Java Platform Threads vs. Virtual Threads (Part 1)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Know the differences between Java platform & virtual threads

---

**Platform threads**

`Thread` supports the creation of *platform threads* that are typically mapped 1:1 to kernel threads scheduled by the operating system. Platform threads will usually have a large stack and other resources that are maintained by the operating system. Platforms threads are suitable for executing all types of tasks but may be a limited resource.

Platform threads are designated *daemon* or *non-daemon* threads. When the Java virtual machine starts up, there is usually one non-daemon thread (the thread that typically calls the application's `main` method). The Java virtual machine terminates when all started non-daemon threads have terminated. Unstarted daemon threads do not prevent the Java virtual machine from terminating. The Java virtual machine can also be terminated by invoking the `Runtime.exit(int)` method, in which case it will terminate even if there are non-daemon threads still running.

In addition to the daemon status, platform threads have a thread priority and are members of a thread group.

Platform threads get an automatically generated thread name by default.

---

**Virtual threads**

`Thread` also supports the creation of *virtual threads*. Virtual threads are typically *user-mode threads* scheduled by the Java virtual machine rather than the operating system. Virtual threads will typically require few resources and a single Java virtual machine may support millions of virtual threads. Virtual threads are suitable for executing tasks that spend most of the time blocked, often waiting for I/O operations to complete. Virtual threads are not intended for long running CPU intensive operations.

Virtual threads typically employ a small set of platform threads used as *carrier threads*. Locking and I/O operations are the *scheduling points* where a carrier thread is re-scheduled from one virtual thread to another. Code executing in a virtual thread will usually not be aware of the underlying carrier thread, and in particular, the `currentThread()` method, to obtain a reference to the `current thread`, will return the Thread object for the virtual thread, not the underlying carrier thread.

Virtual threads gets a fixed name by default.

---

See [docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Thread.html](docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Thread.html)
Java Platform Threads vs. Virtual Threads
A Java Thread has traditionally been an object containing various methods & fields that constitute its “state”

e.g., each Java Thread has its own unique name, identifier, priority, runtime stack, thread-local storage, instruction pointer, & other registers, etc.

See blog.jamesdbloom.com/JVMInternals.html
Java Platform Threads vs. Virtual Threads

• A Java Thread has traditionally been an object containing various methods & fields that constitute its “state”
• Java 19 now refers to these types of Java threads as “platform threads”

See wiki.openjdk.java.net/display/loom/Main
Java Platform Threads vs. Virtual Threads

• Each Java platform thread is associated 1-to-1 with an OS kernel thread

See en.wikipedia.org/wiki/Thread_(computing)#Kernel_threads
Java Platform Threads vs. Virtual Threads

• Each Java platform thread is associated 1-to-1 with an OS kernel thread
• It contains the same unique “state” as a traditional Java Thread object
Java Platform Threads vs. Virtual Threads

- Each Java platform thread is associated 1-to-1 with an OS kernel thread
  - It contains the same unique “state” as a traditional Java Thread object
- Platforms threads are suitable for executing all types of tasks
Java Platform Threads vs. Virtual Threads

- Each Java platform thread is associated 1-to-1 with an OS kernel thread
  - It contains the same unique “state” as a traditional Java Thread object
- Platforms threads are suitable for executing all types of tasks
  - However, they are a limited resource due to their non-trivial runtime stack size
Java Platform Threads vs. Virtual Threads

• In contrast, each Java virtual thread is a “lightweight” concurrency object

See [www.infoq.com/articles/java-virtual-threads](http://www.infoq.com/articles/java-virtual-threads)
Java Platform Threads vs. Virtual Threads

- In contrast, each Java virtual thread is a “lightweight” concurrency object
- It is a user thread rather than a kernel thread

See en.wikipedia.org/wiki/Thread_(computing)#User_threads
Java Platform Threads vs. Virtual Threads

• In contrast, each Java virtual thread is a “lightweight” concurrency object
• It is a user thread rather than a kernel thread
  • It is scheduled by the Java execution environment rather than the underlying OS
Java Platform Threads vs. Virtual Threads

- In contrast, each Java virtual thread is a “lightweight” concurrency object.
- It is a user thread rather than a kernel thread.
  - It is scheduled by the Java execution environment rather than the underlying OS.
- A very large number of virtual threads can therefore be created.

See www.youtube.com/watch?v=Ul50FFmOzU4
Java Platform Threads vs. Virtual Threads

- In contrast, each Java virtual thread is a “lightweight” concurrency object
  - It is a user thread rather than a kernel thread
- Virtual threads are multiplexed atop a pool of “carrier” threads

Blocking operations no longer block the executing thread, which enables the processing of a large # of requests in parallel with a small pool of carrier threads

See [www.happycoders.eu/java/virtual-threads](www.happycoders.eu/java/virtual-threads)
Java Platform Threads vs. Virtual Threads

- In contrast, each Java virtual thread is a “lightweight” concurrency object
  - It is a user thread rather than a kernel thread
- Virtual threads are multiplexed atop a pool of “carrier” threads
  - The Java fork-join framework is currently used to implement the “carrier” threads

See theboreddev.com/understanding-java-virtual-threads
End of Java Platform

Threads vs. Virtual Threads

(Part 1)