Java Platform Threads vs. Virtual Threads (Part 1)

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

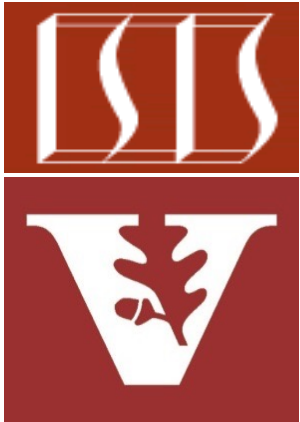
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software Integrated Systems

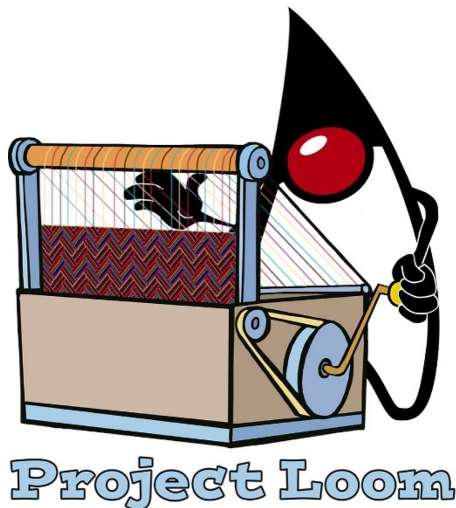
Vanderbilt University

Nashville, Tennessee, USA



Learning Objectives in this 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 get an automatically generated thread name by default.

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 shutdown sequence begins when all started non-daemon threads have terminated. Unstarted non-daemon threads do not prevent the shutdown sequence from beginning.

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

Virtual threads

`Thread` also supports the creation of *virtual threads*. Virtual threads are typically *user-mode threads* scheduled by the Java runtime 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 examples of operations where a carrier thread may be re-scheduled from one virtual thread to another. Code executing in a virtual thread is not aware of the underlying carrier thread. The currentThread() method, used to obtain a reference to the *current thread*, will always return the `Thread` object for the virtual thread.

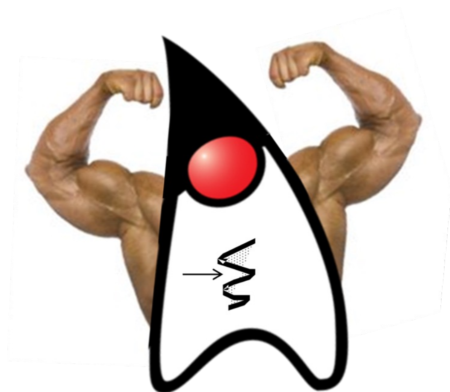
Virtual threads do not have a thread name by default. The `getName` method returns the empty string if a thread name is not set.

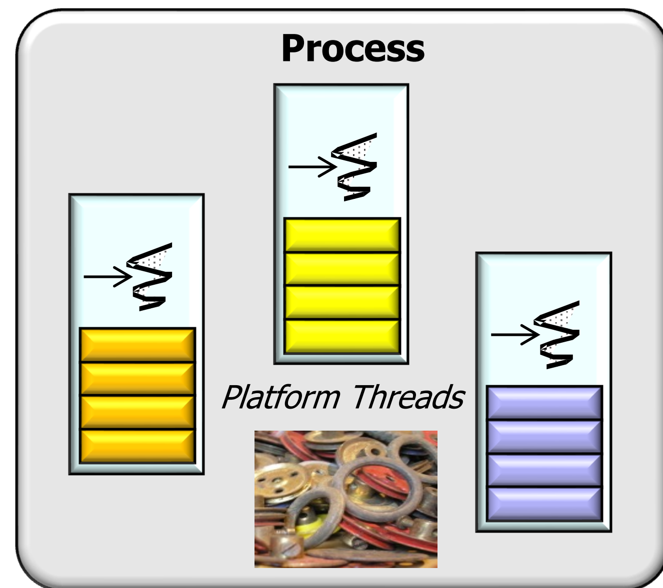
Virtual threads are daemon threads and so do not prevent the shutdown sequence from beginning. Virtual threads have a fixed thread priority that cannot be changed.

See docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/Thread.html

Learning Objectives in this Lesson

- Know the differences between Java platform & virtual threads
  - Platform threads are typically mapped 1-to-1 onto kernel-mode threads scheduled by the OS

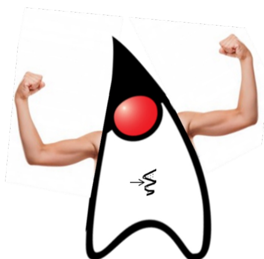


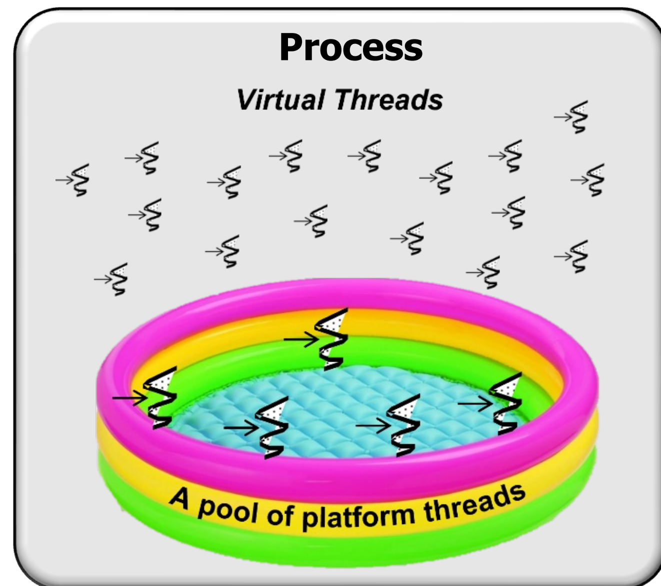


See medium.com/@a_gegov/java-19-virtual-threads-vs-platform-threads-ff0452ca1204

Learning Objectives in this Lesson

- Know the differences between Java platform & virtual threads
  - Platform threads are typically mapped 1-to-1 onto kernel-mode threads scheduled by the OS
  - Virtual threads are typically user-mode threads scheduled by the Java execution environment



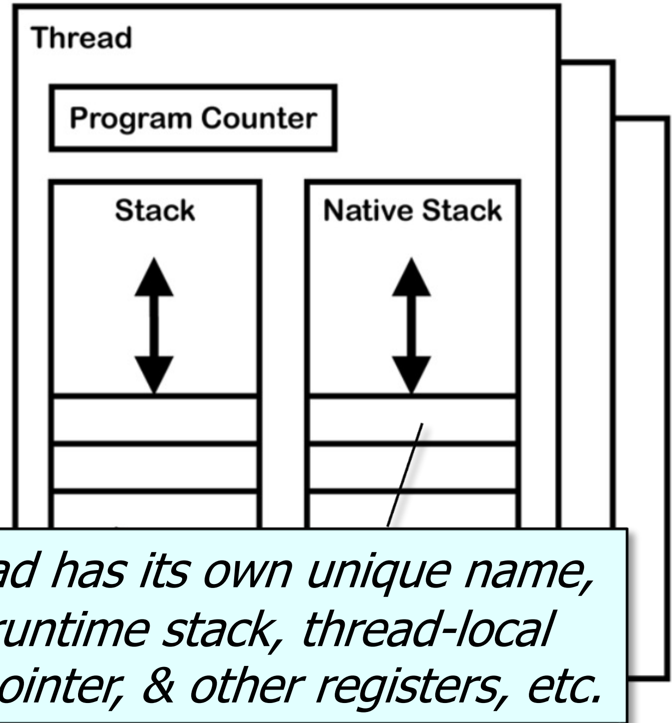


See howtodoinjava.com/java/multi-threading/virtual-threads

Overview of Java Platform Threads

Overview of Java Platform Threads

- A Java Thread has traditionally been an object containing various methods & fields that constitute its "state"

