Using Virtual Threads to Increase Your Throughput

José Paumard

PhD, Java Champion, JavaOne Rockstar

@JosePaumard | https://github.com/JosePaumard

When and where to use virtual threads efficiently

Agenda

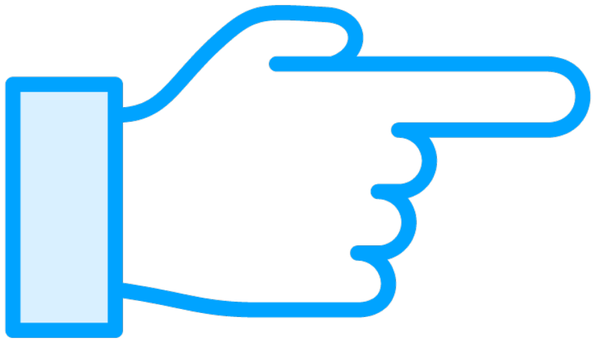
How virtual threads are working under the hood

What is happenning when your code blocks a virtual thread?

What does pinning a virtual thread mean?

How do reactive programming and virtual threads compare, performance-wise?

Launching Virtual Threads



Several ways to use virtual threads

Create them by hand

Use an ExecutorService

Supported by Spring Boot, Quarkus, Micronaut

Helidon was rewritten

More on Virtual Threads

Create a virtual thread

Using the factory method

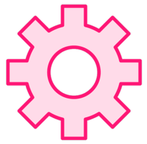
An see that it is built on a platform thread

Virtual Threads and Carrier Threads

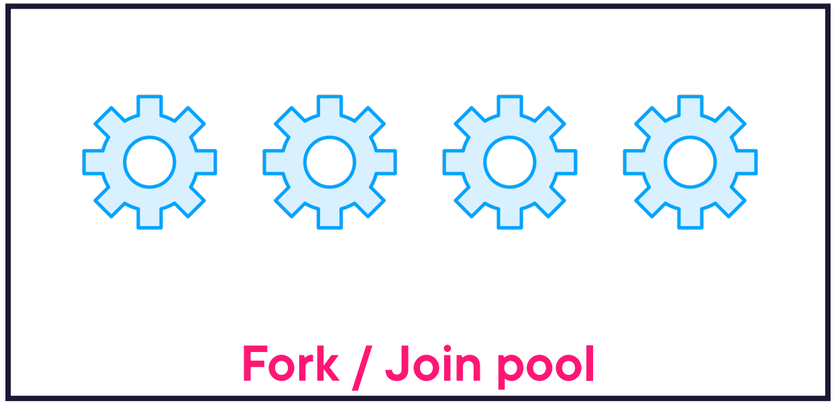
A virtual thread is executed
on a platform thread

