GH repo: https://github.com/evgerritz/cpsc429_labs/tree/lab1
“make run”: Build, insert kernel module and run tests

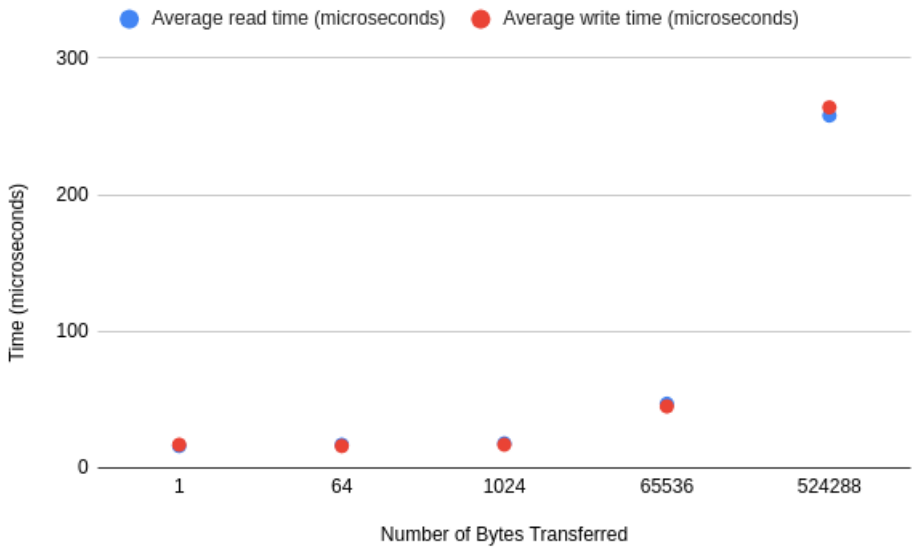
Part 1:

- Acknowledgments
    - Wrote module by referencing https://tldp.org/LDP/lkmpg/2.6/html/x121.html.
- Overview
    - This is a simple kernel module that prints to the kernel ring buffer upon being loaded and unloaded.
- The biggest difficulty here was simply setting up a virtual machine using qemu and cloning the github repository without a GUI.

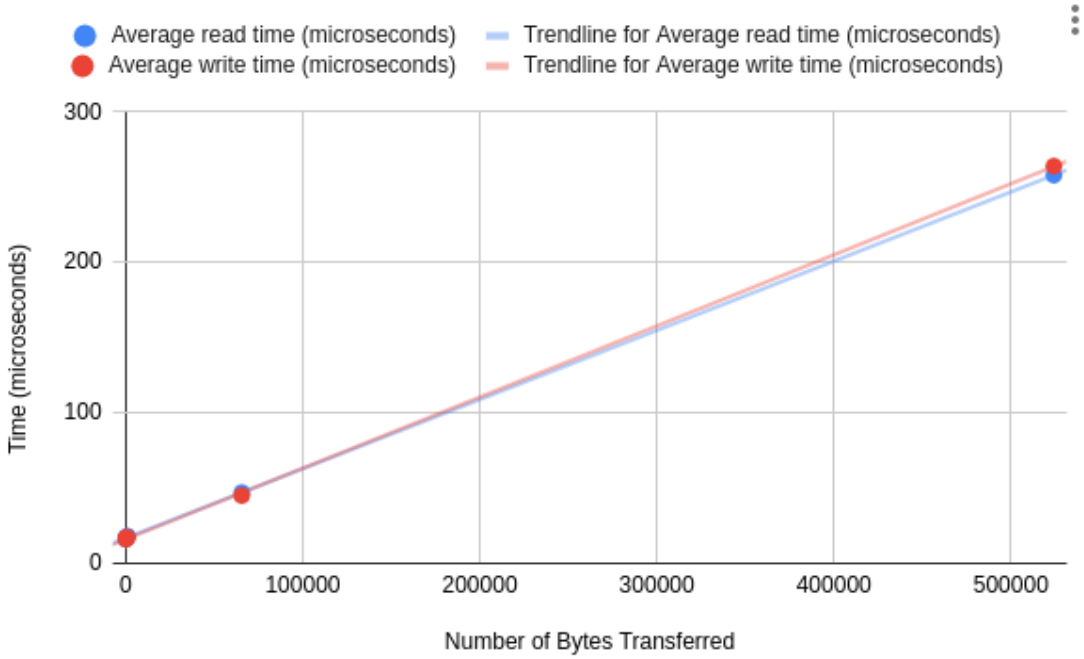
Part 2:

