = new Semaphore(16);

public <T> T queryDatabase(
    Callable<T> query) throws Exception {
  // semaphore limits to 16 concurrent queries
  DB_SEMAPHORE.acquire();
  try {
    return query.call();
  } finally {
    DB_SEMAPHORE.release();
  }
}
```

Copy code snippet

Next, for good scalability with virtual threads, avoid frequent and long-lived pinning by revising synchronized blocks and methods that run often and contain I/O operations, particularly long-running ones. In this case, a good alternative to synchronization is a `ReentrantLock`, as follows:

```
// with synchronization (pinning 👎):
// synchronized guarantees sequential access
public synchronized String accessResource() {
  return access();
}


// with ReentrantLock (not pinning 👍):
private static final ReentrantLock
  LOCK = new ReentrantLock();
```

```
public String accessResource() {
  // lock guarantees sequential access
  LOCK.lock();
  try {
    return access();
  } finally {
    LOCK.unlock();
  }
}
```

Copy code snippet

Finally, another aspect that works correctly in virtual threads but deserves being revisited for better scalability is *thread-local variables*, both regular and inheritable. Virtual threads support thread-local behavior the same way as platform threads, but because virtual threads can be very numerous, thread locals should be used only after careful consideration.

In fact, as part of Project Loom, many uses of thread locals in the `java.base` module were removed to reduce the memory footprint when code is running with millions of threads. There is an interesting alternative for some use cases that is currently being explored in a draft JEP for scope-local variables.

[This article was adapted from Nicolai Parlog's *Inside Java Newscast* #23, "Virtual thread deep dive." — *Ed.*]

Dig deeper

- On parallelism and concurrency
- Going inside Java's Project Loom and virtual threads
- Amber, Lambda, Loom, Panama, Valhalla: The major named Java projects
- Java's evolution into 2022: The state of the four big initiatives

Nicolai Parlog
Developer Advocate, Oracle

Nicolai Parlog (@nipafx) is a Java enthusiast with a passion for learning and sharing, online and offline. He is a developer advocate at Oracle.

Previous Post

Next Post