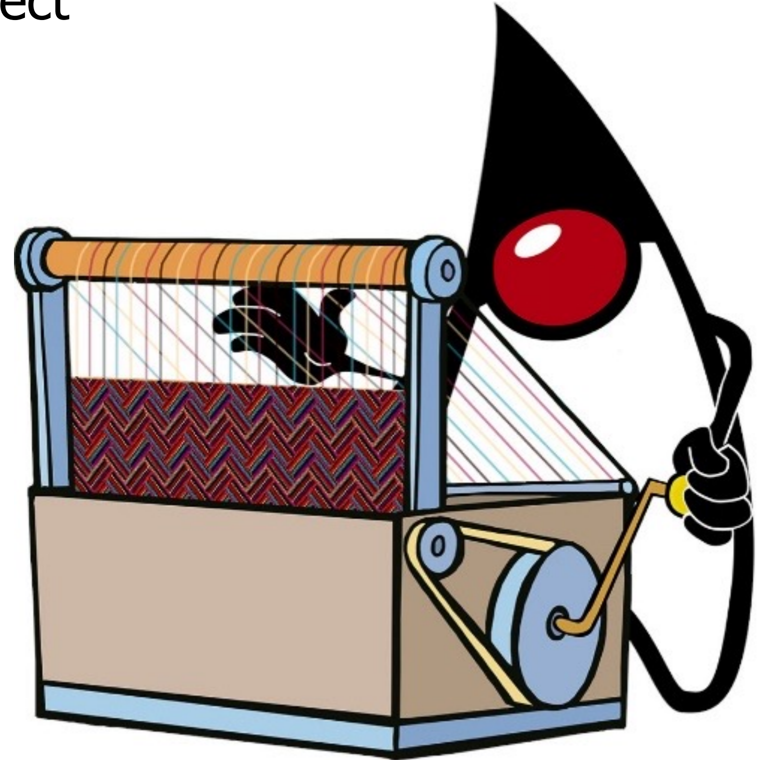


e.g., each Java Thread has its own unique name, identifier, priority, runtime stack, thread-local storage, instruction pointer, & other registers, etc.

See blog.jamesdbloom.com/JVMInternals.html

Overview of Java Platform Threads

- A Java Thread has traditionally been an object containing various methods & fields that constitute its "state"
  - Very modern Java refers to these types of Java threads as "platform threads"



See wiki.openjdk.java.net/display/loom/Main

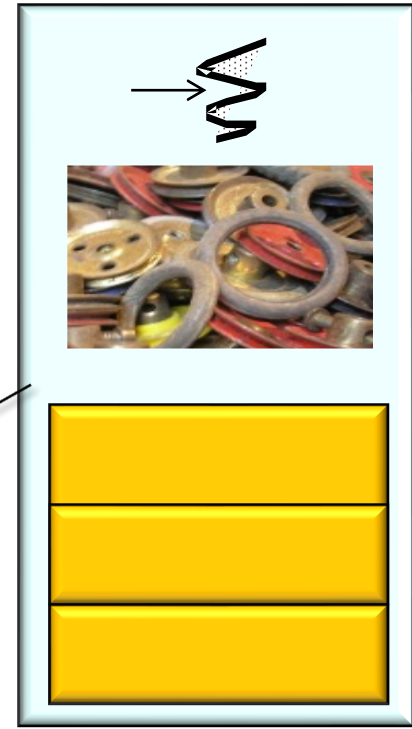
Overview of Java Platform Threads

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

See en.wikipedia.org/wiki/Thread_(computing)#Kernel_threads

Overview of Java Platform 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



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

Overview of Java Platform 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

Overview of Java Platform 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
    - Particularly I/O-bound tasks that block reading/writing on sockets or files