This platform thread is taken from
a special Fork / Join pool

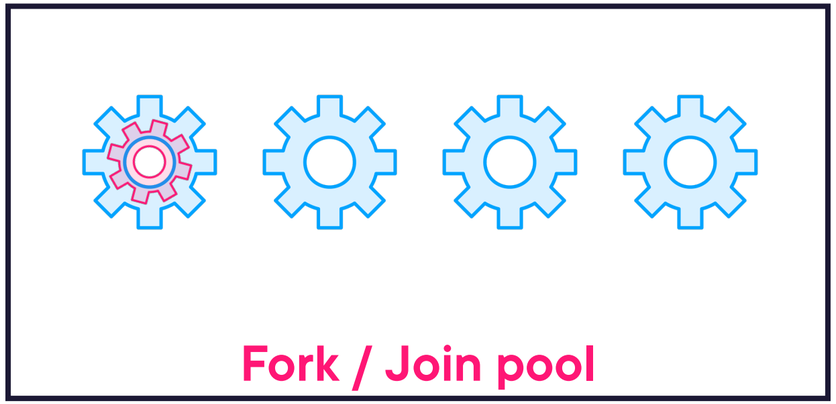
Running a Virtual Thread



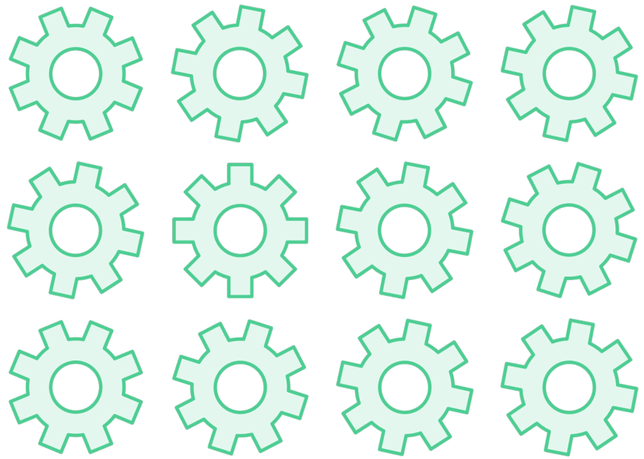
Virtual Thread

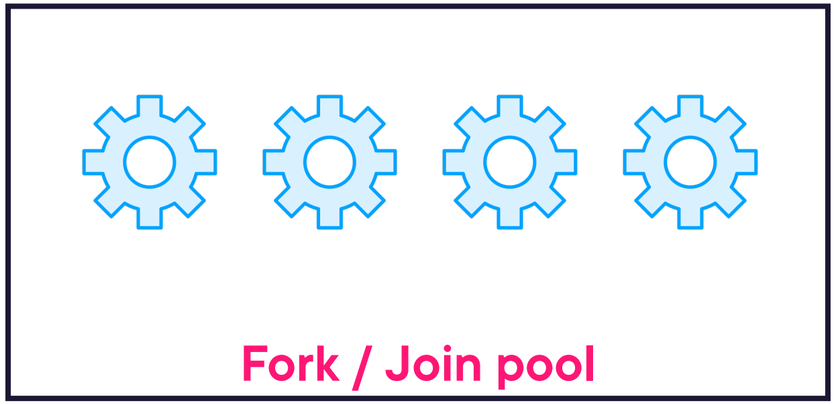


Running a Virtual Thread

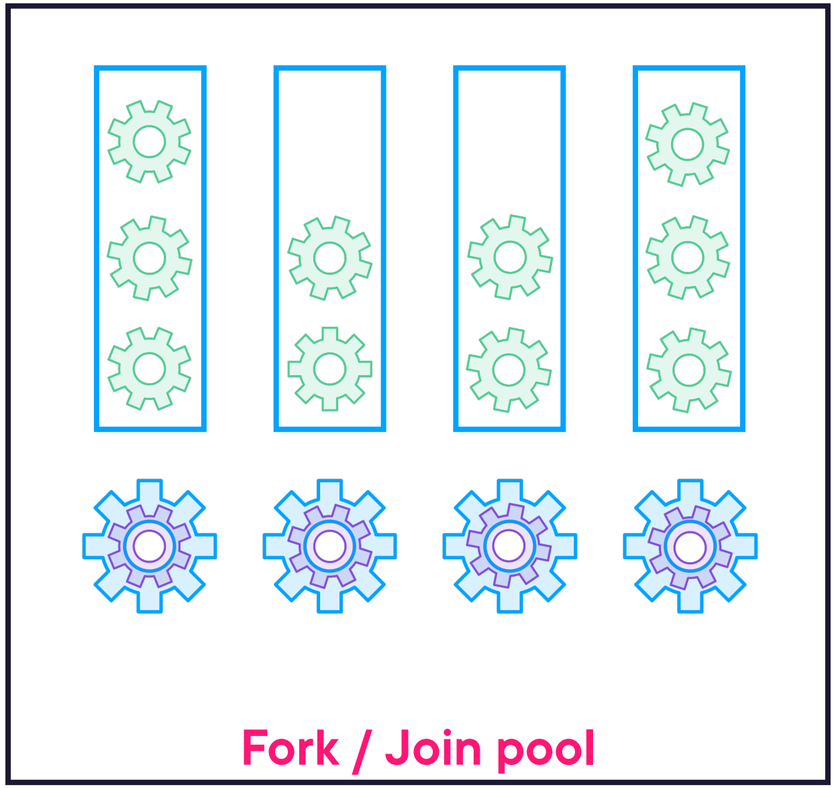


Running a Virtual Thread

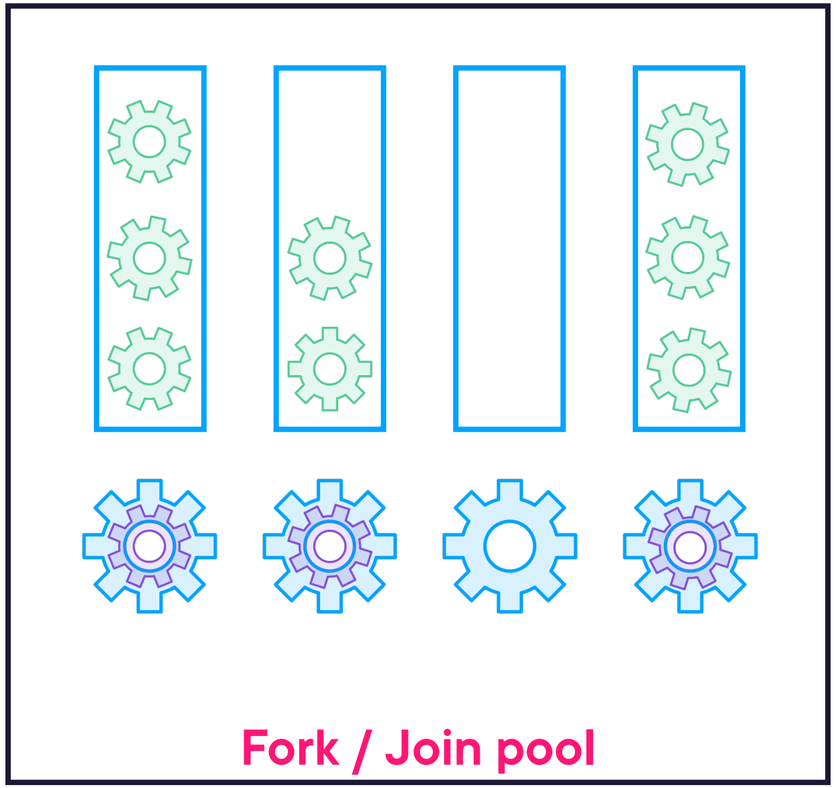




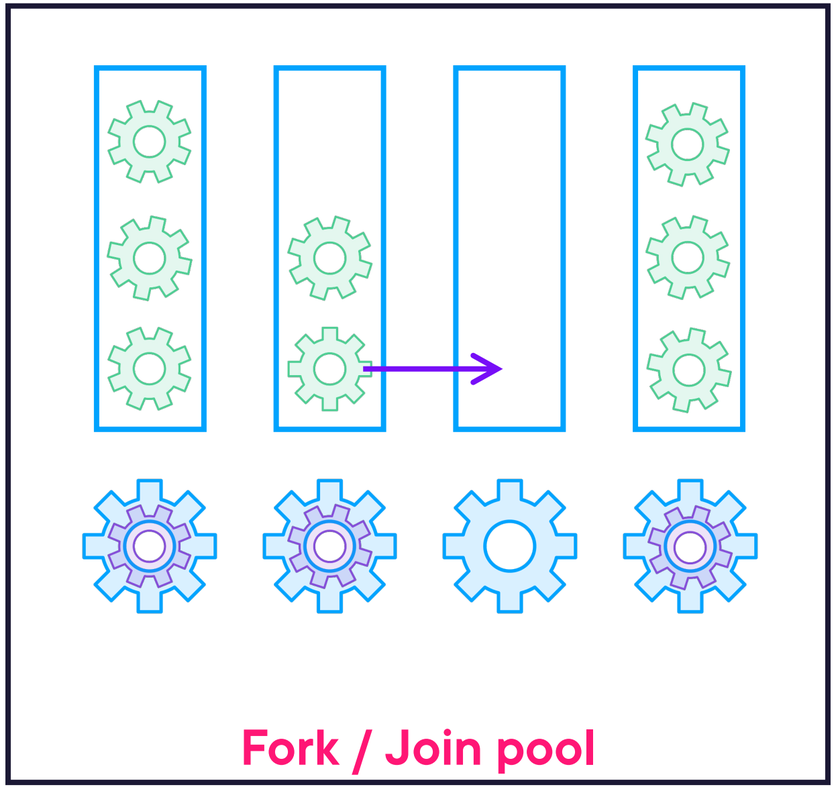
Running a Virtual Thread



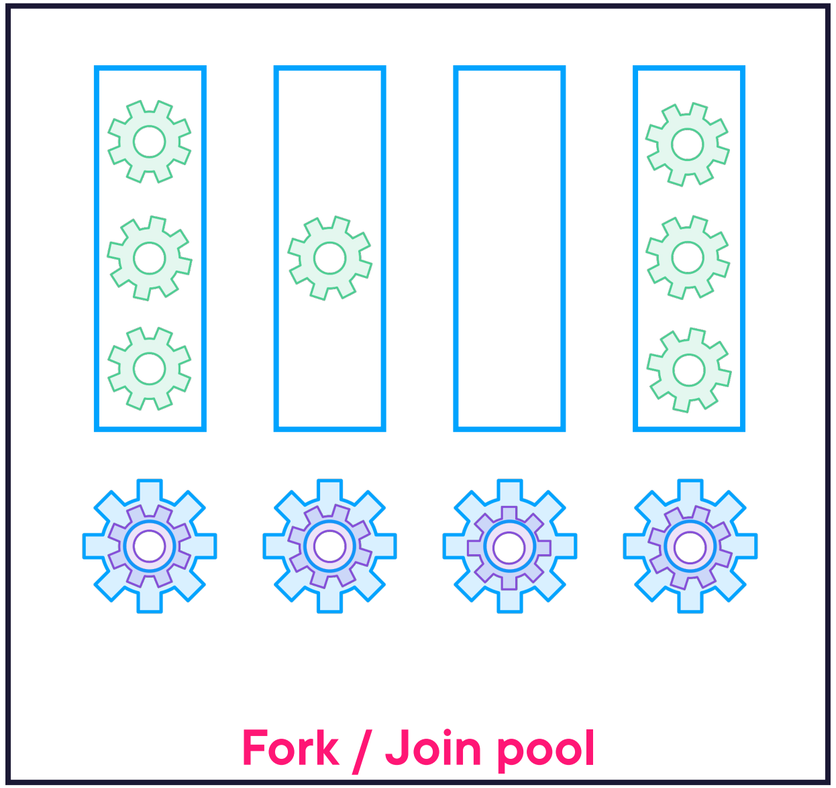
Running a Virtual Thread



Running a Virtual Thread



Running a Virtual Thread



How can it be more efficient than plain concurrent programming?

Running a task in a virtual thread is actually an overhead...

Executing in-memory computations in a virtual thread is more expensive than in a platform thread

Blocking a Virtual Thread

Virtual threads are not made to execute non-blocking code

They are made to execute blocking code

Blocking Virtual Threads

Let us create a virtual thread

Blocks it with a sleep

And observe that it jumps from one thread to the other

Seeing Continuation in Action

Continuation in Action

Let us create a Continuation

Pass a Runnable to it, and run it

Then call yield(), and then run()

