GH repo: https://github.com/evgerritz/cpsc429_labs/tree/1_p5
The rust_mymem module source is in rust_kmod/rust_mymem.rs
The test module source is in rust_kmod/mymem_test.rs

Part 5:

- Acknowledgements
    - Referenced the provided resources, including the rust-for-linux github documentation and the information on the kernel build system
    - In particular:
        - https://rust-for-linux.github.io/docs/kernel/sync/struct.Ref.html
        - https://rust-for-linux.github.io/docs/kernel/task/struct.Task.html
        - https://doc.rust-lang.org/rustc/command-line-arguments.html
- Overview
    - rust_kmod/mymem_test.rs and rust_kmod/rust_mymem.rs have been updated as per the assignment
    - rust_kmod/Makefile includes a similar process to that used by the Rust for Linux rust implementation to export functions from Rust programs so they are available to other parts of the kernel. In this case, our test module.
- Challenges
    - Using only the Rust for Linux libraries was tricky because many utilities have different names, and there is rather limited documentation on everything. For example, I probably wasted an hour trying to figure out how to use ARef, assuming this was the kernel's implementation of Arc. It was only when I saw on github that a recent commit changed the name of Ref to Arc that I guessed Ref was what I really want.
    - Many features in Rust's standard library are also just not available, such as waiting for a task created with spawn. I used the example for Task::spawn to see how they ensured all threads had terminated before trying to process the results; they solved this issue using a condition variable that causes all threads to wait at the end of function until they have all exited.
    - In general, I often found myself searching through the github repository to find examples of the rust functions, but these were often limited or nonexistent.
    - Additionally, figuring out the syntax used by the Kernel Build System to understand the rust Makefile was tricky when we were trying to solve very specific issues for which information was limited.
        - In particular, figuring out how to get the kernel build system to compile the C and Rust source files together as one module was hard, since I could

not find how this was done in the original Makefile, and had to do a lot of experimentation and searching around on my own to figure it out.

- Questions
    - **Why does the inserting order matter here?**
        - If we insert in the wrong order, we get the error “Unknown symbol in module.” This makes sense because at runtime, the test program will try to interact with the mymem module through the module’s interface, but these interactions are done using symbols that have not been loaded into memory, since the mymem module’s binary has not been loaded into the kernel. The test module might try to call read, only to find that the symbol doesn’t exist yet; it becomes available only when the binary corresponding to that function has been inserted into the kernel.
    - Repeat the same performance measurements you did at Part 2. **Is there any performance difference compared to the program running in the user space (Part 4)? Why?**
        - Yes the program with everything in the kernel is significantly faster, almost uniformly across different payload sizes. It intuitively makes sense that a more tightly coupled system that has more privileges and direct access to physical memory would have less overhead when compared to communication that takes place between two systems, one of which is completely untrusted. One role of the kernel is to protect from malicious user space programs, so it has to do more checking of userspace interactions, while our test module has full power so this checking can be skipped. Another source of increased overhead is due to the context switch that must occur when switching between kernel and userspace, as well as the fact that the kernel must copy data from userspace before using to ensure consistency.
    - Repeat the experiments you did for Part 3. **What difference does your Rust-based, all-inside-kernel implementation make regarding the data race problem you had to deal with in Parts 3 and 4? How does the Rust compiler help address the problem? How does having the test module in the kernel make any difference?**
        - I solved the data race problem for the all-inside-kernel implementation in a similar way as in the earlier parts. Rust again guides the user towards implementing thread-safe code with its ownership requirements. One difference with this version is that I needed to use a condition variable in place of the missing *thread*.join() method to atomically ensure that all threads have exited before returning. Additionally, now we have the mymem buffer in a mutex, so the functions in the module’s interface are guaranteed to be atomic with respect to the multiple kernel threads, which

was not the case with the original mymem buffer. After taking these measures, the test module is thread-safe and can consistently launch 50 worker threads each performing 150,000 read/writes across.

- With Rust for Linux, how does a Rust kernel module access functions and data structures from the C part of the kernel? Because the Linux Kernel Build system assumes a C-based environment, how does a kernel module (written in C or Rust) access functions and data structures exported by another Rust kernel module? We raised these questions in Part 4. Now it is time you answer them. Specifically, **what tool is used by Rust for Linux to generate "bindings" for a Rust kernel module to access the function and data structures from the C part of the kernel?**
  - Rust for Linux uses bindgen to generate the bindings from the C part of the kernel into Rust. This works because Rust can use C's calling convention and call C functions using an extern "C" block; this Rust wrapper around a non-Rust function is called a foreign function interface (FFI). Specifically, Rust for Linux uses bindgen on a file called "bindings_helper.h" which includes a bunch of the kernel's libraries for bindgen to convert into Rust bindings.
- **What tool(s) are used by Rust for Linux to generate "bindings" for a kernel module to access the function and data structures from a Rust kernel module?**
  - As discussed in the guide, any module that wants its methods to be available to other modules needs to call one of the EXPORT_SYMBOL_* macros from <linux/module.h>, so the symbol appears in Module.symvers. This needs to be done in C, so we need to automatically a C object file exporting all of the symbols from the Rust object file. Rust for Linux does this by reading all of the symbols in a Rust .o file using the nm program, which lists the symbols in the program, and then doing some stream processing to clean up the output and the exports_mymem_generated.h file. This is then included in a C source file ("exports.c") which is compiled into a .o file that is included with the module we want the bindings for. Below is my slightly edited version of Rust for Linux's custom Kbuild command; I added grep -v 'module' to remove the duplicated and unnecessary 'module_cleanup'and 'module_init' exports.
  -
```
quiet_cmd_exports = EXPORTS $@
      cmd_exports = \
    $(NM) -p --defined-only $< \
        | grep -E ' (T|R|D) ' | cut -d ' ' -f 3 \
        | xargs -Isymbol \
        echo 'EXPORT_SYMBOL_RUST_GPL(symbol);' | grep -v 'module' > $@
```

- Data/plot
  -

Size	Average Read Time (all kernel) (n=1000) (microseconds)	Average Write Time (all kernel) (n=1000) (microseconds)
1	0.063	0.064
64	0.065	0.065
1024	0.081	0.115
65536	1.735	1.935
524288	19.671	22.392

  -