See github.com/douglascraigschmidt/ModernJava/tree/main/CS/ImageStreamGang

Overview of Java Platform 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

LIMITED

See docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/thread_basics.html

Overview of Java Platform Threads

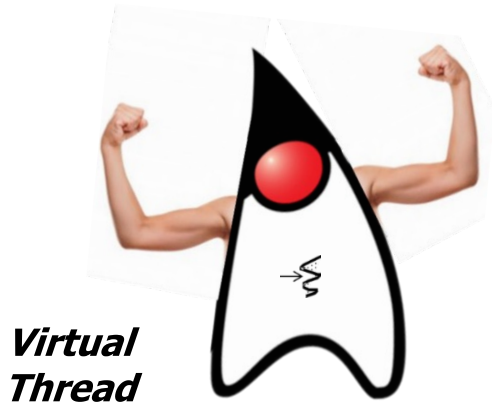
- Each Java platform thread is associated 1-to-1 with an OS kernel thread
  - It contains the same unique "state" as a traditional Java Thread object
  - Platforms threads are suitable for executing all types of tasks
  - However, they are a limited resource due to their non-trivial runtime stack size
    - They may also need to synchronize, which causes an expensive context switch to happen between OS Threads

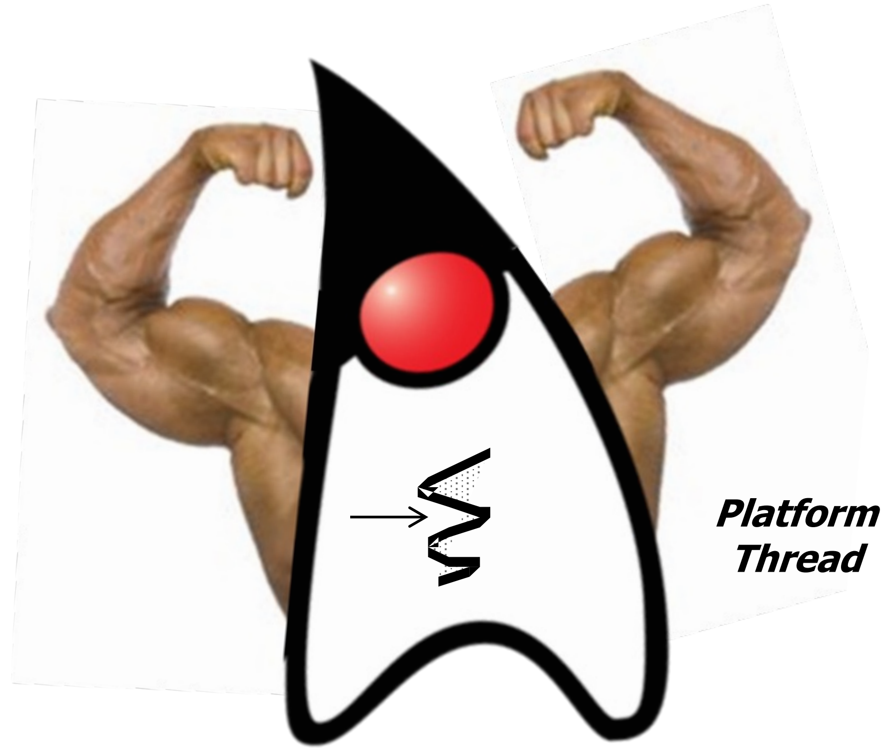
See en.wikipedia.org/wiki/Context_switch

