Async: What is blocking?

Published 2020-12-21

The async/await feature in Rust is implemented using a mechanism known as cooperative scheduling, and this has some important consequences for people who write asynchronous Rust code.

The intended audience of this blog post is new users of async Rust. I will be using the Tokio runtime for the examples, but the points raised here apply to any asynchronous runtime.

If you remember only one thing from this article, this should be it:

> Async code should never spend a long time without reaching an `.await`.

Translations: chinese

Blocking vs. non-blocking code

The naive way to write an application that works on many things at the same time is to spawn a new thread for every task. If the number of tasks is small, this is a perfectly fine solution, but as the number of tasks becomes large, you will eventually run into problems due to the large number of threads. There are various solutions to this problem in different programming languages, but they all boil down to the same thing: very quickly swap out the currently running task on each thread, such that all of the tasks get an opportunity to run. In Rust, this swapping happens when you `.await` something.

When writing async Rust, the phrase “blocking the thread” means “preventing the runtime from swapping the current task”. This can be a major issue because it means that other tasks on the same runtime will stop running until the thread is no longer being blocked. To prevent this, we should write code that can be swapped quickly, which you do by never spending a long time away from an `.await`.

Let's take an example:

```rust
use std::time::Duration;

#[tokio::main]
async fn main() {
    println!("Hello World!");

    // No .await here!
    std::thread::sleep(Duration::from_secs(5));

    println!("Five seconds later...");
}
```

The above code looks correct, and if you run it, it will appear to work. But it has a fatal flaw: it is blocking the thread. In this case, there are no other tasks, so it's not a problem, but this wont be the case in real programs. To illustrate this point, consider the following example:

```
use std::time::Duration;

async fn sleep_then_print(timer: i32) {
    println!("Start timer {}.", timer);

    // No .await here!
    std::thread::sleep(Duration::from_secs(1));

    println!("Timer {} done.", timer);
}

#[tokio::main]
async fn main() {
    // The join! macro lets you run multiple things concurrently.
    tokio::join!(
        sleep_then_print(1),
        sleep_then_print(2),
        sleep_then_print(3),
    );
}
```

```
Start timer 1.
Timer 1 done.
Start timer 2.
Timer 2 done.
Start timer 3.
Timer 3 done.
```

The example will take three seconds to run, and the timers will run one after the other with no concurrency whatsoever. The reason is simple: the Tokio runtime was not able to swap one task for another, because such a swap can only happen at an `.await`. Since there is no `.await` in `sleep_then_print`, no swapping can happen while it is running.

However if we instead use Tokio's sleep function, which uses an `.await` to sleep, the function will behave correctly:

```
use tokio::time::Duration;

async fn sleep_then_print(timer: i32) {
    println!("Start timer {}.", timer);

    tokio::time::sleep(Duration::from_secs(1)).await;
//                                            ^ execution can be pa

    println!("Timer {} done.", timer);
}

#[tokio::main]
async fn main() {
    // The join! macro lets you run multiple things concurrently.
    tokio::join!(
        sleep_then_print(1),
        sleep_then_print(2),
        sleep_then_print(3),
    );
}
```

```
Start timer 1.
Start timer 2.
Start timer 3.
Timer 1 done.
Timer 2 done.
Timer 3 done.
```

The code runs in just one second, and properly runs all three functions at the same time as desired.

Be aware that it is not always this obvious. By using `tokio::join!`, all three tasks are guaranteed to run on the same thread, but if you replace it with `tokio::spawn` and use a multi-threaded runtime, you *will* be able to run multiple blocking tasks until you run out of threads. The default Tokio runtime spawns one thread per CPU core, and you will typically have around 8 CPU cores. This is enough that you can miss the issue when testing locally, but sufficiently few that you will very quickly run out of threads when running the code for real.

To give a sense of scale of how much time is too much, a good rule of thumb is no more than 10 to 100 microseconds between each `.await`. That said, this depends on the kind of application you are writing.

What if I want to block?

Sometimes we just want to block the thread. This is completely normal. There are two common reasons for this:

1. Expensive CPU-bound computation.
2. Synchronous IO.

In both cases, we are dealing with an operation that prevents the task from reaching an `.await` for an extended period of time. To solve this issue, we must move the blocking operation to a thread outside of Tokio's thread pool. There are three variations on this:

1. Use the `tokio::task::spawn_blocking` function.
2. Use the `rayon` crate.
3. Spawn a dedicated thread with `std::thread::spawn`.

