Virtual Thread Debugging Support

***This page is obsolete. Use Debugger Support instead.

Overview

Support for debugging Fibers has been partially implemented. Fibers will appear to the debugger as ordinary threads. When you view the list of threads from within the debugger, you will see both fibers and the carrier threads they run on. Depending on the debugging mode, not all fibers may appear in the thread list. See Fiber Debugging Modes below.

If a fiber is mounted, both the fiber and its carrier thread are basically the same execution context. If one is at a breakpoint, they both will appear to be at the same breakpoint. They will have the same stack traces, and single stepping in one advances the PC in the other. They are essentially two different views of the same thread (with some minor exceptions in debugger behavior noted below).

You can tell which carrier thread a fiber is mounted on by the name of the fiber, which will be `fiber@<hashcode>[<carrier_thread_name>]`. If it is not mounted, you will instead see `fiber@<hashcode>[<no carrier thread>]`. Here is an example debugger threads list with 2 carrier threads and 3 fibers, two of which are mounted:

```
ForkJoinPool-1-worker-3
ForkJoinPool-1-worker-1
Fiber@5a1fbaa6[<no carrier thread>]
Fiber@50e7d1d6[ForkJoinPool-1-worker-3,CarrierThreads]
Fiber@5e2035ba[ForkJoinPool-1-worker-1,CarrierThreads]
```

Single Stepping

If you single step while in the context of a fiber, execution flow will remain in the fiber even if you step over a method call that results in a yield, causing the fiber to be unmounted and later remounted on a different carrier thread. If you are in the context of the carrier thread when the fiber yields, the behavior is not well defined yet, and there are also known issues, especially when single stepping in and around `Continuation.yield()`.

Fiber Stack Traces

Stack traces for fibers will include any scheduler related frames. For example, only the top three frames below belong to the Fiber's implementation. The rest are considered to be scheduler related frames:

```
[1] Main.taker (Main.java:29)
[2] Main.lambda$static$2 (Main.java:35)
[3] Main$$Lambda$7.1705736037.run (null)
[4] java.lang.Fiber.lambda$new$0 (Fiber.java:161)
[5] java.lang.Fiber$$Lambda$9.250421012.run (null)
[6] java.lang.Continuation.enter0 (Continuation.java:233)
[7] java.lang.Continuation.enter (Continuation.java:220)
[8] java.lang.Continuation.run (Continuation.java:178)
[9] java.lang.Fiber.runContinuation (Fiber.java:309)
[10] java.lang.Fiber$$Lambda$10.295530567.run (null)
[11] java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec (ForkJoinTask.java:1,425)
[12] java.util.concurrent.ForkJoinTask.doExec (ForkJoinTask.java:290)
[13] java.util.concurrent.ForkJoinPool$WorkQueue.topLevelExec (ForkJoinPool.java:1,020)
[14] java.util.concurrent.ForkJoinPool.scan (ForkJoinPool.java:1,656)
[15] java.util.concurrent.ForkJoinPool.runWorker (ForkJoinPool.java:1,594)
[16] java.util.concurrent.ForkJoinWorkerThread.run (ForkJoinWorkerThread.java:189)
```

If the fiber is unmounted, currently the stack trace is a bit confusing because the fiber will temporarily be mounted on a suspended helper thread. So you get something like:

```
[1] java.lang.Thread.suspend0 (native method)
[2] java.lang.Thread.suspendThread (Thread.java:1,229)
[3] java.lang.Fiber.lambda$tryMountAndSuspend$3 (Fiber.java:1,225)
[4] java.lang.Fiber$$Lambda$14.660017696.run (null)
[5] java.lang.Fiber.maybePark (Fiber.java:529)
[6] java.lang.Fiber.parkNanos (Fiber.java:488)
[7] java.lang.Thread.sleep (Thread.java:364)
[8] Main.taker (Main.java:27)
[9] Main.lambda$static$2 (Main.java:35)
[10] Main$$Lambda$7.1705736037.run (null)
[11] java.lang.Fiber.lambda$new$0 (Fiber.java:161)
[12] java.lang.Fiber$$Lambda$9.250421012.run (null)
[13] java.lang.Continuation.enter0 (Continuation.java:233)
[14] java.lang.Continuation.enter (Continuation.java:220)
[15] java.lang.Continuation.run (Continuation.java:178)
[16] java.lang.Fiber.tryRun (Fiber.java:1,173)
[17] java.lang.Fiber.lambda$tryMountAndSuspend$4 (Fiber.java:1,223)
[18] java.lang.Fiber$$Lambda$13.153425756.run (null)
[19] java.lang.Thread.run (Thread.java:935)
[20] jdk.internal.misc.InnocuousThread.run (InnocuousThread.java:134)
```