Overview of Java Virtual Threads

Overview of Java Virtual Threads

- Each Java virtual thread is a "lightweight" concurrency object

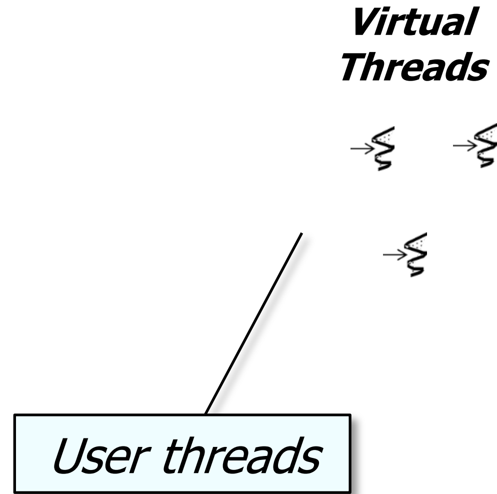




See www.infoq.com/articles/java-virtual-threads

Overview of Java Virtual Threads

- 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

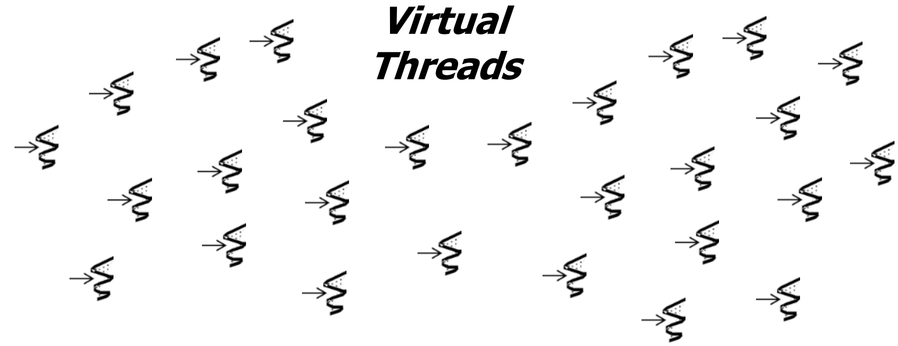
Overview of Java Virtual Threads

- 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

See openjdk.org/jeps/444

Overview of Java Virtual Threads

- 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 # of virtual threads can therefore be created

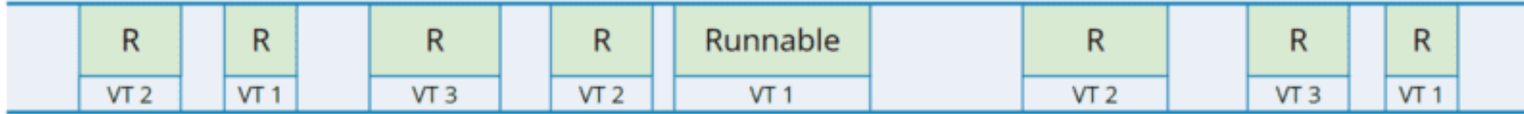


See www.youtube.com/watch?v=Ul50FFmOzU4

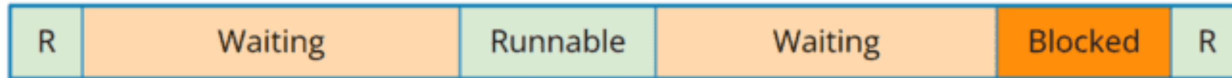
Overview of Java Virtual Threads

- Virtual threads are multiplexed atop a pool of "carrier" threads

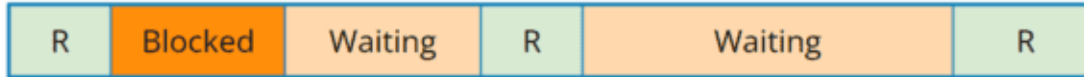
Carrier thread:



Virtual thread 1:



Virtual thread 2:



