How (and why) nextest uses tokio

2022-10-03 (updated 11-03) • 19 min read #nextest #tokio #case-study

1. Introduction

Update 2022-11-03: This was originally the first in a series of two blog posts. I've now marked this as a standalone post. Content originally slated for part 2 will now be published as followup posts.

I'm the primary author and maintainer of cargo-nextest, a next-generation test runner for Rust released earlier this year. (It reached a thousand stars on GitHub recently; go star it here!)

Stars 2.5k

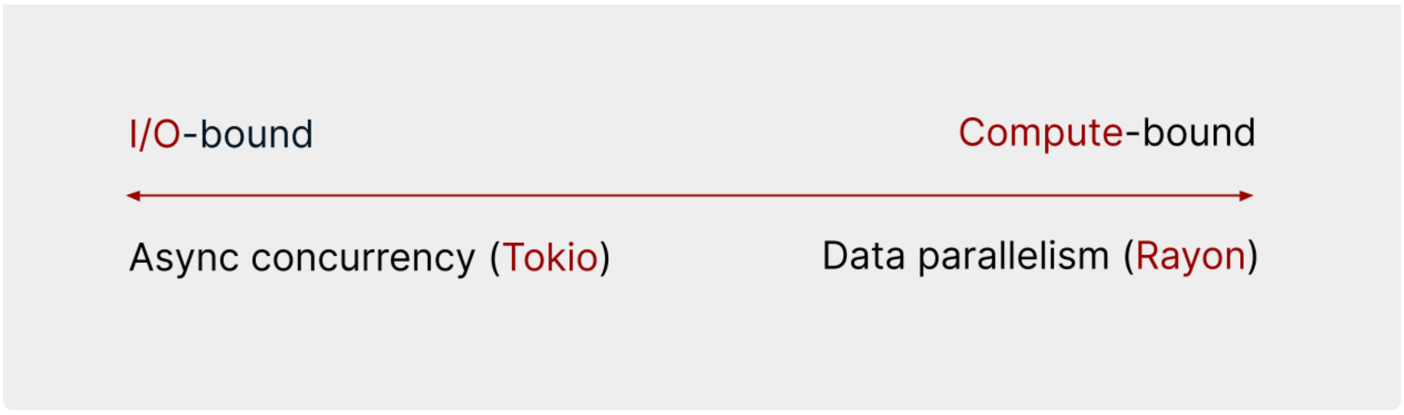
Nextest is faster than `cargo test` for most Rust projects, and provides a number of extra features, such as test retries, reusing builds, and partitioning (sharding) test runs. To power these features, nextest uses Tokio, the leading Rust async runtime for writing network applications.

Wait, what? Nextest isn't a network app at all[1]. However, nextest has to juggle multiple concurrent tests, each of which can be in the process of starting, printing output, failing, timing out, being cancelled or even forcibly stopped. In this post, I'll try to convince you that an async runtime is actually the perfect approach to handle a large number of heterogenous events like this.

There's more than one kind of distributed system

The challenges posed by interprocess communication *within a system* are really similar to those posed by remote communication *across systems*. Your computer is a distributed system. Just like most network services, nextest spends almost all of its time waiting rather than computing.

In other words, nextest provides the prototypical use-case for *asynchronous concurrency*, one of the two main paradigms for non-sequential code.



- The async concurrency style is best for I/O-bound applications, where most time is spent waiting on other parts of the overall system to respond. An I/O-bound application becomes faster if I/O becomes faster—for example, if your network link becomes faster, or you replace a spinning disk hard drive with a solid-state drive.
- On the other end of the spectrum, *data parallelism* with libraries like Rayon is best for compute-bound applications, where you're running a lot of computations in parallel on a set of input data. These become faster if you replace your CPU with a faster one.

Why is an async runtime like Tokio good for I/O-bound applications? The answer has to do with *heterogenous selects*[2].

What are heterogenous selects?

You might have heard of the *select* operation: in computer parlance, it means to observe two or more operations, and then to proceed with the first one that completes.

The name of the operation comes from the Unix `select` syscall. With `select` and its successors like `epoll`, you can typically only wait on one of several I/O read or write operations (represented through file descriptors). But real-world I/O-bound programs need to do much more than that! Here are some other kinds of operations they may need to wait on:

- Timeouts or recurring timers
- Channel reads and writes when using message-passing, including bounded channels with backpressure
- Process exits
- Unix signals

I/O-bound programs often need to wait across several of these sources at the same time. For example, it's common for services to make a network request, then select across:

- a network response
- a timeout

- a termination signal, so any pending network operations can be canceled

This is an example of a *heterogenous select* (or, spelled slightly differently, a *heterogeneous select*): a select operation that can span several different kinds of sources.

