Exercise 1 : Concurrency Essentials

1: If you are not on the lab

This exercise does not require that you use the machines at the real-time lab. However, you will need some way to run code written in C and Go. Here are some alternatives if you are not on the lab:

- C:
  - Windows: Use TDM-GCC (https://jmeubank.github.io/tdm-gcc/download/), then use `gcc` from the command line or powershell.
  - OSX: You will need to download the Apple Developer Tools (https://developer.apple.com/xcode/), then either use xcode, or `gcc` or `cc` from the command line.
- Go:
  - Download from golang.org (https://golang.org/)

Go has an interactive tour (http://tour.golang.org/list) you can take. Go's syntax is a bit different, so it may be worth skimming through, or at least using as a quick reference.

2: Version control

A version control system is a tool that helps a group of people work on the same files in a systematic and safe manner, allowing multiple users to make changes to the same file and merge the changes later. It is also possible to create diverging branches so that independent areas of development can happen in parallel, and then have these merged safely later. Version control systems also keep track of all previous versions of files, so that you can revert any or all changes made since a given date.

In this course, we will not be using a GitHub Classroom (as has been done in previous years), but you are encouraged to use git or some other version control system. Unless you're already familiar with git, it's highly recommended to have a look at the following resources before moving on. Don't let the feeling that you have to google everything discourage you, this is perfectly fine, even expected. And don't forget that the student assistants are there to help you.

- Do the interactive tutorial (https://try.github.io/)
- Feature branch workflow (https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow)

Some prefer the command line while some prefer something graphical, both are fine. An overview of graphical git clients can be found on https://git-scm.com/downloads/guis/. Some of these have already been installed on the lab computers, but feel free to install whatever you prefer.

3: Sharing a variable

Implement this in C and Go:

```
global shared int i = 0
```

```
main:
    spawn thread_1
    spawn thread_2
    join all threads (or wait for them to finish)
    print i

thread_1:
    do 1_000_000 times:
        i++
thread_2:
    do 1_000_000 times:
        i--
```

There is some starter code in the folder "shared variable". Fill out the missing code and run the programs.

Create a new file called `result.md` inside this directory explaining what happens, and why (Hint: the result should not always be zero...). Then add, commit, and push the updated code and the results file, and verify that you can see the updated version on the web.

4: Sharing a variable, but properly

Modify the code from the previous part such that the final result is always zero.

In your solution, make sure that the two threads intermingle, and don't just run one after the other. Running them sequentially would somewhat defeat the purpose of using multiple threads (at least for real-world applications more interesting than this toy example).

It may be useful to change the number of iterations in one of the threads, such that the expected final result is not zero (say, -1). This way it is easier to see that your solution actually works, and isn't just printing the initial value after doing nothing.

C

- POSIX has both mutexes (`pthread_mutex_t` (http://pubs.opengroup.org/onlinepubs/7990989775/xsh/pthread.h.html)) and semaphores (`sem_t` (http://pubs.opengroup.org/onlinepubs/7990989775/xsh/semaphore.h.html)). Which one should you use? Add a comment (anywhere in the C file, or in the `results.md` file) explaining why your choice is the correct one.
- Acquire the lock, do your work in the critical section, and release the lock.
- Reminder: Make sure that the threads get a chance to interleave their execution.

Go

Using shared variable synchronization is possible, but not the idiomatic approach in Go. You should instead create a "server" that is responsible for its own data, `select{}` (http://golang.org/ref/spec#Select_statements)s messages, and perform different actions on its data when it receives a corresponding message.

In this case, the data is the integer `i`, and the three actions it can perform are increment, decrement, and read (or "get"). Two other goroutines should send the increment and decrement requests to the number-

server, and `main` should read out the final value after these two goroutines are done.

Before attempting to do the exercise, it is recommended to have a look at the following chapters of the interactive go tutorial:

- Goroutines (https://tour.golang.org/concurrency/1)
- Channels (https://tour.golang.org/concurrency/2)
- Select (https://tour.golang.org/concurrency/5)

Remember from before where we had no good way of waiting for a goroutine to finish? Try sending a "finished"/"worker done" message from the workers back to main on a separate channel. If you use different channels for the two threads, you will have to use `select { /*case...*/ }` so that it doesn't matter what order they arrive in, but it is probably easier to have multiple senders on the same channel that is read twice by `main`.

Hint: you can "receive and discard" data from a channel by just doing `<-channelName`.

Commit and push your code changes to GitHub.

5: Bounded buffer

From the previous part, it may appear that message passing requires a lot more code to do the same work - so naturally, in this part the opposite will be the case. In the folder "bounded buffer" you will find the starting point for a *bounded buffer* problem.

The bounded buffer should work as follows:

- The `push` functionality should put one data item into the buffer - unless it is full, in which case it should block (think "pause" or "wait") until room becomes available.
- The `pop` functionality should return one data item, and block until one becomes available if necessary.

C

The actual buffer part is already provided (as a ring buffer, see `ringbuf.c` if you are interested, but you do not have to edit - or even look at - this file), and your task is to use semaphores and mutexes to complete the synchronization required to make this work with multiple threads. If you run it as-is, it should crash when the consumer tries to read from an empty buffer.

If you are working from home and need a C compiler online, try this link: https://repl.it/@klasbo/ScientificArcticInstance#main.c . It should be instanced with the full starter code.

