Async Rust: What is a runtime? Here is how tokio works under the hood

Tue, Feb 1, 2022

Last week, we saw the difference between Cooperative and Preemptive scheduling and how it enables resources-efficient I/O operations. Today, we are going to learn how runtimes work under the hood.

Rust does not provide the execution context required to execute Futures and Streams. This execution context is called a **runtime**. You can't run an `async` Rust program without a runtime.

The 3 most popular runtimes are:

Runtime	All-time downloads (July 2022)	Description
tokio	59,048,636	An event-driven, non-blocking I/O platform for writing asynchronous I/O backed applications.
async-std	8,002,852	Async version of the Rust standard library
smol	1,491,204	A small and fast async runtime

However, there is a problem: today, runtimes are not interoperable and require acknowledging their specificities in code: you can't easily swap a runtime for another by changing only 1-2 lines of code.

Work is done to permit interoperability in the future, but today, the ecosystem is fragmented. You have to pick one and stick to it.

Introducing tokio

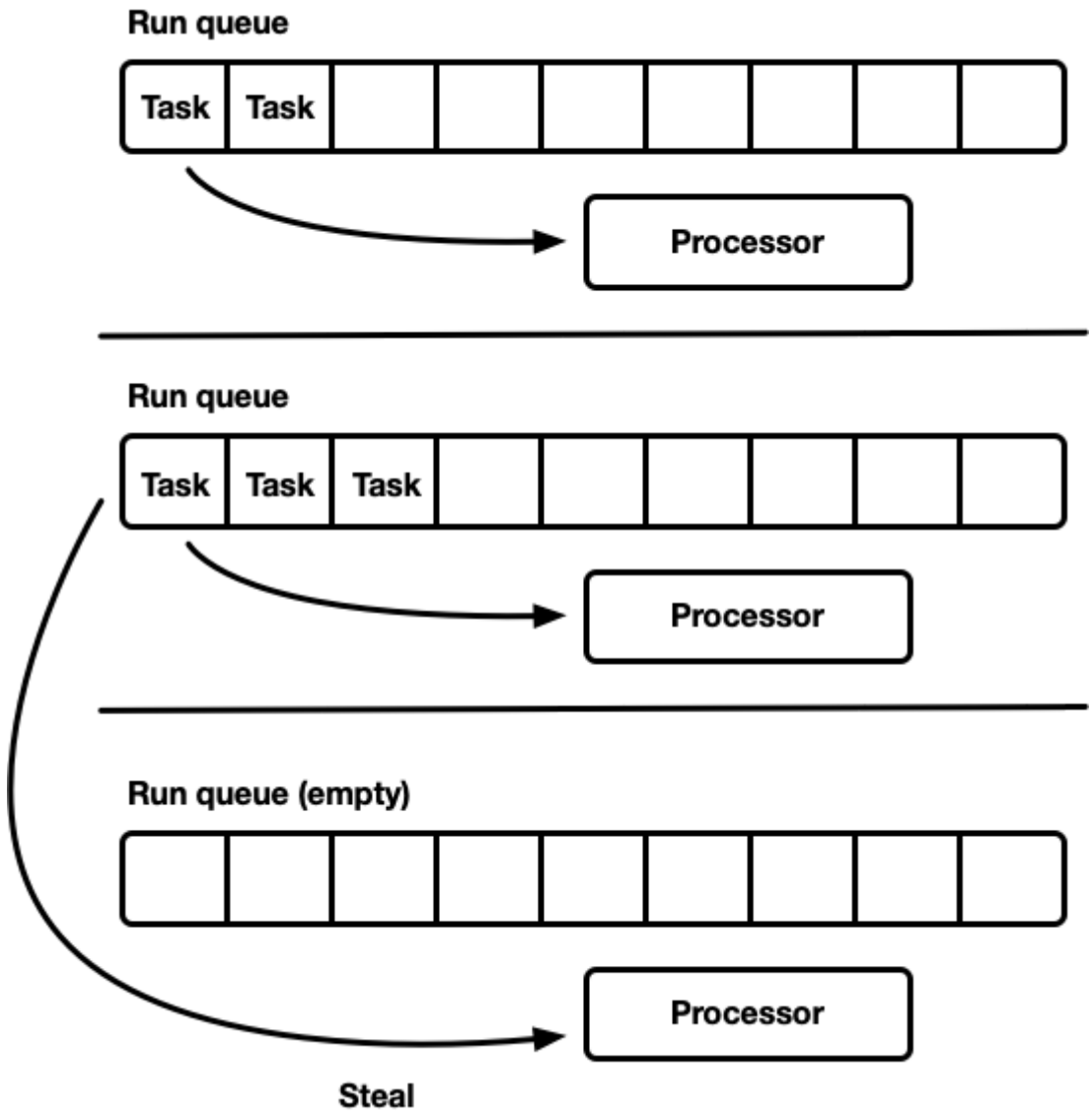
Tokio is the Rust async runtime with the biggest support from the community and has

many sponsors (such as Discord, Fly.io, and Embark), which allow it to have paid contributors!