Wrapping up Continuations

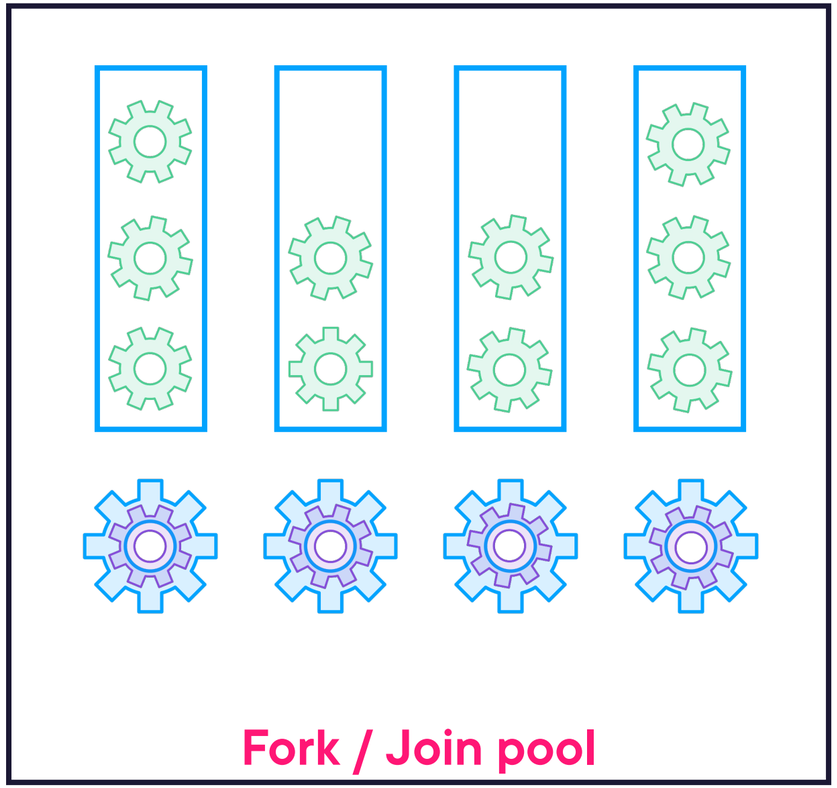
Under the hood, a
virtual thread is launched by a Continuation

When your virtual thread runs a blocking operation

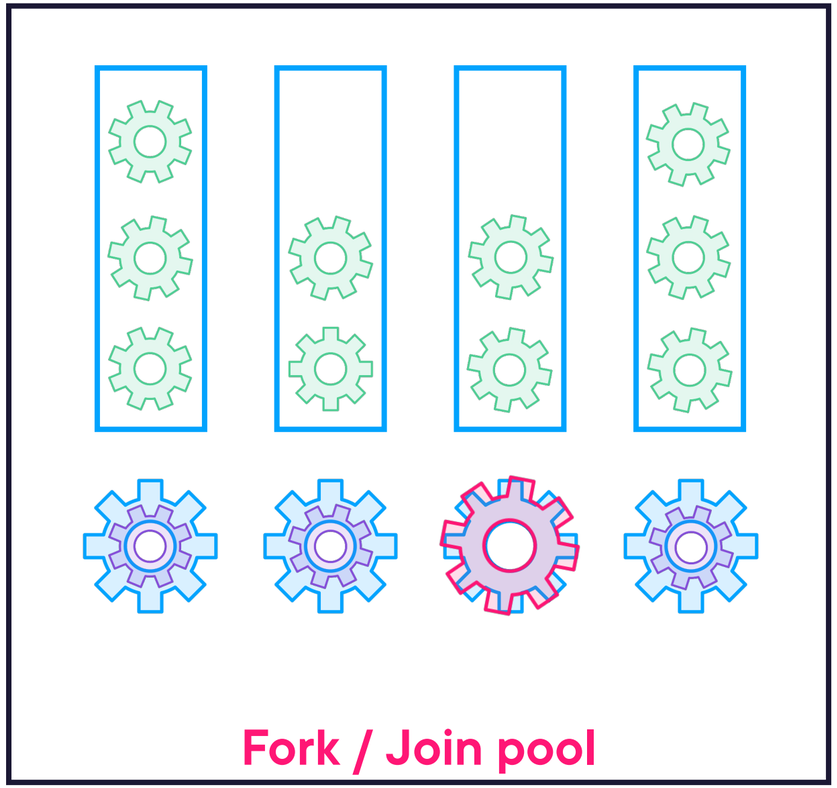
This operationcalls Continuation.yield()