We also run into portability considerations: `select` is Unix-specific, and `epoll` is for the most part Linux-only[3]. Other operating systems have their own `epoll`-like abstractions, each with their own peculiarities:

- BSD-like platforms have `kqueue` (FreeBSD, macOS).
- Windows has I/O completion ports.

Some platforms may also support selecting across some kinds of heterogenous sources. For example, Linux has `signalfd` and `timerfd`. To the extent that these are supported, there are significant differences in the APIs available on each platform. And all of them make selecting across channels and other user code quite difficult.

The divergence in platform APIs and capabilities may not be a huge deal for certain kinds of network services that only target one platform, but is a real problem for a cross-platform developer tool like nextest.

So there are two separate but related sets of problems here:

1. The need to select over arbitrary kinds of heterogenous sources, not just I/O operations.
2. The need to do so in a cross-platform manner.

Async runtimes like Tokio solve both sets of problems in one go[4]. In fact, I'd like to make a bold claim here:

The ability to do heterogenous selects is *the point of async Rust*.

Async runtimes vs async/await

At this point it's worth mentioning that there are two related, but distinct, things here:

1. An *async runtime*: a framework that manages an event loop, handling events from heterogenous sources and associating them with individual tasks.
2. The *`async` and `await` keywords*, which provide an easy way to write tasks that wait on events handled by an async runtime.

It is definitely possible to leverage async runtimes without ever using the `async` and `await` keywords—in fact, I've written large chunks of a production service in Rust that used an async runtime before `async/await` was ever a thing.

However, without syntactic support it is often very difficult to do so, since you may need to hand-write complicated state machines. The `async` and `await` keywords dramatically simplify the experience of writing state machines. David Tolnay talks about this more in his blog post. (The `poll_next/EncodeState` example in his blog post was part of the aforementioned production service, and it was my code that he rewrote.)

2. The initial implementation

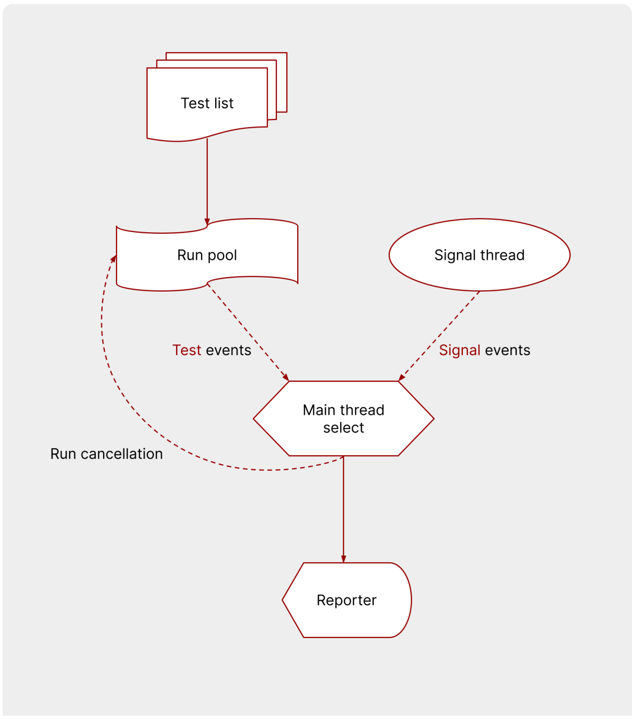
With this background, we're now ready to start looking at how nextest's current architecture came about.

The first versions of nextest all ran tests using "normal" threads. Here's the general structure they used for their execution model:

The run pool:

Tests were run via a thread pool.

1 Nextest first built up a list of tests.
2 It then spun up a thread pool with the number of tests[5], called the *run pool*.
3 For each test in the list, nextest used a thread in the run pool. The thread ran the test using the duct library, then blocked until the test exited. (nextest used duct for convenience—we could have used `std::process::Command` instead, which wouldn't have changed much.)
4 To send information about test events (the start, end, success/failure, and time taken for individual tests) to the main thread, nextest used a crossbeam MPSC channel[6]: we'll call this the *test event channel*.



The main ways data moves around among nextest's components.

