Threading Models in Java

Last updated: January 8, 2024


Written by:
Graham Cox


Reviewed by:
Grzegorz Piwowarek

Java Concurrency

Threads

1. Introduction

Often in our applications, we need to be able to do multiple things at the same time. We can achieve this in several ways, but key amongst them is to implement multitasking in some form.

Multi-tasking means running multiple tasks at the same time, where each task is performing its work. These tasks typically all run at the same time, reading and writing the same memory and interacting with the same resources, but doing different things.

2. Native Threads

The standard way of implementing multi-tasking in Java is to use threads. Threading is usually supported down to the operating system. We call threads that work at this level "native threads".

The operating system has some abilities with threading that are often unavailable to our applications, simply because of how much closer it is to the underlying hardware. This means that executing native threads are typically more efficient. These threads directly map to threads of execution on the computer CPU – and the operating system manages the mapping of threads onto CPU cores.

The standard threading model in Java, covering all JVM languages, uses native threads. This has been the case since Java 1.2 and is the case regardless of the underlying system that the JVM is running on.

This means that any time we use any of the standard threading mechanisms in Java, then we're using native threads. This includes *java.lang.Thread*, *java.util.concurrent.Executor*, *java.util.concurrent.ExecutorService*, and so on.

3. Green Threads

In software engineering, **one alternative to native threads is green threads**. This is where we are using threads, but they do not directly map to operating system threads. Instead, the underlying architecture manages the threads itself and manages how these map on to operating system threads.
Typically this works by running several native threads and then allocating the green threads onto these native threads for execution. The system can then choose which green threads are active at any given time, and which native threads they are active on.

This sounds very complicated, and it is. But it's a complication that we generally don't need to care about. The underlying architecture takes care of all of this, and we get to use it as if it was a native threading model.

So why would we do this? Native threads are very efficient to run, but they have a high cost around starting and stopping them. Green threads help to avoid this cost and give the architecture a lot more flexibility. If we are using relatively long-running threads, then native threads are very efficient. **For very short-lived jobs, the cost of starting them can outweigh the benefit of using them**. In these cases, green threads can become more efficient.

Unfortunately, **Java does not have built-in support for green threads.**

Very early versions used green threads instead of native threads as the standard threading model. This changed in Java 1.2, and there has not been any support for it at the JVM level since.

It's also challenging to implement green threads in libraries because they would need very low-level support to perform well. As such, a common alternative used is fibers.

4. Fibers

Fibers are an alternative form of multi-threading and are similar to green threads. In both cases, we aren't using native threads and instead are using the underlying system controls which are running at any time. The big difference between green threads and fibers is in the level of control, and specifically who is in control.

Green threads are a form of preemptive multitasking. This means that the underlying architecture is entirely responsible for deciding which threads are executing at any given time.

responsible for deciding which threads are executing at any given time.

This means that all of the usual issues of threading apply, where we don't know anything about the order of our threads executing, or which ones will be executing at the same time. It also means that the underlying system needs to be able to pause and restart our code at any time, potentially in the middle of a method or even a statement.

Fibers are instead a form of cooperative multitasking, meaning that a running thread will continue to run until it signals that it can yield to another. It means that it is our responsibility for the fibers to co-operate with each other. This puts us in direct control over when the fibers can pause execution, instead of the system deciding this for us.

This also means we need to write our code in a way that allows for this. Otherwise, it won't work. If our code doesn't have any interruption points, then we might as well not be using fibers at all.

Java does not currently have built-in support for fibers. Some libraries exist that can introduce this to our applications, including but not limited to:

4.1. Quasar

Quasar is a Java library that works well with pure Java and Kotlin and has an alternative version that works with Clojure.

It works by having a Java agent that needs to run alongside the application, and this agent is responsible for managing the fibers and ensuring that they work together correctly. The use of a Java agent means that there are no special build steps needed.

Quasar also requires Java 11 to work correctly so that might limit the applications that can use it. Older versions can be used on Java 8, but these are not actively supported.

4.2. Kilim

Kilim is a Java library that offers very similar functionality to Quasar but does so by using bytecode weaving instead of a Java agent. This means that it can work in more places, but it makes the build process more complicated.

Kilim works with Java 7 and newer and will work correctly even in scenarios where a Java agent is not an option. For example, if a different one is already used for instrumentation or monitoring.

4.3. Project Loom

Project Loom is an experiment by the OpenJDK project to add fibers to the JVM itself, rather than as an add-on library. This will give us the advantages of fibers over threads. By implementing it on the JVM directly, it can help to avoid complications that Java agents and bytecode weaving introduce.

There is no current release schedule for Project Loom, but we can download early access binaries right now to see how things are going. However, because it is still very early, we need to be careful relying on this for any production code.

5. Co-Routines

Co-routines are an alternative to threading and fibers. **We can think of co-routines as fibers without any form of scheduling**. Instead of the underlying system deciding which tasks are performing at any time, our code does this directly.

Generally, we write co-routines so that they yield at specific points of their flow. These can be seen as pause points in our function, where it will stop working and potentially output some intermediate result. When we do yield, we are then stopped until the calling code decides to re-start us for whatever reason. **This means that our calling code controls the scheduling of when this will run.**

Kotlin has native support for co-routines built into its standard library. There are several other Java libraries that we can use to implement them as well if desired.

6. Conclusion

We've seen several different alternatives for multi-tasking in our code, ranging from the traditional native threads to some very light-weight alternatives. Why not try them out next time an application needs concurrency?