CS 214 Extra Credit 4:
Log Processing in Rust (100 points)

The goal of this assignment is to explore the Rust programming language. Recall Rust was developed as a memory-safe language without garbage collection. It thus requires manual memory management although with far greater safety features than C. However, these mandatory safety features come with a cost — the Rust programmer must navigate a much more complex type system, ownership and liveness models which have greater complexity than in simpler languages like C or Python.

For this project, you will write a multi-threaded program that reads in a log file and prints lines which contain a string given as a command line argument. The program should become faster as more threads are added.

How run Rust on the ilabs:

Follow the instructions below on how to run the Rust compiler, rustc, on the ilabs. **Do not use cargo** as it is a general package manager and build tool designed for larger, more complex projects. As this assignment is a simple program, cargo only adds unnecessary complexity – make sure the program compiles only using `rustc`.

https://resources.cs.rutgers.edu/docs/new-users/beginners-info/running-rust-compiler-on-cs-linux-machines/

Guidelines:

It is best to gradually build a small program up from a working kernel than try to develop the entire program. One approach is to open a file and search each line for a matching string. This works well in single threaded mode. However, to allow multiple threads to operate on different parts of the program, some other approach must be used.

There are two different approaches to multithreading: The first reads all the lines into a data structure and then has different threads operate on different parts of the data structure. For example, a vector of strings could be partitioned to different threads.

The second approach doesn't try to read the lines into a data structure, but rather finds indexes in the file where different threads will search for the matching keyword. For example, the program could start by dividing up a file of size N bytes with T threads into byte indexes at N/T boundaries, and then adjust the boundaries to match where the line breaks are. The first approach is more straightforward and works with standard libraries, but has a large sequential component.

Here are some smaller tasks with links to get you started along with links to example code:

Compile and run a Rust hello world program:
https://doc.rust-lang.org/book/ch01-02-hello-world.html

Read in environment variables:
https://doc.rust-lang.org/book/ch12-05-working-with-environment-variables.html
Also:
https://glenngillen.com/learning-rust/environment-variables/

Open a file and read it line-by-line:
https://doc.rust-lang.org/rust-by-example/std_misc/file/read_lines.html
Also:
https://stackoverflow.com/questions/45882329/read-large-files-line-by-line-in-rust

Using Rust vectors:
https://doc.rust-lang.org/rust-by-example/std/vec.html
and:
https://www.programiz.com/rust/vector

Measuring time and using timestamps:
https://doc.rust-lang.org/std/time/struct.Instant.html
and:
https://stackoverflow.com/questions/13322479/how-to-benchmark-programs-in-rust

Spawning Threads:
https://doc.rust-lang.org/std/thread/fn.spawn.html

Splicing vectors into separate pieces:
https://doc.rust-lang.org/std/primitive.slice.html#method.split_at_mut
and:
https://www.reddit.com/r/learnrust/comments/138k81f/concurrent_access_to_disjoint_elements_of_a/

Here are some example libraries you could use for the above tasks: :

```
use std::env;           // get the environment variables
use std::path::Path;    // open a file
use std::fs::File;      // read a file
use std::io::{self, BufRead};
use std::vec::Vec;      // store lines in a variable sized vectors
use std::time::Instant;     // get a timestamp for time measurement
use std::thread;        // create threads
```

What to hand in:

Upload to gradescope a single Rust source file called “logprocess.rs”. The program must take in 3 arguments: the filename to process, a string to match for each line of the file, and a number of threads.

After opening the file, the program should print out all lines which contain the string to match as substring in the file. After all the matching lines are printed, the program should report the number of nanoseconds from when the file is opened to when the last string is printed out. The last line should be the timing report as:

<file name> processed in <S> nanoseconds

You can work in a team of up to three people.

Example inputs and outputs:

2 example input files, a search term per input file, and the outputs are provided in the canvas directory with this extra credit assignment.