The signal thread:

To handle signals such as Ctrl-C cleanly, nextest spun up a separate thread, and if a signal is detected, wrote to another channel: the *signal event channel*.

The main thread:

The main thread was dedicated to monitoring the two event channels. This thread would select across the *test event* and *signal event* channels:

▸ *Click to expand code sample*

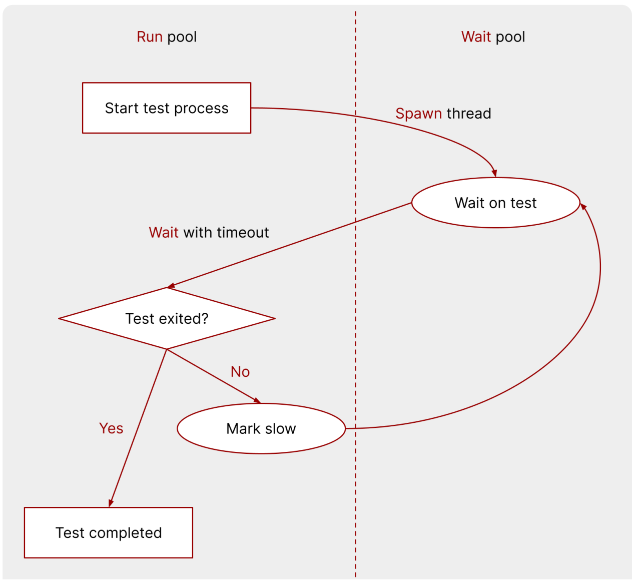
- If this loop received a test event, it would bubble that up to the reporter (the component that prints nextest's output). If a test failed, it would set a cancellation flag which would cause no further tests to be scheduled.
- If the loop received a signal event, it would set the cancellation flag.

Importantly, `crossbeam_channel::select!` only lets you select over crossbeam channels, not heterogenous sources in general. Anything else, including pipe reads or timers, must be translated over to crossbeam channels somehow.

The first issues with the model

One of the first demands placed on nextest was the need to identify slow tests. Note that duct's `Handle::wait` method doesn't have any notion of a timeout: it just blocks until the process is completed.

Luckily, `crossbeam_channel` has a `recv_timeout` method. So the solution was straightforward: set up another thread pool, called the *wait pool*, which would wait on test processes to exit.



How nextest's original runner loop detected slow tests.

So now this becomes:

1. Start the test process using a thread in the run pool.
2. Spawn a wait pool thread to wait on the test.
3. In the run pool, wait on the thread in the wait pool with a timeout.
   - If the test finishes before the timeout, mark it done.
   - If the timeout is hit, mark the test slow and go back to waiting on the test.

Here's the code:

▸ *Click to expand code sample*

This works, but you can already see the cracks starting to show. To solve this issue we had to introduce a whole new thread pool on top of the existing run pool.

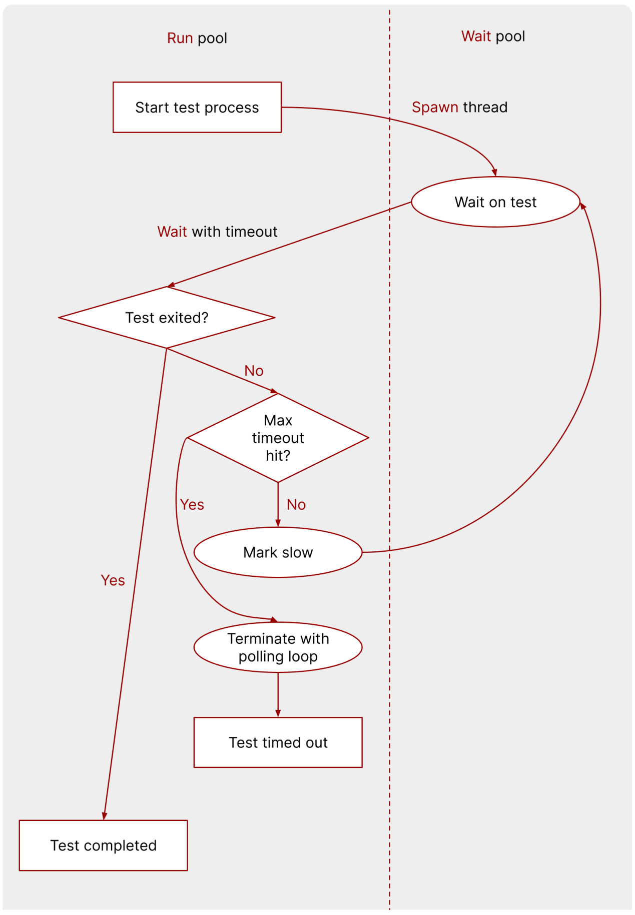
Terminating tests using a polling loop

A few months after nextest was released, a user requested another feature: the ability to terminate tests that take more than a certain amount of time.

