Introduction

In this assignment, you'll write a program that will get you familiar with writing multi-threaded programs in Rust.

Learning Outcomes

- Write simple programs in Rust (Module 9, MLO 2)
- Compile, debug, manage and run Rust programs (Module 9, MLO 3)
- Explain the facility for threads provided by Rust (Module 10, MLO 2)

Map Reduce

According to the Wikipedia article on MapReduce

(Links to an external site.)

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

A MapReduce program is composed of a map procedure, which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name), and a reduce method, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies).

Instructions

For this assignment, we are providing you with a single-threaded Rust program

(Links to an external site.)

that processes input numbers to produce a sum. The program contains extensive comments that explain the functionality and give directions on what parts of code you are allowed to change (look for comments starting with CHANGE CODE).

Here is a description of the program: currently the `main()` function does the following

- Generates data for the rest of the program
    - Calls `generate_data()` to generates a vector of numbers that serves as input for the rest of the program.
- Partitions the data
    - Calls `partition_data_in_two()` which partitions the input numbers into two partitions
- Performs the map step
    - Calls `map_data()` for each of the two partitions, which returns the sum of all the numbers in that partition.
- Performs the reduce step
    - Gathers the intermediate results produced by each call to `map_data()`
    - Calls `reduce_data()` that sums up the intermediate results produced by the map step to produce the final sum of all the input numbers.

You have to modify the program to accomplish the following tasks:

- Modify the program to create 2 threads, each of which concurrently runs the `map_data()` function on one of the two partitions created by the program given to you.
- Add code for the function `partition_data()` to partition the data into equal-sized partitions based on the argument `num_partitions`
    - In case num_elements is not a multiple of num_partitions, some partitions can have one more element than other partitions
- Add code to the function `main()` to
    - Partition the data into equal-size partitions

    - Create as many threads as the number of partitions and have each thread concurrently run the `map_data()` function to process one partition each
    - Gather the intermediate results returned by each thread
    - Run the reduce step to process the intermediate results and produce the final result

See detailed comments in the provided program to see how you can go about making the required changes.

Example Usage

Here are some example executions of the program.

An execution of the program with 5 partitions and 150 elements.

- Since the number of elements is a multiple of the number of partitions, it is required that each partition should have the same number of elements.
- However, there is no requirement about which element is put into which partition. Thus the intermediate sums in your solution can be different from what is shown below.

```
./main 5 150
```

```
Number of partitions = 2
```

```
   size of partition 0 = 75
```

```
   size of partition 1 = 75
```

```
Intermediate sums = [2775, 8400]
```

```
Sum = 11175

Number of partitions = 5

   size of partition 0 = 30

   size of partition 1 = 30

   size of partition 2 = 30

   size of partition 3 = 30

   size of partition 4 = 30

Intermediate sums = [435, 1335, 2235, 3135, 4035]

Sum = 11175
```

An execution of the program with 6 partitions and 150 elements.

```
./main 6 150

Number of partitions = 2

   size of partition 0 = 75

   size of partition 1 = 75

Intermediate sums = [2775, 8400]
```

```
Sum = 11175

Number of partitions = 6

   size of partition 0 = 25

   size of partition 1 = 25

   size of partition 2 = 25

   size of partition 3 = 25

   size of partition 4 = 25

   size of partition 5 = 25

Intermediate sums = [300, 925, 1550, 2175, 2800, 3425]

Sum = 11175
```

An execution of the program with 5 partitions and 153 elements.

- In this example the number of elements is not a multiple of the number of partitions.
- Based on the requirement that some partitions can have one element more than other partitions, in this case 3 partitions must have 31 elements and 2 partitions must have 30 elements.
- In the example shown below, the 3 partitions with 31 elements are at position 0, 1 and 2 in the vector of partitions, and the 2 partitions with 30 elements are at position 3 and 4 in that vector. However, there is no requirement about the order in which partitions that have one more element than other partitions

appear in the vector of partitions. Thus, the order of the partitions in your solution can be different from what is shown below.

```
./main 5 153

Number of partitions = 2

   size of partition 0 = 76

   size of partition 1 = 77

Intermediate sums = [2850, 8778]

Sum = 11628

Number of partitions = 5

   size of partition 0 = 31

   size of partition 1 = 31

   size of partition 2 = 31

   size of partition 3 = 30

   size of partition 4 = 30

Intermediate sums = [585, 1486, 2387, 3135, 4035]

Sum = 11628
```

Hints

- The function thread::spawn()
- (Links to an external site.)
- returns `JoinHandle<T>` where T is the type of the return value of the function the thread runs. This means that
    - Because `map_data()` returns an integer of type `usize`
    - If you spawn a thread that runs `map_data()`
    - `thread::spawn()` will return a value of type `JoinHandle<usize>`

What to turn in?

- Required: Upload one file `main.rs` with all of your code.
    - When you resubmit a file in Canvas, Canvas can attach a suffix to the file, e.g., the file name may become `main-1.rs`. Don't worry about this name change as no points will be deducted because of this.
- Optional: If you have any meta-comments about the program, create a file README.txt with these comments, and upload it with your submission as a separate file (i.e., don't zip up the two files together).

Grading Criteria

- This assignment is worth 8% of your final grade. The breakup of points is given in the grading rubric.
- The grading will be done on os1.
- To test your program, we will compile the code as follows

```
rustc main.rs
```

- We will run the program as follows

`./main num_partitions num_elements`

E.g.,

`./main 5 150`