It moves the virtual thread away from its carrier thread

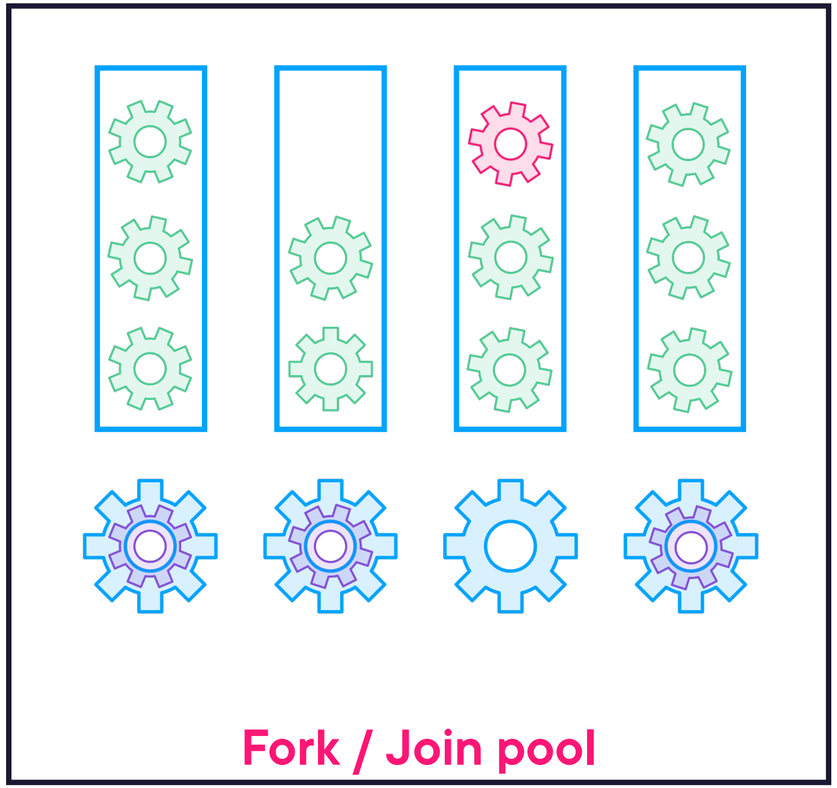
Running a Virtual Thread



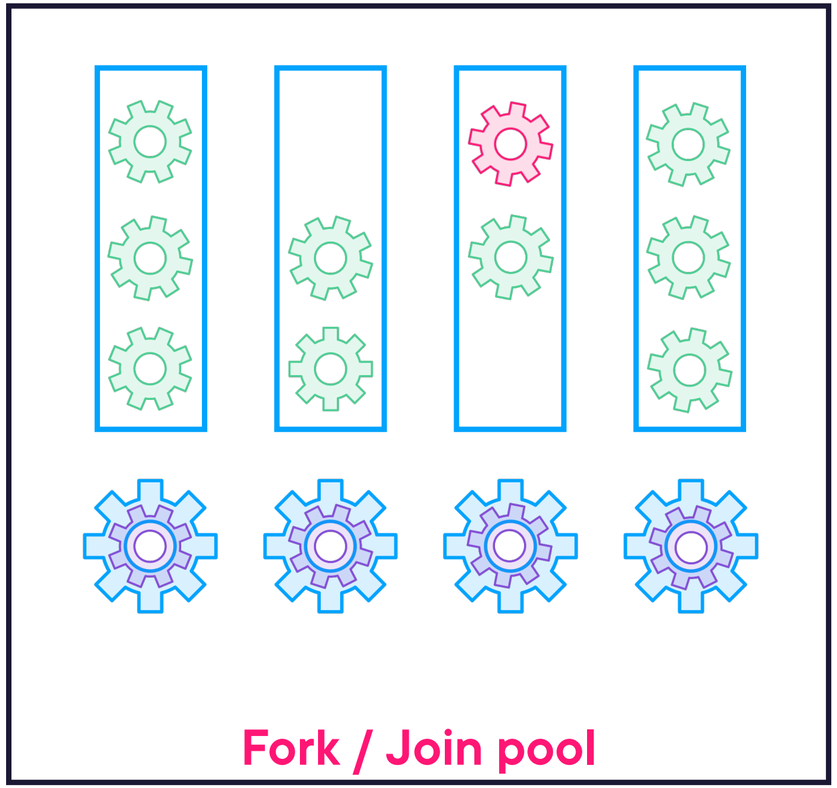
Running a Virtual Thread



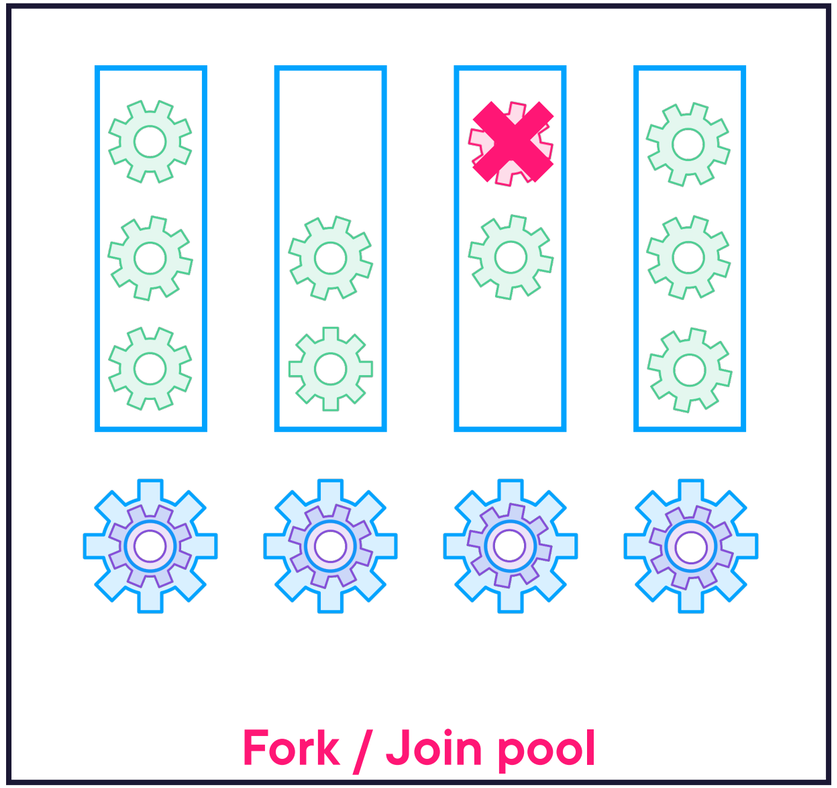
Running a Virtual Thread



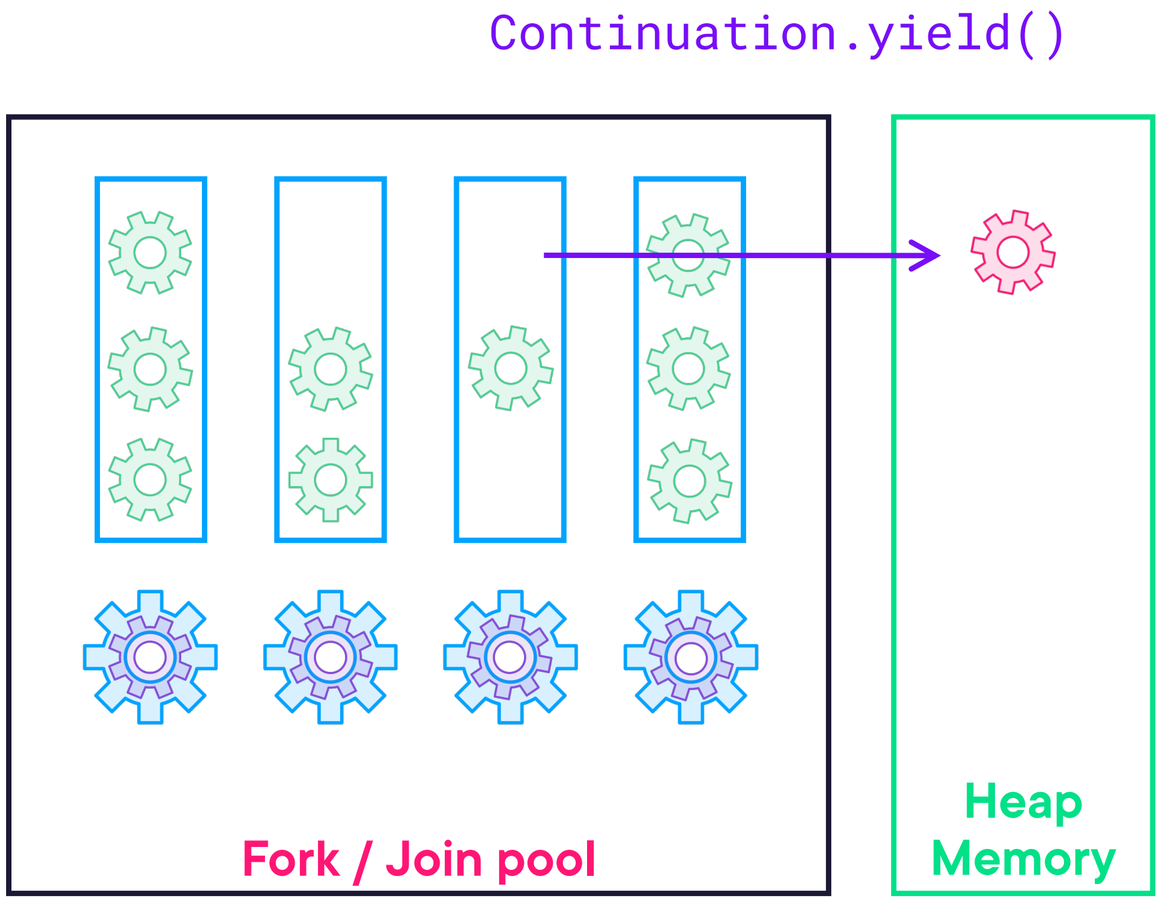
Running a Virtual Thread



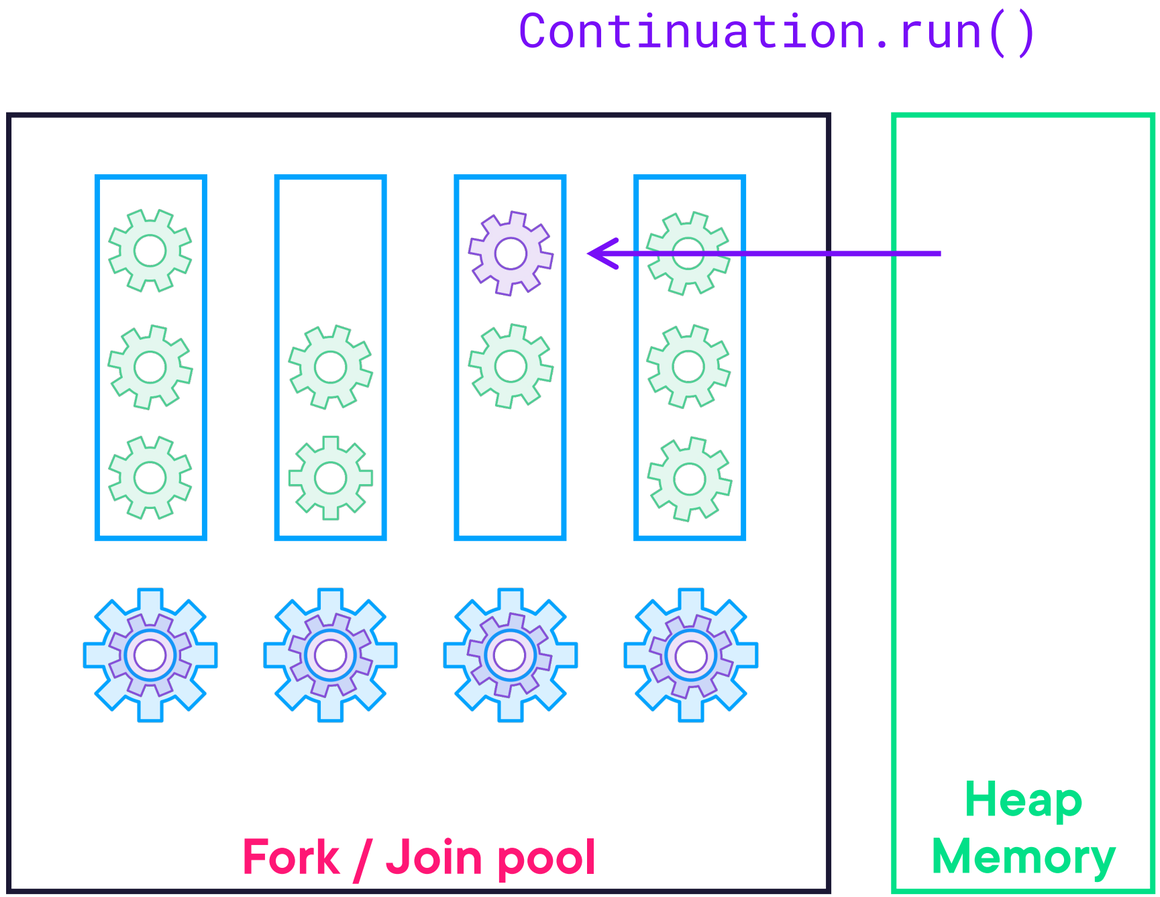
Running a Virtual Thread



Running a Virtual Thread



Running a Virtual Thread



Continuations make it so that