Let us go through each solution to see when we should use it.

The `spawn_blocking` function

The Tokio runtime includes a separate thread pool specifically for running blocking functions, and you can spawn tasks on it using `spawn_blocking`. This thread pool has an upper limit of around 500 threads, so you can spawn quite a lot of blocking operations on this thread pool.

Since the thread pool has so many threads, it is best suited for blocking IO such as interacting with the file system or using a blocking database library such as `diesel`.

The thread pool is poorly suited for expensive CPU-bound computations, since it has many more threads than you have CPU cores on your computer. CPU-bound computations run most efficiently if the number of threads is equal to the number of CPU cores. That said, if you only need a few CPU-bound computations, I wont blame you for running them on `spawn_blocking` as it is quite simple to do so.

```rust
#[tokio::main]
async fn main() {
    // This is running on Tokio. We may not block here.

    let blocking_task = tokio::task::spawn_blocking(|| {
        // This is running on a thread where blocking is fine.
        println!("Inside spawn_blocking");
    });

    // We can wait for the blocking task like this:
    // If the blocking task panics, the unwrap below will propagate
    // panic.
    blocking_task.await.unwrap();
}
```

The rayon crate

The rayon crate is a well known library that provides a thread pool specifically intended for expensive CPU-bound computations, and you can use it for this purpose together with Tokio. Unlike `spawn_blocking`, the `rayon` thread pool has a small maximum number of threads, which is why it is suitable for expensive computations.

We will use the sum of a large list as an example of an expensive computation, but note that in practice, unless the array is very very large, just computing a sum is probably cheap enough that you can just do it directly in Tokio.

The main danger of using `rayon` is that you must be careful not to block the thread while waiting for rayon to complete. To do this, combine `rayon::spawn` with `tokio::sync::oneshot` like this:

```rust
async fn parallel_sum(nums: Vec<i32>) -> i32 {
    let (send, recv) = tokio::sync::oneshot::channel();

    // Spawn a task on rayon.
    rayon::spawn(move || {
        // Perform an expensive computation.
        let mut sum = 0;
        for num in nums {
            sum += num;
        }

        // Send the result back to Tokio.
        let _ = send.send(sum);
    });

    // Wait for the rayon task.
    recv.await.expect("Panic in rayon::spawn")
}

#[tokio::main]
async fn main() {
    let nums = vec![1; 1024 * 1024];
    println!("{}", parallel_sum(nums).await);
}
```

This uses the rayon thread pool to run the expensive operation. Be aware that the above example uses only one thread in the rayon thread pool per call to `parallel_sum`. This makes sense if you have many calls to `parallel_sum` in your application, but it is also possible to use rayon's parallel iterators to compute the sum on several threads:

```rust
use rayon::prelude::*;

// Spawn a task on rayon.
rayon::spawn(move || {
    // Compute the sum on multiple threads.
    let sum = nums.par_iter().sum();

    // Send the result back to Tokio.
    let _ = send.send(sum);
});
```

Note that you still need the `rayon::spawn` call when you use parallel iterators, because parallel iterators are blocking.

Spawn a dedicated thread

If a blocking operation keeps running forever, you should run it on a dedicated thread. For example consider a thread that manages a database connection using a channel to receive database operations to perform. Since this thread is listening on that channel in a loop, it never exits.

Running such a task on either of the two other thread pools is a problem, because it essentially takes away a thread from the pool permanently. Once you've done that a few times, you have no more threads in the thread pool and all other blocking tasks fail to get executed.

Of course you can also use dedicated threads for shorter lived purposes if you are okay with paying the cost of spawning a new thread every time you start a new one.

Summary

In case you forgot, here's the main thing you need to remember:

> Async code should never spend a long time without reaching an `.await`.

Below you will find a cheat sheet of what methods you can use when you want to block:

	CPU-bound computation	**Synchronous IO**	**Running forever**
`spawn_blocking`	Suboptimal	OK	No
`rayon`	OK	No	No
Dedicated thread	OK	OK	OK

Finally, I recommend checking out the chapter on shared state from the Tokio tutorial. This chapter explains how you can correctly use `std::sync::Mutex` in async code, and goes more in-depth with why this is okay even though locking a mutex is blocking. (Spoiler: if you block for a short time, is it really blocking?)

I also strongly recommend the article Reducing tail latencies with automatic cooperative task yielding from the Tokio blog.

Thanks to Chris Krycho and snocl for reading drafts of this post and providing helpful advice. All mistakes are mine.