If you are **not** doing embedded development, this is the runtime you should use. There is no hesitation to have.

The event loop(s)

At the core of all `async` runtimes (whether it be in Rust, Node.js, or other languages) are the **event loops**, also called **processors**.



Work stealing runtime. By Carl Lerche - License MIT - https://tokio.rs/blog/2019-10-scheduler#the-next-generation-tokio-scheduler

In reality, for better performance, there are often multiple processors per program, one per CPU core.

Each event-loop has its own queue of tasks `await`ing for completion. Tokio's is known to be a work-stealing runtime. Each processor can steal the task in the queue of another processor if its own is empty (i.e. it has nothing to do and is "sitting" idle).

To learn more about the different kinds of event loops, you can read this excellent article by Carl Lerche: https://tokio.rs/blog/2019-10-scheduler.

Spawning

When you want to dispatch a task to the runtime to be executed by a processor, you `spawn` it. It can be achieved with tokio's `tokio::spawn` function.

For example:
ch_03/tricoder/src/ports.rs

```
tokio::spawn(async move {
    for port in MOST_COMMON_PORTS_100 {
        let _ = input_tx.send(*port).await;
    }
});
```

This snippet of code spawns 1 task that will be pushed into the queue of one of the processors. As each processor have its own OS thread, by spawning a task, we use all the resources of our machine without having to manage threads ourselves. Without spawning, all the operations are executed on the same processor and thus the same thread.

As a matter of comparison, in Go we use the `go` keyword to spawn a task (called a goroutine):

```
go doSomething()
```

Avoid blocking the event loops

THIS IS THE MOST IMPORTANT THING TO REMEMBER.

The most important rule to remember in the world of `async-await` is **not to block the event loop**.

What does it mean? Not calling functions that may run for more than 10 to 100 microseconds directly. Instead, `spawn_blocking` them.

CPU intensive operations

So, how to execute compute-intensive operations, such as encryption, image encoding, or file hashing?

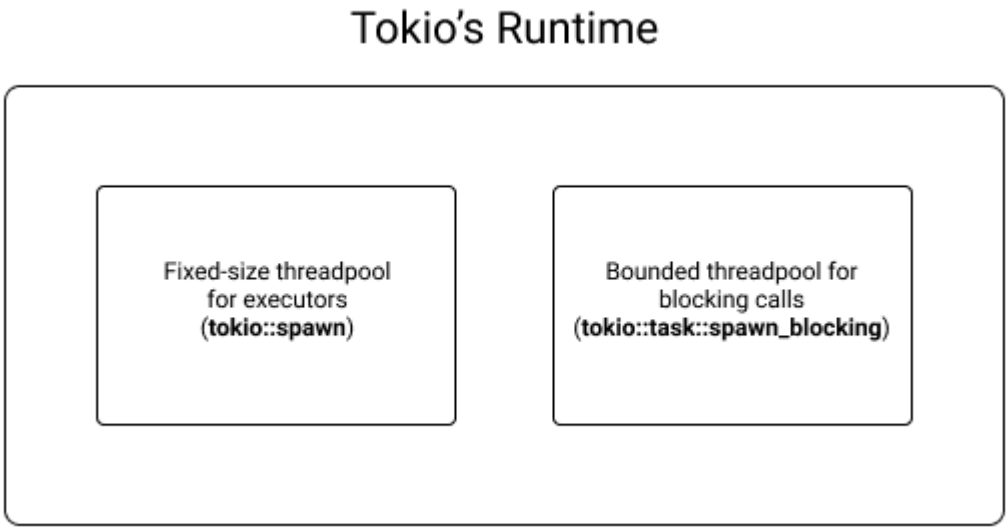
`tokio` provides the `tokio::task::spawn_blocking` function for blocking operations that eventually finish on their own. By that, I mean a blocking operation which is not an infinite background job. For this kind of task, a Rust Thread is more appropriate.

Here is a an example from an application where `spawn_blocking` is used:

```
let is_code_valid = spawn_blocking(move || crypto::verify_password(&code, &c
```

Indeed, the function `crypto::verify_password` is expected to take hundreds of milliseconds to complete, it would block the event loop.

Instead, by calling `spawn_blocking`, the operation is dispatched to tokio's blocking tasks thread pool.



Under the hood, tokio maintains two thread pools.

One fixed-size thread pool for its executors (event-loops, processors) which execute async tasks. Async tasks can be dispatched to this thread pool using `tokio::spawn`.

And one dynamically sized but bounded (in size) thread pool for blocking tasks. By default, the latter will grow up to 512 threads. This thread pool scale up and down depending on the number of running blocking tasks. Blocking tasks can be dispatched to this thread pool using `tokio::task::spawn_blocking`. You can read more about how to finely configure it in tokio's documentation.

This is why `async-await` is also known as "Green threads" or "M:N threads". They look like threads for the user (the programmer), but `spawning` is cheaper, and you can spawn way more green threads than the actual number of OS threads the runtime is going to use under the hood.