In this case frames 7-10 (and actually also 5-6) are considered fiber frames. The frames above it are just there to support temporarily mounting the fiber. This will be improved in the future.

Debugging Support Limitations

There are a number of debugger features that are known to not work (or likely won't work) if you try them. They include (but are probably not limited to):

- Suspending or resuming a specific fiber. When the debugger asks for all threads to be suspended (which is usually what happens at breakpoints), the fibers will also be considered suspended. However, do not try to suspend a specific fiber, or resume any fiber that was suspended as a result of suspending all threads.
- Setting local variables in fibers (you can view them). This actually applies to fiber frames, even if the current thread is a carrier thread. More specifically, you cannot set locals in continuation frames.
- Dealing with a large number of fibers. For one, you will overwhelm the debugger because it will see them all. Also, fibers are not tracked in an efficient manner by the debug agent, so it can also become bogged down.
- Forcing a fiber frame to return ("Drop Frame" in Intellij), although this should work fine if executed from the carrier thread context instead of the fiber context.

Fiber Debugging Modes

By default the debugger is only notified of fibers for which certain events have arrived on (breakpoint, exception, and watchpoint being the most notable). This is necessary when debugging a vary large number of fibers, because you wouldn't want every single fiber to appear as a thread in the debugger. This behavior can be overridden with the "fibers" option passed on the command line to the debug agent.

- `fibers=all`: The debugger will see all fibers. Use only when you have a manageable number of fibers, and you don't mind all of them appearing in the debugger's thread list.
- `fibers=n`: The debugger will not see any fibers. Use this option when using a large number of fibers and not needing any fiber specific debugging support (like single stepping through a fiber as it switches carrier threads).
- `fibers=y`: This is the default. The debugger will only see fibers for which certain events have arrived on. This will be useful when using a large number of fibers, and only needing to debug some fibers. The debugger will become aware of a fiber when certain events arrive on it, such a breakpoint, watchpoint, or exception. Once one of these events arrives, you can then single step through the fiber code, even as it switches carrier threads.

Miscellaneous Implementation Details

For this first version of Fiber debugger support, we decided not to make any changes the JDWP or JDI, and therefore no changes are needed in the debuggers. There are some JVMTI additions that provide the Debug Agent with events to indicate Fiber start/terminate and also mount/unmount. There are also new JVMTI APIs that allow getting the Fiber running on a Carrier Thread (if one is running), and to get the Carrier Thread a Fiber is mounted on (if it is mounted).

For JDWP, JDI, and debuggers, Fibers appear to be ordinary threads, and no changes have been made in these layers of the debugging support. The heavy lifting of Fiber debugger support is in the Debug Agent, which manages the mapping between a Fiber and its Carrier Thread. For example:

- If the Debug Agent gets a BREAKPOINT event from JVMTI on a carrier thread, it will deliver that event to the debugger on the Fiber instead (depending on the debugging mode as described in the Fiber Debugging Modes section above).
- If you single step in a Fiber, this results in the Debug Agent enabling single stepping on the Carrier Thread that the Fiber is running on. When the JVMTI delivers the SINGLE_STEP event to the Debug Agent on the Carrier Thread, it is relayed to the debugger on the Fiber instead of the Carrier Thread.

In the future, we may decide to allow debuggers to be Fiber aware, possibly allowing for improved fiber debugging support. This would also require changes to JDI, JDWP, the Debug Agent, and JVMTI. We expect these design decisions to be driven by a combination of debugger user needs and what debugger vendors are willing to implement support for.