your platform threads are never blocked

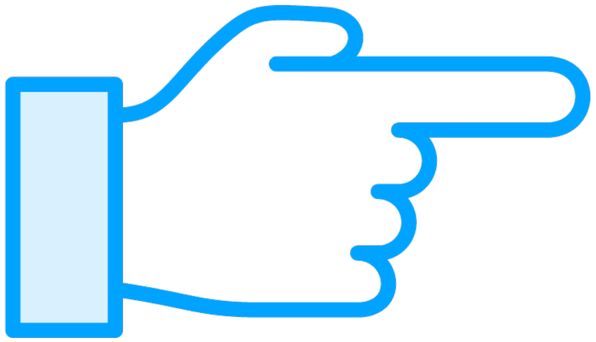
All your blocked virtual threads
are stored in your heap memory

And invoked again when your data is available

Pinning a Virtual Thread

```
int someFunction() {
  int number;
  int* numberRef;

  numberRef = &number;
  ...
}
```



There are cases where your virtual thread cannot be moved to the heap memory

If you are calling some native code

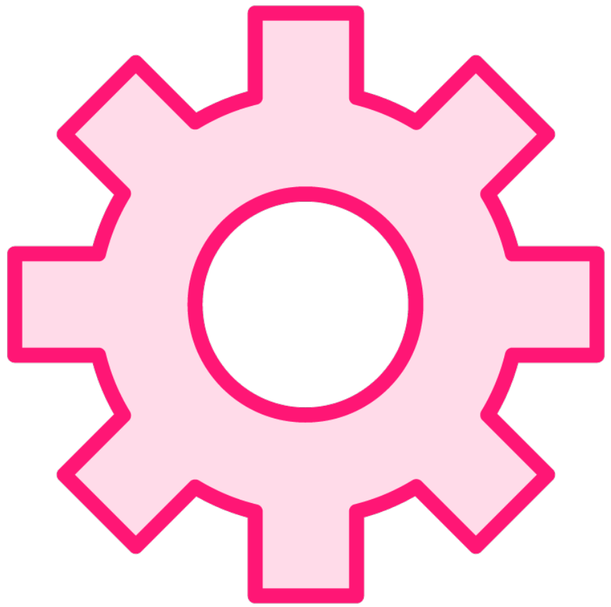
