Extracted from:

Rust Brain Teasers

Exercise Your Mind

This PDF file contains pages extracted from *Rust Brain Teasers*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Rust Brain Teasers

Exercise Your Mind

Herbert Wolverson

edited by Tammy Coron

Rust Brain Teasers

Exercise Your Mind

Herbert Wolverson

The Pragmatic Bookshelf
Raleigh, North Carolina

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Tammy Coron
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-680509-17-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2022

To Henry, my loyal canine coding companion of thirteen years—who sadly didn't live to see the book's release.

Puzzle 19

Sleepless in Tokio

sleepless/Cargo.toml

```
[package]
name = "sleepless"
version = "0.1.0"
edition = "2018"

[dependencies]
tokio = { version = "1.7", features = ["full"] }
```

sleepless/src/main.rs

```
use tokio::join;
use std::time::Duration;

async fn count_and_wait(n: u64) -> u64 {
    println!("Starting {}", n);
    std::thread::sleep(Duration::from_millis(n * 100));
    println!("Returning {}", n);
    n
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Join runs multiple tasks concurrently and returns when they all
    // complete execution.
    join!(count_and_wait(1), count_and_wait(2), count_and_wait(3));
    Ok(())
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program will display the following output:

```
Starting 1
Returning 1
Starting 2
Returning 2
Starting 3
Returning 3
```

Discussion

The outcome is surprising because the `join` macro promises to run the three instances of `count_and_wait` concurrently, but the output shows that the tasks are running sequentially, which tends to surprise newcomers to Rust's `async` system. Understanding the differences between asynchronous and thread programming can help you avoid pitfalls—and help you pick the right model for your program.

Asynchronous programs and multithreaded programs operate differently, each with their own strengths and weaknesses. Asynchronous (`Future`-based) tasks aren't the same as threaded tasks, and they require some care to ensure that they operate concurrently. However, it's entirely possible to run an asynchronous program on one thread.

The diagram on page 9 shows the basic differences between threaded and asynchronous execution:

In a *threaded* model, each task operates inside a full operating system-supported thread. Threads are scheduled independently of other threads and processes. An *asynchronous* model stores tasks in a task queue and runs them until they *yield* control back to the executing program.

Let's examine a few approaches to running this teaser concurrently.

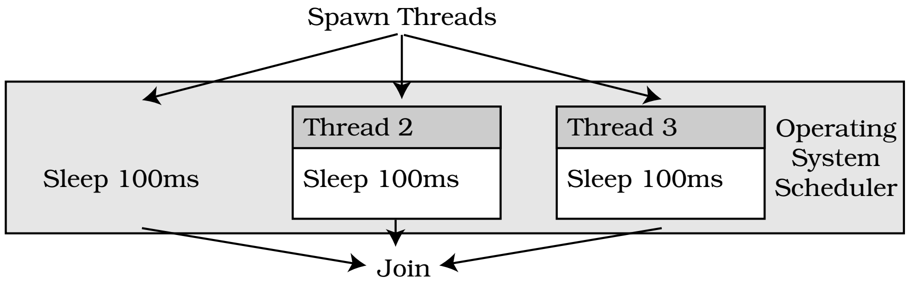
Native Threads

Threads are *preemptively* scheduled by your operating system. While the thread is suspended, other threads continue to run. A purely threaded version of this teaser looks like this:

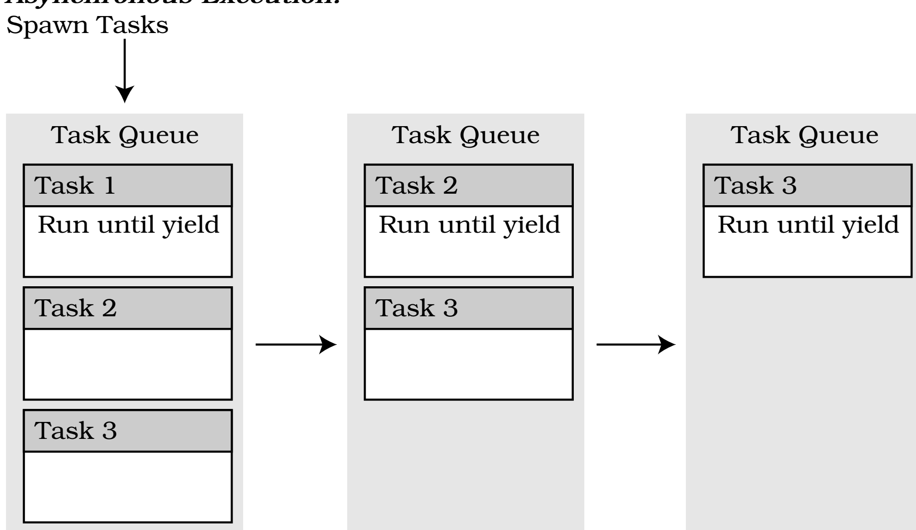
async_threaded/src/main.rs

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

Threaded Execution:



Asynchronous Execution:




```rust
fn count_and_wait(n: u64) -> u64 {
    println!("Starting {}", n);
    std::thread::sleep(Duration::from_millis(n * 100));
    println!("Returning {}", n);
    n
}

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let a = thread::spawn(|| count_and_wait(1));
    let b = thread::spawn(|| count_and_wait(2));
    let c = thread::spawn(|| count_and_wait(3));
    a.join().unwrap();
    b.join().unwrap();
    c.join().unwrap();
    Ok(())
}
```

The program spawns three threads, and they each run concurrently. Because the program calls `sleep` and delays execution on each thread, you're almost—subject to having a *really* busy computer—sure to see the following output:

```
Starting 1
Starting 2
Starting 3
Returning 1
Returning 2
Returning 3
```

Threads provide excellent concurrency, but it comes at a cost. Threads have their own context maintained by the operating system. Starting a thread requires a system call, which can be slow if you need to make *many* threads. Different operating systems have varying limitations, but there's a hard limit to the number of threads you can create—and your OS is generally not designed to schedule thousands of threads at a time. Native thread syntax can also be clunkier than an equivalent async `join` or `await` call.

Threads start running as soon as you call `Thread::spawn`. The thread then runs—scheduled by the operating system—until it's done or sent a termination signal.

Asynchronous Tasks

Asynchronous tasks are *cooperatively scheduled*. The operating system doesn't intervene to ensure that each thread gets a fair allocation of execution time. Tasks run until they `yield` control. Yielding returns control to the executor—the code responsible for maintaining the async environment. Tasks yield when:

- The task returns a result (either an error message or a value).
- The task completes execution.
- The task `awaits` one or more tasks.
- The task explicitly calls `yield_now()`, suspending itself until the executor resumes it.

Used correctly, asynchronous task-based code can provide fantastic performance. This is especially true for I/O bound programs—programs that have to wait for databases, files on disk, or other processes to complete. Lightweight tasks send requests to the other systems and `await` a result. Each task queue can then keep processing requests *very* fast, executing tasks only when the requested data is ready for them.

What Is an Executor?

Rust's `async` implementation provides everything you need to make an asynchronous environment, but it only provides the functionality required to implement an `executor`. The executor is responsible for tracking spawned tasks, executing them, and providing services such as `yield`.

`Tokio` is one of the most popular executors, providing a "batteries-included" system with functionality available for most common tasks. The `std-async` and `futures` crates are also popular. If you need specific functionality, you can also write your own executor.

Many executors allocate tasks to queues in a group of threads, but they don't have to. Most schedule multiple tasks per thread—known as `M:N` green threading—but an async setup can be entirely single-threaded.

Other platforms use this paradigm as well. NodeJS, Erlang/Elixir, and various .NET systems provide similar functionality.

As it turns out, asynchronous tasks only provide outstanding performance if you play by their rules and avoid any *blocking* calls. Blocking calls suspend process execution and resume when the call is complete. Furthermore, blocking calls don't yield control back to the executor—a call to `Thread::sleep` suspends the entire thread's execution, *including* the executor. That's why the example program runs serially, even though the `join` macro promises concurrency.

For the common task of sleeping, Tokio provides a safe, nonblocking call to make a task pause for the specified time. Replace `Thread::sleep` the `count_and_wait` function with the following code:

```
tokio::time::sleep(Duration::from_millis(n*100)).await;
```

Run the program, and you'll see the same output as the threaded version, meaning your program ran concurrently.

Asynchronous Blocking Tasks

Sometimes, you *need* to block execution; for example, when you have a long-running task, need to communicate with some hardware that doesn't provide an `async` friendly code wrapper, or have to use another library. `tokio` provides a function for these situations that won't stall the execution pipeline:

```
let blocking_task = tokio::spawn_blocking(|| {
  // Do something really slow and blocking here
});

// Run the task
blocking_task.await.unwrap();
```

The spawn_blocking code tells tokio that your task will block, and tokio will spawn it inside its own thread, suspending the current task until the thread returns. Your task runs in the background, and your executor can keep processing other tasks. Notice that the blocking task still awaits a return; Tokio will awaken the parent task when the blocking task completes.

Long-Running Asynchronous Tasks

Occasionally, you need to perform some heavy computation inside your async task. A task may call yield_now at any time to suspend operation and let other tasks run. When the scheduler returns to the task, it'll continue where it left off. For example, have a look at this code:

```
async fn my_big_task() {
  for i in 0..1_000_000 {
    // Do something intensive with i
    tokio::task::yield_now();
  }
}
```

This task will yield control back to the executor after each calculation, which reduces the stalling effects of your heavy calculation without creating a thread.

Choosing Threaded or Asynchronous Operation

tokio and other systems provide an async version of the more common operations that require input/output. Reading and writing files, connections to databases, and even logging are available in executor-friendly formats. Task-based asynchronous code can be amazingly fast for programs that frequently have to wait for another system. Web and other servers often benefit significantly from a task-based structure and provide very high throughput.

Threads are more appropriate for CPU-bound tasks and tasks that *must* block. Threads incur their own overhead, but if the threaded task is sufficiently "heavy" in terms of CPU load, they can outperform asynchronous task-based systems. In the embedded world, or when writing performance-critical code, you often want to favor threads because you can control their scheduling properties (and pin them to individual CPUs)—providing much more of a guarantee of execution time.

Rayon: Task-Based Threading

Rayon is a popular Rust crate that implements task-based threading. Rayon creates a pool of threads that sit idly, waiting to be given work. When you create a Rayon task, the next available thread executes it. The task executes independently and doesn't stall the pipeline when you make a blocking call. Rayon can provide the best of both worlds for CPU-heavy tasks—task-based syntax, easier management, and lower overhead.

Rayon performs very well but is still frequently outperformed on input/output bound server tasks by a more traditional asynchronous setup. Of course, you can mix the two, but you'll have to pay attention to the size of your worker thread pools to ensure that your executor isn't starved of CPU time.

Further Reading

Asynchronous Programming in Rust
: https://rust-lang.github.io/async-book/01_getting_started/01_chapter.html

Rust Futures
: https://github.com/rust-lang/futures-rs

Tokio
: https://github.com/rayon-rs/rayon

Async-Std
: https://github.com/async-rs/async-std

Rayon
: https://github.com/rayon-rs/rayon