- Acknowledgments
    - Based module initialization/device file creation on https://sysplay.github.io/books/LinuxDrivers/book/Content/Part05.html.
    - Used this method to change the permissions of the created device file.
    - Used https://tldp.org/LDP/lkmpg/2.4/html/x579.html for information regarding struct file_operations.
    - Referenced http://www.makelinux.net/ldd3/chp-6-sect-5.shtml for implementing llseek().
- Overview
    - Mymem.c implements the mymem kernel module
        - We initialize the module by creating and allocating a device file along with a buffer that can be used by the kernel
            - The device file was created using the cdev.h, device.h headers
        - When the module exits, we need to destroy the device file and deallocate the buffer
        - Opening/closing does not require implementing any additional logic, as we never actually interact with the /dev/mymem file itself
        - Reading simply entails copying the requested number of bytes from the kernel heap allocation to the user’s buffer; writing is the same in reverse
        - Implementing the behavior of llseek requires handling each of the SEEK_* cases

- Challenges
    - One challenge was sorting through all the various steps, headers, and function calls associated with creating device files.
        - This was overcome through a great deal of googling, trying approaches out, and changing them if they didn't work. Once I got something working, I tried to make sure I knew what was happening and why it worked.
    - Another challenge was ensuring the accuracy of the testing information.
        - In particular, I wanted to ensure the full operations were carried out for each trial (such that later trials would not benefit from any caching of values).
        - To do this, I initialized the write buffer to random elements for each trial, ensuring that each buffer would be written/read from scratch on each trial.
        - The time required to initialize this buffer was not included in the final time.
- Questions: when plotting the throughput on a graph, do you observe any difference, i.e., more or less bytes read/written per second? If so, what do you think is responsible for the difference in throughput (what is the source of overhead)?
    - Interestingly, the read/write times were quite comparable, although the read time was slightly faster in the most extreme case.
    - If the memory read/write times themselves were the source of the overhead, we should expect this difference to be greater, as the time required to write the same amount of information physically usually takes more time than reading.
    - Since the times were, in fact, quite close, this implies that the main overhead was caused by multitasking done by the kernel.
    - The first few times are all roughly the same, as caching implies that whenever I/O operations are done, more data will be read than is actually requested.
        - Only once this minimum is passed do the times actually increase as a function of the number of bytes transferred.
- Data for read/write times (1000 trials):
    - Normalized bytes (exponential x-axis):



- Unnormalized number of bytes (linear x-axis)



- Data table:

Size	Average Read Time (n=1000) (microseconds)	Average Write Time (n=1000) (microseconds)
1B	16	17
64B	17	16
1KB	18	17