Now, nextest *could* terminate tests immediately with the unblockable `SIGKILL` signal[7] (aka `kill -9`). But it's generally a good idea to give tests a bit of a grace period. So we chose to first send tests a `SIGTERM` signal, then wait a bit for them to clean up before sending `SIGKILL`.

To do this, we defined two timeouts:

1. A *slow timeout*, after which tests are marked as slow. Nextest's default slow timeout is 60 seconds.
2. A *termination timeout*, after which tests are killed. Nextest doesn't have a default termination timeout, but a typical one users might set is 5 minutes (300 seconds).



How nextest's original runner loop implemented test timeouts.

The test execution flow is then:

1. Start the test process using a thread in the run pool.
2. Spawn a wait pool thread to wait on the test.
3. In the run pool, wait on the thread in the wait pool with the slow timeout.
   - If the test finishes before the slow timeout, mark it done.
   - If the slow timeout is hit and the termination timeout hasn't been hit yet, mark the test as slow and go back to waiting on it.
   - If the termination timeout is hit, set up a termination loop.

And the termination loop:

1. Send the process `SIGTERM`.
2. In a loop, for up to 10 seconds:
   - Check if the test process exited.
   - If it did, break out of the loop.
   - Otherwise, wait 100ms and check the test's status again.
3. If the test hasn't exited by the end of the loop, send it `SIGKILL`.

Here's the code:

▸ *Click to expand code sample*

You can already start seeing the code becoming long and rather janky, with its 100ms polling loop.

Detecting leaked handles

Some tests spawn server or other subprocesses, and sometimes those subprocesses don't get cleaned up at the end of the test. In particular, nextest is concerned about processes that *inherit* standard output and/or standard error. Here's an example test:

```
#[test]
fn test_subprocess_doesnt_exit() {
    // Spawn a subprocess that sleeps for 2 minutes, then exits.
    let mut cmd = std::process::Command::new("sleep");
    cmd.arg("120");
    // This sleep command inherits standard output and standard error from the
test.
    cmd.spawn().unwrap();
}
```

Since `sleep` inherits standard output and standard error from the test, it keeps those file descriptors open until it exits. Previously, nextest would simply wait until those subprocesses exited. In the worst case, the subprocesses wouldn't exit at all: think of a server process that gets started but never shut down. That can result in nextest simply hanging, which isn't great.

How can nextest detect these sorts of *leaky* tests? One solution is to wait for standard output and standard error to be closed until a short duration (say 100ms) passes. While this is conceptually simple, *implementing it* is surprisingly complicated. Waiting on one handle being closed is nontrivial; waiting on *two* handles being closed (standard output and standard error) adds an extra challenge.

I spent some time looking at how to implement it manually, and came away from it with the understanding that I'd have to maintain separate implementations for Windows and Unix. Not only that, I'd just be duplicating a lot of the work Tokio had already done to abstract over platform differences. So I started looking at porting over nextest to Tokio.

3. Using Tokio

I spent a weekend porting over nextest to use async Rust with Tokio. After a bit of iteration, I settled on this general architecture:

1. Create a Tokio `Runtime` in the beginning. The `Runtime` lasts for as long as the runner loop does.
2. Turn the list of tests into a `Stream`, then for each test, create a future that executes the test. Use `buffer_unordered` to control concurrency[8].
3. Switch over crossbeam channels to tokio channels as needed, and `crossbeam_channel::select!` invocations to `tokio::select!`.

Here's the bit that selects over the test event and signal event channels:

▸ *Click to expand code sample*

This is really similar to the crossbeam-based implementation above, with just one real change: The code tracks a `signals_done` state to see if the signal event channel has been dropped. This is to ensure that the loop doesn't repeatedly get `None` values from the signal handler, wasting cycles. (This is more important with futures that, once completed, cannot be waited on further, as we'll see in the next section.)

Waiting on process completion with Tokio

Where this gets really interesting is with the loop nextest uses to wait for tests to complete. It turns out that *all* the code we wrote above can be replaced with a set of Rust and Tokio primitives, written in a roughly declarative style.