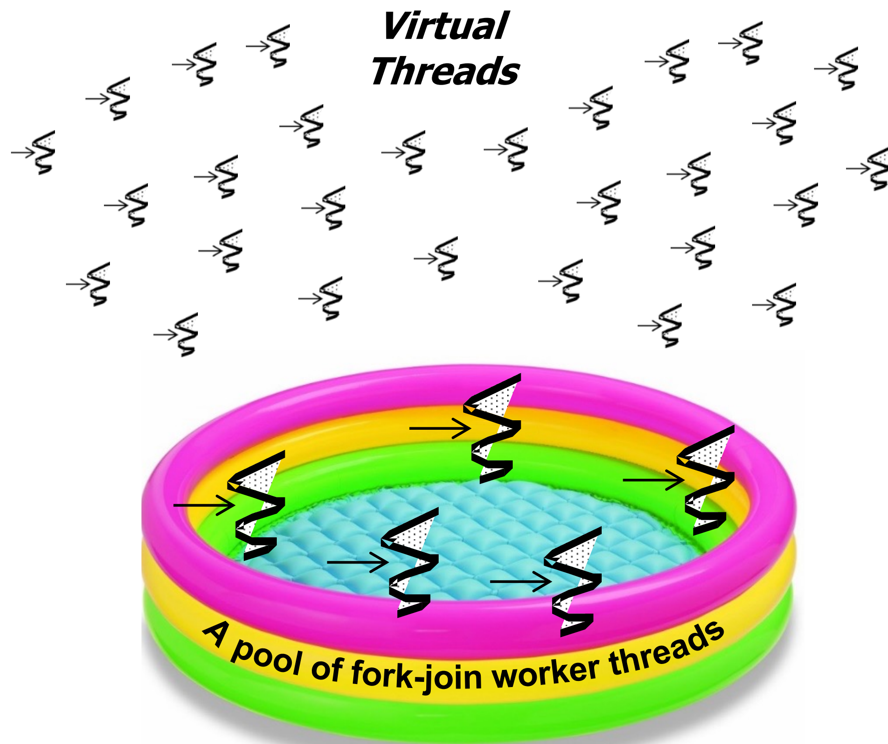
Virtual thread 3:

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

See www.happycoders.eu/java/virtual-threads

Overview of Java Virtual Threads

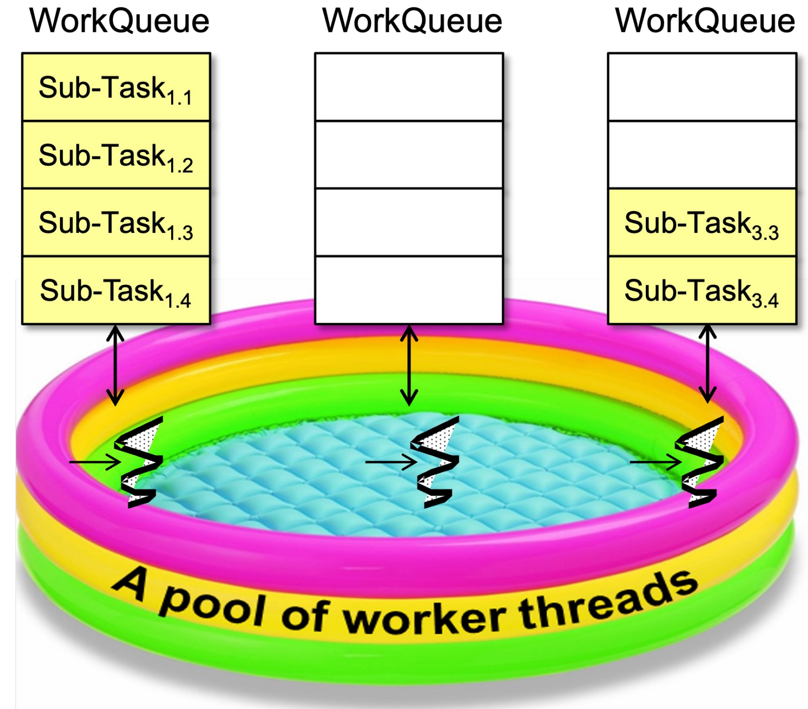
- Virtual threads are multiplexed atop a pool of "carrier" threads
  - The Java fork-join framework is currently used to implement these "carrier" threads