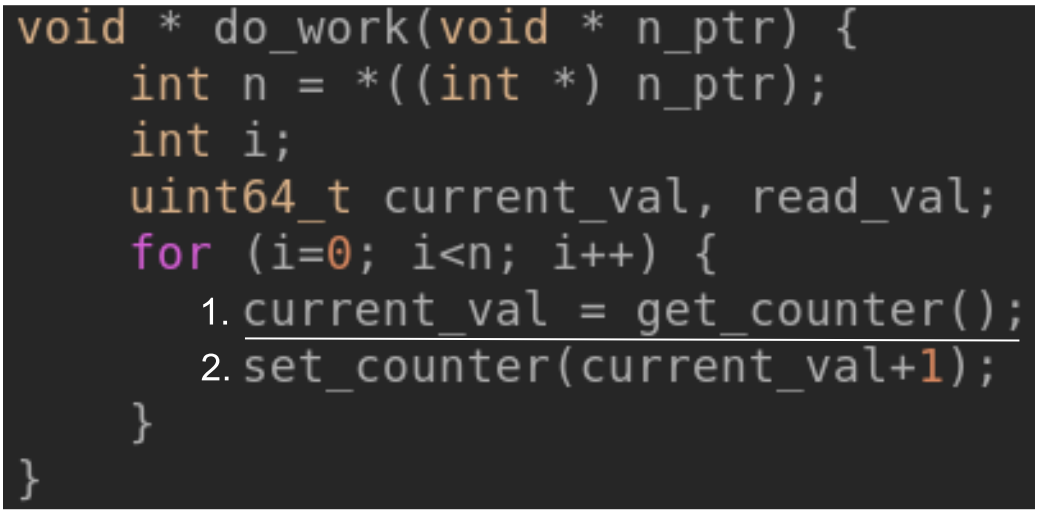
64KB	47	45
512KB	258	264

GH repo: https://github.com/evgerritz/cpsc429_labs/tree/1_p3
“make run”: Build, insert kernel module, run threads program with w=5, n=150000

Part 3:

- Acknowledgements
    - Used this for information on pthreads, mutexes: https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
- Overview
    - threads.c contains the code for running workers
        - Usage: ./threads *w n*
    - create_workers uses pthread_create to start *w*, each of which run do_work
    - do_work is the function each thread begins executing
        - It does the following *n* times:
            - Get the current value of the counter
            - Set the value to the original value plus one
        - Later, I added pthread mutexes to synchronize reads and writes.
    - At the end the program checks to see if the value of the counter is the value that would be obtained if all instructions were performed sequentially
        - If it isn’t, it outputs the percent error
- Challenges
    - Math in C is tricky. I tried to calculate percent error (as a double) from the actual and correct value for the counter (both uint64_t) using “double q = (actual - expected)/((double) expected);” and this always resulted in a huge, obviously incorrect value. The solution was to cast the values to doubles and then do the calculation, though I am still not entirely sure why.
    - Another small challenge was in implementing the lock. Initially I did so without fully understanding the race condition, so I make sure all reads and write were individually atomic (i.e., the mutex locking and unlocking was local to the *get_* and *set_counter* functions). As spelled out in the discussion below, the race condition actually arises from not protecting a larger critical section.
- Questions
    - Document at what values of W you receive incorrect results. What is the exact reason the incorrect results occur? Be specific.
        - Using n = 150,000, I received incorrect results on all values of $W > 1$. The incorrect results occur because the threads do not always write (line 2 in below image) the incremented value of the counter immediately after

reading its current value (line 1). Without, for example, mutexes, one thread may read the current value of *x*, after which the scheduler switches execution to another thread and possibly even more threads, ultimately changing the value to *x* + *a*. Once the scheduler returns to the thread with a variable *current_val* == *x*, it writes the value *x*+1 into memory, as it has no way of knowing that the actual value of the counter in memory (and not outdated the value of *current_val*) is *x* + *a*.



- Describe your solution in the lab report and explain any pros/cons it has.
  - I used the pthread implementation of mutexes to ensure that no context switching could occur within an iteration of the read-write loop. This ensures that each reading from memory is immediately followed by writing the incremented value into memory. The read-write section becomes atomic, so only one thread can execute those instructions at once. This is exactly the behavior we wanted to have initially. The pros of this method are that it's simple, intuitive, and it works well. One con is that this implementation results in a new global variable (the alternative to which would be adding parameters to functions and passing tramp data). A more serious con is that there is no guarantee that the scheduler won't preempt this thread from within the critical section. We definitely don't want this to happen, as it will waste cycles executing other threads which will get stuck waiting for the lock until execution returns to the thread with access.

