ECE 459: Programming for Performance
Lab 2—Channels and Shared Memory[1]

Patrick Lam & Jeff Zarnett
with acknowledgement and thanks to Douglas Harder and Stephen Li
updated by Bernie Roehl, December 2020
Due: February 18, 2022 at 11:59 PM Eastern Time

Learning Objectives:

- Become familiar with message-passing for communicating between threads; and,
- Become familiar with the use of shared memory.

The core content for this lab is in Lecture 3, though there is somewhat-relevant content up to Lectures 16 and 17. You can also see a tutorial video from 2021 at https://www.youtube.com/watch?v=b3YDPsX6e-8.

Background

In this lab, you'll be cracking JWT signatures. JWT stands for JSON Web Token, which is (from the Wikipedia page) "an Internet standard for creating data with optional signature and/or optional encryption whose payload holds JSON that asserts some number of claims". Basically, it's a string consisting of a header, a payload and a signature. The three parts are separated by dots, and each of the three parts is encoded in base 64. Base 64 is an encoding that takes arbitrary binary data and converts it to a text format, and it's typically used in URLs. Many web applications use JWTs for user authentication.
Here is a typical JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dnZWRJbkFzIjoiYWRtaW4iLCJpYXQiOjE0MjI3Nzk2Mzh9.gzSraSYS8EXBxLN_oWn
FSRgCzcmJmMjLiuyu5CSpyHI
```

Specifically:

- header = "`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ`";
- payload = "`eyJsb2dnZWRJbkFzIjoiYWRtaW4iLCJpYXQiOjE0MjI3Nzk2Mzh9`";
- signature = "`gzSraSYS8EXBxLN_oWnFSRgCzcmJmMjLiuyu5CSpyHI`".

The signature is used to verify that the header and payload have not been tampered with. The header and the payload are combined with a value called the "secret", and then hashed using HMAC-SHA256 to produce the signature. Specifically, the signature defined as:

HMAC-SHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

[1] v0, 28Jan22

The “secret” is what prevents malicious actors from generating fake JWT. Your task is to write a program that brute forces a JWT’s secret.

There are three positional command-line arguments to your program. The first is the token, the second is the maximum possible length of the secret (it could be shorter), and the third is the alphabet (i.e. the set of characters that might be used in the secret).

You need to extract the signature portion from the token, and then use a brute-force approach to find the secret. You do this by exploring all possible values for the secret until one is found that produces the correct signature. The possible values for the secret are constrained by the key length and the alphabet. In a real-world situation, the problem is intractable because the alphabet is larger and the key length is longer (at least 32 characters). The examples used in this lab are chosen to have a short key length (four or five characters) and an alphabet consisting of the lowercase letters plus the digits 0 through 9.

Please don’t try to use this code on UW hardware to crack anything other than some test data for this exercise. The test data is sufficiently small that you can crack it in a few minutes; if it’s taking much longer than that, something is wrong.

We provide starter code that consists of a single-threaded implementation in `main.rs`. Instructions for running it can be found in the README file. Your job is to modify the code to improve its performance by using multiple threads.

In some cases, this is straightforward since the tasks are largely independent of each other. However, in this case things are more complex since the threads need to be able to communicate with each other. In particular, you want the threads to end once a solution has been found rather than exploring the entire solution space.

The two approaches you will be using are message passing and shared memory. The files `message-passing.rs` and `shared-mem.rs` are initially just copies of `main.rs`, and you’re expected to modify them to use message-passing and inter-thread communication respectively.

You’re free to implement things as you see fit. You may want to look at the Crossbeam crate, since it will do a lot of the work for you. In particular, you can use `crossbeam::scope` for shared memory. Just don’t use channels. You can use other crates as long as you don’t trivialize the problem.

For message passing, you are expected to use channels. There are multiple ways to do this, but we recommend that you start by looking at the unbounded channels provided by Crossbeam. Note that you should rely only on channels. You should not (and should not need to) make direct use of mutexes, atomics, or other shared state primitives. The channels do all this work for you.

By the same token (so to speak. . .), in the shared memory version you should not (and should not need to) use channels.

Public service announcement. Please be sure to clean up after yourself and not leave long-running tasks hanging around and hogging all the CPUs.

Rubric

The general principle is that correct solutions earn full marks. However, it is your responsibility to demonstrate to the TA that your solution is correct. Well-designed, clean solutions are therefore more likely to be recognized as correct.

Solutions that do not compile will earn at most 39% of the available marks for that part. Segfaulting or otherwise crashing solutions earn at most 49% .

The mark breakdown is as follows:

- Message-passing solution (40 marks).
- Shared-memory solution (40 marks).
- Written report (20 marks):
  - 8 marks for discussion of the message-passing solution;
  - 8 marks for discussion of the shared-memory solution; and
  - 4 marks for clarity.

Clarifications

Do I have to do the impossible? No. We will always test your code with solvable test cases.

Is there a performance target? No specific target, though faster is better. We're just marking your use of the required idioms (message-passing and shared memory). We'd be sad if your parallel solution was slower than the sequential code, but we wouldn't take it out on you.

What do you mean by shared memory? Our definition of shared memory means that you have a mutable variable which is accessed by multiple threads protected by locks and are not using the Rust channels. What you do with this mutable variable is up to you. (You can also use lock-free data structures).

Do I have to stop immediately once I find a solution? A small delay is fine, but you do need to communicate the stopping condition to all threads.

Valgrind/Helgrind complain at me. Yeah, it happens that sometimes it's not your fault (e.g. library code). Obviously, in this class, you are only responsible for things that are your fault.

How many threads? You are not required to support variable numbers of threads (though it's not a bad idea to do that experiment). The recommended value is the number of CPUs on the machine (value returned by `num_cpus::get()`).

How many possible solutions exist? The number of possible solutions will fit in a `u64`.

How big is the alphabet? It can be smaller than the number of CPUs.

Do we care about file layout (e.g. modules)? No, go ahead and modularize if you want, as long as it compiles and runs using the default commands.

What am I required to put in the report? Just a clear explanation of your implementation. We do appreciate benchmark results, though. If there's any explaining to do for the TAs about why you know that Helgrind reported errors are not a real problem, then that's sensible in that it might prevent having to debate correctness with a TA.

I'm still worried about Helgrind errors. To make it clear: if you are sure that the Helgrind errors are from library code, they're not a problem. tokio semaphores and Crossbeam channels are known to make Helgrind upset. Also, it's safe to share Crossbeam channels by reference across threads (that's why the compiler allows it), so you don't need to worry about channel-related race conditions specifically.