See theboreddev.com/understanding-java-virtual-threads

Overview of Java Virtual Threads

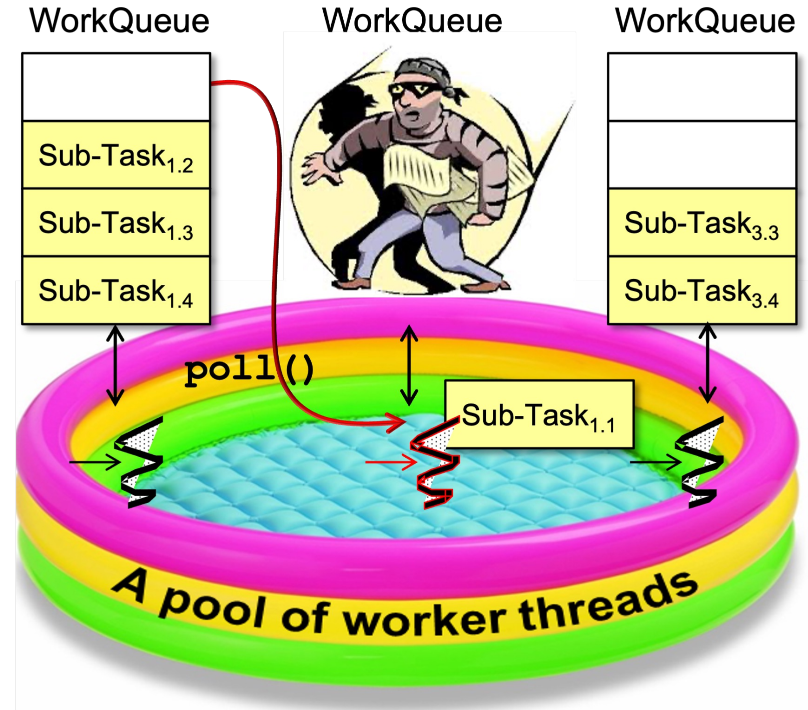
- Virtual threads are multiplexed atop a pool of "carrier" threads
  - The Java fork-join framework is currently used to implement these "carrier" threads
  - More info on the Java fork-join framework is available online



See www.dre.vanderbilt.edu/~schmidt/cs254

Overview of Java Virtual Threads

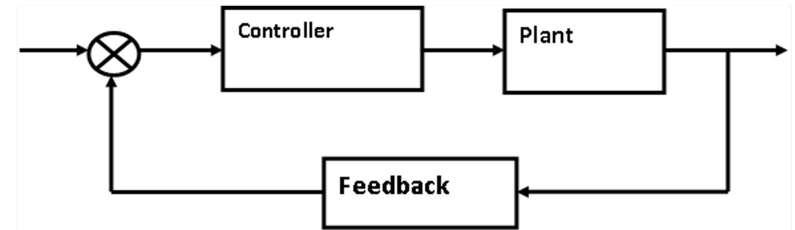
- Virtual threads are multiplexed atop a pool of "carrier" threads
  - The Java fork-join framework is currently used to implement these "carrier" threads
  - More info on the Java fork-join framework is available online
    - "Work-stealing"
      - Enables idle worker threads to "steal" work from busy threads



See en.wikipedia.org/wiki/Work_stealing

Overview of Java Virtual Threads

- Virtual threads are multiplexed atop a pool of "carrier" threads
  - The Java fork-join framework is currently used to implement these "carrier" threads
  - More info on the Java fork-join framework is available online
    - "Work-stealing"
    - Managed blocking
      - Helps avoid starvation & improve performance by adding new worker threads when existing ones block



See www.javaspecialists.eu/archive/Issue223-ManagedBlocker.html

End of Java Platform Threads vs. Virtual Threads (Part 1)