After starting the process, nextest sets up a timer, and futures to read from standard output and standard error:

▸ *Click to expand code sample*

Some notes about the code above:

- On being polled, the futures repeatedly read standard output and standard error into their corresponding buffers in 4 KiB chunks. They only exit when 0 bytes are read, which

corresponds to the handle being closed.
- The code needs to maintain `stdout_done` and `stderr_done` flags: this is similar to `signals_done` above, except this time it's necessary because async blocks cannot be polled once completed: if done so, they panic with the message "`async fn` resumed after completion".
- The futures borrow data from the stack. This is possible thanks to async Rust's integration with the borrow checker.
- The futures are *pinned* to the stack, which means that they cannot be moved. This is a fundamental part of how zero-overhead async Rust works. An async block must be pinned before it can be polled.

Remember how we talked about heterogenous selects in part 1 of this post? This is where we start using them. Notice how `crossbeam_channel::select!` only works on crossbeam channels. On the other hand, Tokio's select works on arbitrary, heterogenous sources of asynchronicity.

Here's how nextest now waits on tests:

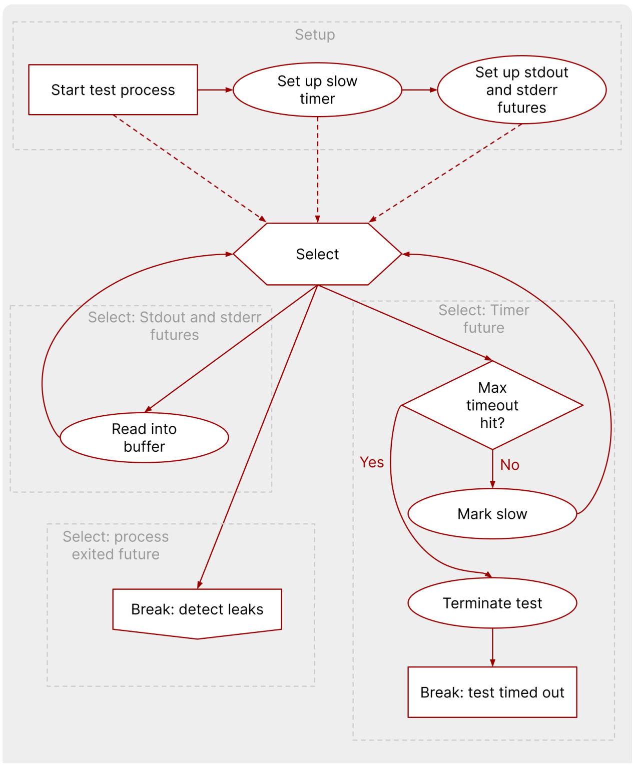
Setup:

1. Start the test process, which sets up a future that completes when the test exits.
2. Set up a timer for the slow timeout.
3. Set up futures to read from standard output and standard error.

Then, select across the futures listed above in a loop, using Tokio's ability to handle heterogenous selects with `tokio::select!`.

If data is available via the standard output or standard error futures:

1. Read data into the corresponding buffer.
2. Loop back and select again.



How nextest's Tokio-based runner loop executes tests.

If the timer fires:

1. If the terminate timeout is hit, terminate the test and exit the loop.
2. If not, mark as slow, loop back and select again.

If the process exits:

1. Exit the execution loop.

2 Proceed to the leak detection loop below.

Here's the code:

▸ *Click to expand code sample*

And then, how does nextest detect leaked handles? After the main loop ends, nextest just runs another little loop for that:

1 Set up a short (by default 100ms) leak timer.
2 Select across the leak timer and the standard output and error futures from before, in a loop.

If data is available via the standard output or standard error futures:

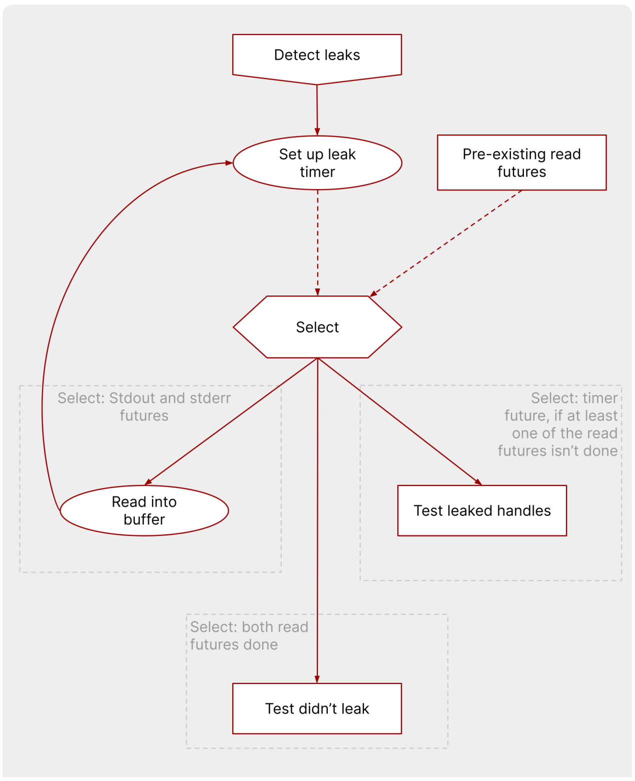
1 Read data into the corresponding buffer.
2 Loop back, and select again.

If both the standard output and standard error futures have completed:

1 Note that the test didn't leak.
2 Conclude test execution.

If the timer fires before both futures have completed:

1 Note that the test leaked handles.
2 Conclude test execution.



Detect leaks
Set up leak timer
Pre-existing read futures
Select
Select: Stdout and stderr futures
Select: timer future, if at least one of the read futures isn't done
Read into buffer
Test leaked handles
Select: both read futures done
Test didn't leak

How nextest's Tokio-based runner loop detects leaked handles in tests.

Here's the code:

▸ *Click to expand code sample*

That's it! We've replaced all of that unwieldy code with a couple of declarative `select!` loops.

If it compiles, does it work?

One of the promises of Rust's type system is that it'll catch most kinds of bugs before you ever run the program. In my experience, this promise is *mostly* upheld, even when dealing with async code. For a few compile-time issues like pinning, it was easy to find solutions.

I ran into two runtime issues:

1. A panic with the message "`async fn` resumed after completion". The solution to this was easy to find.
2. The main loop that selected across the test event and signal event channels hung during longer test runs; this was caused by a `biased` statement. Removing the `biased` fixed it, but I mostly found that out by trial and error. I'm not sure how to debug this kind of issue in a principled manner.

4. Conclusion

There's a bit of a meme in the Rust world: that if you can "just use" threads, you should; and that async code is really only "worth it" for network services that need to be massively concurrent. (I don't mean to pick on the individual authors here. It's a general view in the Rust community.)

While I agree that async Rust isn't *easy*, I believe nextest's experience using tokio directly challenges that general view. The point of async, even more than the concurrency, is to make it easy to operate across arbitrary sources of asynchronicity. This is exactly what nextest needs to do. Any Rust program that is I/O-bound, or any program that needs to do heterogenous selects, should strongly consider using an async runtime like Tokio.

Thanks

Thanks to Fiona, Inanna, Eliza, Brandon, Predrag, and Michael for reviewing drafts of this article. And thank you for reading it to the end.

The flowcharts are available in this Google folder, and are licensed under CC BY 4.0.

Changelog

2022-10-06:

- Add a note that platform-specific APIs like `signalfd` and `timerfd` can provide heterogenous selects to some extent, though not as much as Tokio can.
- Updated `stdout_fut` and `stderr_fut` implementations to match changes in #560.
- I'd forgotten to add Michael Gattozzi to the reviewers list. Fixed, sorry!

1 Other than its self-update functionality, which isn’t relevant here. ↩

2 Full credit for the term and framing goes to Eliza Weisman. ↩

3 Some other systems have added `epoll` for compatibility with Linux: for example, illumos. ↩

4 The platform-specific implementations are usually abstracted away with a low-level library like Mio. ↩

5 The runner loop needed a scoped thread pool because it relied on borrowed data—it used rayon’s `ThreadPool` for that purpose. ↩

6 MPSC stands for “multi-producer, single consumer”. This kind of channel lets many threads (in this case, the threads running the tests) send data to a single “status” thread. ↩

7 Or its equivalent on Windows, TerminateProcess. ↩

8 Why `buffer_unordered`? Because we’d like tests to be *started* in lexicographic order, but results to be reported in the order tests finish in. That is exactly the behavior `buffer_unordered` provides.

One alternative was to spawn every test as a separate future and use a semaphore to control concurrency, but that might have resulted in tests being started in arbitrary order. ↩