If your virtual thread is executing a synchronized block

Pinning on Synchronization

Pinning Virtual Threads

Let us create yet another virtual thread

And execute a synchronized block with it



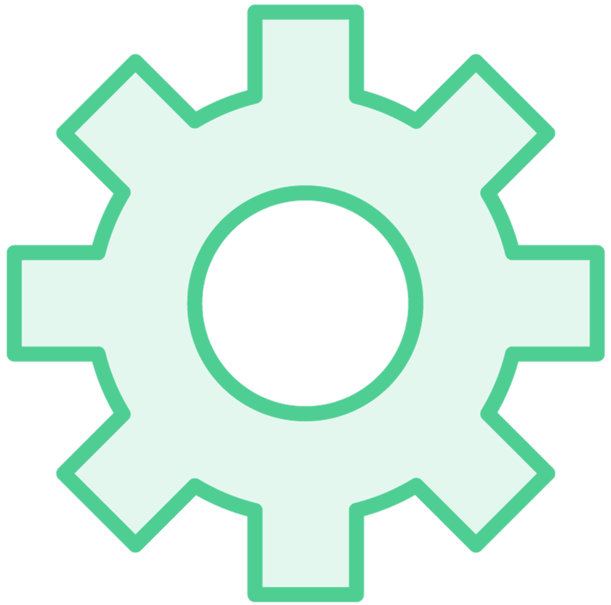
If a virtual thread stack can have addresses of elements contained in that stack

Then you cannot move it

Your virtual thread is pinned on your platform thread

You cannot have an adress on the stack in Java

But you can in C, in C++, ...



