Week 08 Weekly Exercises

Objectives

- Use basic rust threads
- Understand how rust provides fearless concurrency
- Apply ownership semantics

Activities To Be Completed

The following is a list of all the markedactivities available to complete this week...

- **Train Game**
- **Channels!**
- **WebServer**
- **Send and Sync**

The following practice activities are optional and are not **marked, or required** to be completed for the week.

- **Obelisk - using web frameworks**

Preparation

Before attempting the weekly exercises you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this week's exercises called `lab08`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab08
$ cd lab08
$ 6991 fetch lab 08
```

Or, if you're not working on CSE, you can download the provided code as a tar file.

EXERCISE:
Train Game

NOTE:

Objectives:
- A first program using threading
- Using thread::scope and thread::spawn
- Understand the concept of breaking down calculations into smaller chunks
- Exposure to the useful itertools crate

In Sydney, all train carraiges have an identifiable four digit number.

For example, in the above image, the train carriage number is 3592.

A popular game amongst Sydney train passengers is to find some arrangement of the numbers, and four mathematical operators (+, -, x, /), that will give a result of 10.

For example, in the above image, the numbers 3 5 9 2 can be arranged as

```
3 x 2 - 5 + 9 = 10
```

In this exercise, you are given starter code that currently takes in a **FIVE** digit number, and prints out all possible arrangements of the numbers, and four mathematical operators (+, -, x, /), that will give a result of 10.

It does this by generating all possible combinations into a vec of tuples. Each tuple is a combination of the numbers and operators: `(<Vec<i32>, Vec<char>>)`, where the first element is a vector of the digits, and the second element is a vector of the operators.

It then iterates through the vec of tuples, evaluates the expression **left to right, without order of operations**, and prints out the expression if it evaluates to 10. **Your task** is to modify the code to take advantage of rust's fearless concurrency to speed up the program.

To do this, you will need to roughly follow the following steps:

1. Chunk the vec of tuples into a vec of vecs of tuples
2. Create a new thread scope
3. For each chunk (a singular vec of tuples) spawn a new thread (inside the scope)
4. Have each thread iterate through its chunk, and evaluate the expression
5. Have each thread print out the expression if it evaluates to 10

> HINT:
>
> This type of work is usually referred to as "data parallelism". In the rust ecosystem, there are crates such as rayon that are usually used for this type of work.
>
> In this exercise, you will be implenting this from scratch.
>
> The documentation for the `scope` function may be useful.
>
> The documentation for the `spawn` function may be useful.
>
> The documentation for the `chunks` function may be useful.
>
> The documentation for the `join` function may be useful.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 9 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

Channels!

In order to get started, you will need to create a Cargo project

You can do this by running the `cargo new` command in your terminal

```
$ 6991 cargo new channels
Created binary (application) `channels` package
```

NOTE:

Objectives:

- Understand the concept of channels
- Practically use channels
- Exposure to move closures

There is no starter code for this exercise, as you should use use your solution from the previous exercise, as explained below.

Last exercise, we refactored a single threaded set of calculations to use multiple threads. Our threads all printed out the specific calculations that they were able to find that met the requirements, but, we have no way of knowing how many calculations each thread found!

In this exercise you will be making use of Rust's channels to communicate between workers.

To do this, it is assumed that you have completed the previous exercise, and have a working solution that:

- chunks the work into a set of smaller vectors
- creates a thread scope
- spawns a thread for each sub task/vector
- does the calculation (and prints) in each thread

NOTE:

Before you start, copy your solution from the previous exercise, and add the itertools dependency to your Cargo.toml file.

```
$ cp train_game/src/{test,main}.rs channels/src/
$ 6991 cargo add itertools
```

You should modify your code, such that:

- Before the new thread scope is created, you create a channel
- Each thread sends the number of calculations it found to the channel
- After all the calculations are done, you receive the number of calculations from each thread, and print out the total number of calculations

Your output should look something like this:

```
$ 6991 cargo run -- 12345
cr -- 12345
   Compiling either v1.8.0
   Compiling itertools v0.10.5
   Compiling channels v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 1.74s
     Running `target/debug/channels 12345`
There are 2880 potential combinations
2 / 4 + 3 - 1 * 5 = 10
// -- CUT FOR BREVITY, YOUR PROGRAM WILL OUTPUT MORE --
2 * 4 - 3 + 5 / 1 = 10
Thread 0 found 5 combinations
Thread 3 found 11 combinations
Thread 4 found 19 combinations
Thread 2 found 4 combinations
Thread 5 found 9 combinations
Thread 1 found 11 combinations
Total: 59
```

HINT:

It may help to change the return type of `calculate` to indicate whether or not a calculation was found for a combination of numbers. and operators.

This will allow you to have a running "count" of how many solutions have been found, and at the end of the thread's execution, you can send back the thread_id and count to the channel.

The `Shared usage` example in the `mspc` documentation will be quite useful.

You should not need any shared state primatives such as `Arc` or `Mutex` for this exercise.

You may find the Rust Book chapter on message passing to be useful: Message Passing

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 9 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

WebServer

NOTE:

Objectives:

- Understand the concept of shared state and what types Rust provides for it
- Manipulate bytes, threads and TcpStreams
- Understand the lifecycle of a typical web server

This exercise does **NOT** require any networking knowledge.

You have been given a simple web server - a piece of code that listens to a port for web connections, and serves back a simple HTML page. This html page two major features:

- Some text that looks exactly like: `{{{ counter }}}`
- A button that will send a POST request to `/counter`

The server will be started by running:

```
$ 6991 cargo run -- 12345
Server running on port 12345
```

This will start the server on port 12345. You can then connect to it with your browser by visiting:

```
$ http://localhost:12345
```

The server will be listening for connections on that port, and will respond to requests with the HTML page. The HTML page looks something similar to this:

```
$ http://localhost:12345
```

Hello World!!

{{{ counter }}}

increment

You will notice that the text in the form is currently set to `{{{ counter }}}`. This is a placeholder, and should be replaced with the actual value of the counter, which will be 0 initially.

> DANGER:
>
> DO NOT edit the HTML file.
> Instead, manipulate the string/bytes object to replace the placeholder with the actual value of the counter.

When you click the button, the server will receive a POST request to `/counter`. This currently does nothing! You should make it so that the server increments the counter by one, and then returns the new HTML page with the new value of the counter.

Your task is to complete the implementation of the server, such that it will:

- Spawn a thread per connection
- Make `State` shared across threads, by using a locking primitive
- Increment the counter in state when a POST request is received
- Replace the {{{ counter }}} placeholder with the actual value of the counter

To test your code, you can run the server, and then visit the page in your browser. Marks will be determined based off `6991 cargo test` correctly running. The given `6991 autotest` will simply call `6991 cargo test`.

> NOTE:
>
> **If you see the following error when testing**
>
> `Error: Os { code: 98, kind: AddrInUse, message: "Address already in use" }`
>
> This means that the server is already running on the port you are trying to use. You can either kill the server, or change the port number.
>
> If your browser tests work fine, but you cannot get `6991 cargo test` to pass despite changing port numbers, etc - please submit anyways, have faith in your work :D
>
> There is no give dryrun.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 9 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

Send and Sync

> DANGER:

This exercise tests your understanding of the Send and Sync traits, and how Rust provides "Fearless Concurrency".
Please note this exercise is **worth marks, but not manually marked** as there will not be reasonable time for teaching staff to mark in a future workshop, or provide written feedback.

However, submissions will still be accepted, and any submission with some answers will obtain full marks (despite correctness). It is expected that you would be able to answer the below questions, and would be typical of a Concurrency Theory question you might be asked in the final exam.

You have been asked by a colleague to explain how Rust provides static guarantees about Concurrency.

They know some basic rust (equiv to up to week 7 of the course), but are not familiar with the details of the Send and Sync traits (etc).

Your task is to write answers to their questions, which can be found in the starter code `questions.md`.

You will find this useful as revision for the final exam.

HINT:

You may find this video useful for explaining some of the concepts. Alongside the course lectures :)

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs6991 lab08_send_sync questions.md
```

The due date for this exercise is **Week 9 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

(OPTIONAL) EXERCISE:

Obelisk - using web frameworks

NOTE:

The goals of this exercise are to:

- Learn about common web server frameworks in Rust
- Learn how invariants on types can be used to ensure correctness of our webserver This optional exercise was added after someone asked what alternatives to common web server frameworks (e.g. expressJS or flask) there are in Rust.

In this exercise, you will be using the `axum` crate to build a simple web server.

The starter code has two simple endpoints, designed to show off some of the advantages of building web servers in Rust, specifically around type safety and compile-time error checking.

The first endpoint is `GET /ping`, which returns a simple string `"pong"`.

The second endpoint is `POST /users`, which expects a JSON body with a single field, `username`, and returns a JSON response with the `username` and some `id`.

These are represented by the `CreateUser` and `User` structs in the starter code! The `axum` framework allows the function for that route to use these structs as parameters and return types, and returns a `400` status code for bad arguments, if the input cannot be transformed into those structs.

Your task is to add a third endpoint, `GET /hello/{name}`, which returns a string `"Hello, {name}!"`.

You can test your code by running `RUST_LOG=info 6991 cargo run` and then running the following commands in a separate terminal:

```
$ curl https://localhost:3000/hello/shrey
```

There are no autotests for this exercise.

The book Zero2Prod is a great book that highlights the benefits of building web servers in Rust, and goes through the large process of building a production-ready web server in Rust, including testing, logging, deployment and more.

Submission

When you are finished each exercise make sure you submit your work by running `give`.

You can run `give` multiple times.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

The due date for this week's exercises is **Week 9 Wednesday 21:00:00**.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

Automarking will be run continuously throughout the term, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run you can view your results here or by running this command on a CSE machine:

```
$ 6991 classrun -sturec
```