The expected behavior (dependent on timing from the sleeps, so it may not be completely consistent):

```
[producer]: pushing 0
[producer]: pushing 1
[producer]: pushing 2
[producer]: pushing 3
[producer]: pushing 4
[producer]: pushing 5
```

```
[consumer]: 0
[consumer]: 1
[producer]: pushing 6
[consumer]: 2
[consumer]: 3
[producer]: pushing 7
[consumer]: 4
[consumer]: 5
[producer]: pushing 8
[consumer]: 6
[consumer]: 7
[producer]: pushing 9
   -- program terminates here(-ish) --
```

Go

Read the documentation for `make` (https://golang.org/pkg/builtin/#make) carefully. Hint: making a bounded buffer is one line of code.

Modify the starter code: Make a bounded buffer that can hold 5 elements, and use it in the producer and consumer.

The program will deadlock at the end (main is waiting forever - as it should, and the consumer is waiting for a channel no one is sending on). Since this is a toy example, don't worry about it. But if you have any plans on doing more work with the Go language, you should take a look at the error message and try to understand it, as it will help you debug any such problems in the future.

As usual - commit and push your changes to GitHub.

6: Some questions

The file "questions.md" contains a few questions regarding some of the concepts this exercise covers, as well as some broader engineering questions. Modify the file with your answers.

You do not need "perfect" or even complete answers to these questions. Feel free to ask the student assistants (even during the exercise approval process) to discuss any questions you get stuck on - you might find you learn more in less time this way.

Things to think about until next time

This part of the exercise is not for handing in, just for thinking about. Talk to other groups, assistants, or even people who have taken the course in previous years.

7: Thinking about elevators

The main problem of the project is to ensure that no orders are lost.

- What sub-problems do you think this consists of?
- What will you have to make in order to solve these problems?

Maybe try thinking about the happy case of the system:

- If we push the button one place, how do we make (preferably only) one elevator start moving?
- Once an elevator arrives, how do we inform the others that it is safe to clear that order?

Maybe try thinking about the worst-case (http://xkcd.com/748/) behavior of the system:

- What if the software controlling one of the elevators suddenly crashes?
- What if it doesn't crash, but hangs?
- What if a message between machines is lost?
- What if the network cable is suddenly disconnected? Then re-connected?
- What if the elevator car never arrives at its destination?

8: Thinking about languages

In the next exercises (the first of which being networking) and the project, you can use any language of your own choice. You are of course free to change your mind at any time - you do not have to do the exercises and the project in the same language (or even different parts of the same exercise, for that matter). But you may want to start doing some research already now.

Here are a few things you should consider:

- Think about how want to move data around (reading buttons, network, setting motor & lights, state machines, etc). Do you think in a shared-variable way or a message-passing way? Will you be using concurrency at all?
- How will you split into modules? Functions, objects, threads? Think about what modules you need, and how they need to interact. This is an iterative design process that will take you many tries to get "right".
- Does the language "look right"? Does the standard library feel comprehensive?
- While working on new sections on the project you'll want to avoid introducing bugs to the parts that already work properly. Does the language have a framework for making and running tests, or can you easily create one?
- Code analysis/debugging/IDE support?

Extra

9: Multithreading in other languages

This is an optional exercise. You are not recommended to do this for "completion" or "achievement points". You should only do it if you're interested in learning more about how different languages can protect against data races, or you're considering to use one of these languages in your project.

The languages exemplified below are chosen primarily based on their historical popularity among students who have taken this course previously. It should not be interpreted as "is well supported" or "is endorsed".

Erlang

Erlang disallows mutability of variables completely, a new state will instead be reached by calling into a different function (or the same function with different arguments). This means it will be impossible to solve

the "shared variables but properly" task with lock-based synchronization. Instead you will have to take the "go-channel" approach, and make a number-server.

These servers are so common in Erlang that they have been made an OTP design pattern (http://erlang.org/doc/design_principles/gen_server_concepts.html). To not obfuscate the Erlang code, this approach has not been taken in the starter code. Instead a program very similar to the Go solution has been made. If you are planning on doing more work with Erlang/Elixir, you may want to take a closer look at the generic server pattern.

Task: Complete the program and verify that the answer is 0.

Rust

Rust uses its type system and compile-time checks to make sure that no data races are possible. This is possible by using the marker traits (https://doc.rust-lang.org/std/marker/) `Send` and `sync` (https://doc.rust-lang.org/beta/nomicon/send-and-sync.html). A data type is `Send` if you're allowed to send the data to another thread. If a data type is not marked as `Send`, it generates a compile-time error if you try to send it between threads. Similarly for `Sync`, but sync data can be accessed from several threads at the same time, not just sent around between them.

The primitive integer types in rust are not "thread safe", and thus not `Sync`. But there is no reason they can't be sent between threads, so `Send` is implemented. Since Rust doesn't take a stance in which concurrency model to use (as long as you avoid undefined behavior) both the "channel" and "lock" solutions are possible.

A `Mutex` (https://doc.rust-lang.org/std/sync/struct.Mutex.html) takes something that is `Send` and makes it `Sync`, while `mpsc` (https://doc.rust-lang.org/std/sync/mpsc/index.html) allows you to create "channels" for data types that implement `Send`.

The lock based approach has been taken in the starter code, you are of course free to re-write it into the `mpsc` approach if you feel like it.