You have an address on the stack when you use a synchronized block

Is it bad?

Not always!

Guarding a list, or a map is OK

Querying an I/O resource is not OK

Fixing Synchronization

Avoid Pinning During Synchronization

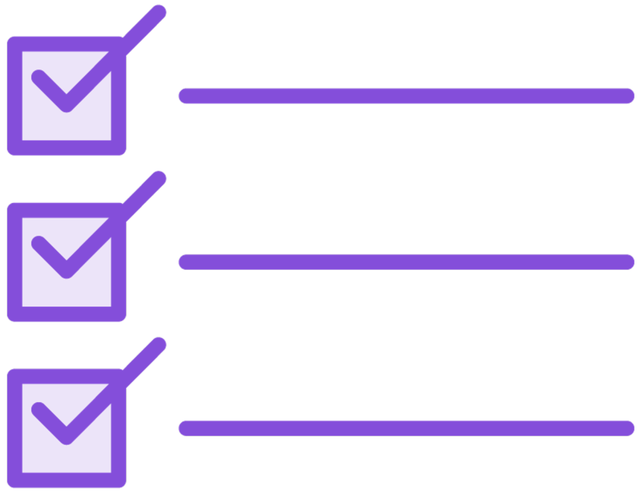
```
//
// Avoid running this is a Virtual Thread
//
var lock = new Object();
Runnable task = () -> {

   synchronized(lock) {
      // some I/O code
      // that takes milliseconds
      // to execute
   }
}
```

Avoid Pinning During Synchronization

```
//
// Use this pattern instead
//
var lock = new ReentrantLock();
Runnable task = () -> {
   lock.lock();
   try (lock) {
      // some I/O code
      // that takes milliseconds
      // to execute
   } finally {
      lock.unlock();
   }
}
```

Comparing Reactive and Virtual Threads

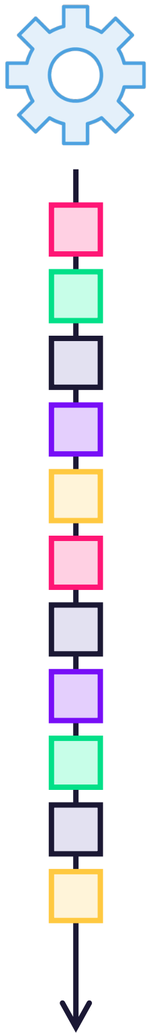


Both approaches rely on the fact that your platform thread should never be blocked

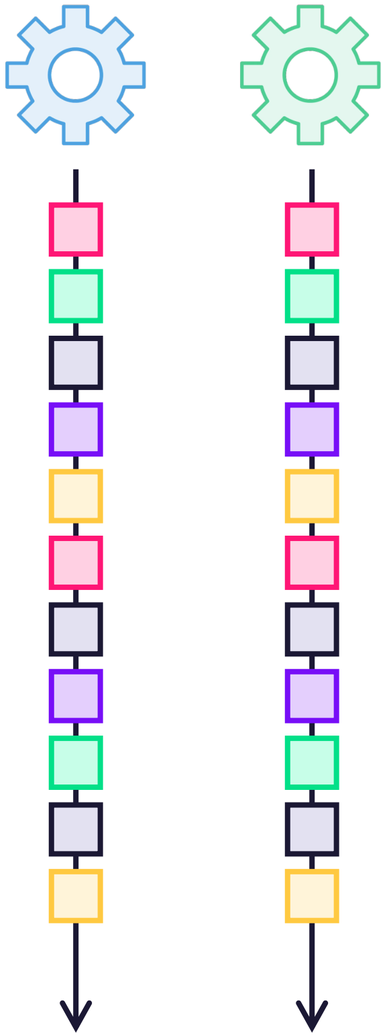
Reactive frameworks put this responsibility on you

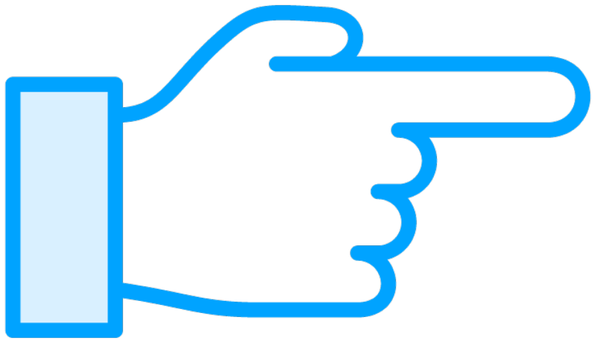
Virtual threads delegates this on the API (I/O, JDBC, etc...)

Platform Thread Usage with Reactive Programming



Platform Thread Usage with Virtual Threads





From your platform thread perspective,

There is no difference

between reactive programming
and virtual threads

The differences come from the frameworks

The most important difference
is the programming model

Reactive Programming =
you need to make sure that your code
is non-blocking

Virtual Threads =
your API takes care of that for you

Module Wrap Up

How virtual threads are working under the hood

They are made to execute blocking code

When they are blocked, they do not block this carrier thread

They can be pinned, you can use a reentrant lock instead of a synchronized block

Up Next:

Introducing Structured Concurrency