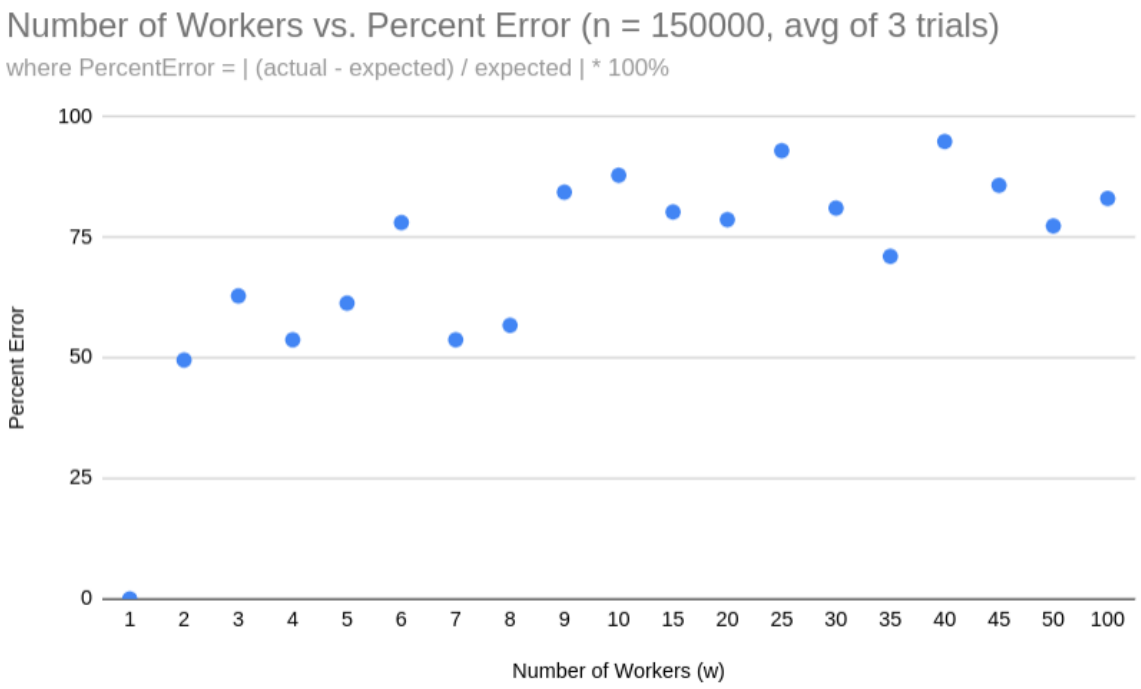
```
for (i=0; i<n; i++) {
    pthread_mutex_lock(&mymem_lock);
    current_val = get_counter();
    set_counter(current_val+1);
    pthread_mutex_unlock(&mymem_lock);
}
```

- How could you avoid this problem from within kernel space? In other words, how could your memory driver module ensure that user programs get the correct result no matter how many threads are simultaneously accessing it?
    - Here is the issue: in C, arbitrary user code may call the memory access functions in an arbitrary order, with differing intentions for which sections should be executed atomically. Since the driver doesn't know which request comes from which thread, it needs to know which sections of instructions are expected to run in parallel. Simply adding a mutex to each access function would not work, as it would not solve the issue of user code getting preempted within a critical section. We don't want a mutex to ensure only one function is executed at a time, but rather a mutex to ensure only one critical section is running at a time. Thus, one solution is to add an interface to the user for a locking mechanism in the kernel. Essentially, we are just moving the user-space mutex into kernel space and providing the user with some access. Another solution would be to write the module using Rust because then the compiler would guarante that the module is thread safe. The simplest would solution would probably be to add new access functions like, say, alter(fd, f, len) that could read len bytes and replace it with f performed on those bytes. One could then wrap this in a mutex to ensure no two kernel threads are executing this part of the driver at once. This method would allow users to perform (an admittedly specific set of) concurent operations without worrying about mutexes.
- What other kinds of unsafe behavior does your module allow to happen? For example, what happens if you close the file descriptor before all worker threads have completed? What happens if a worker thread tries to access a byte of memory outside of the region's bounds? How could you avoid these problems?
    - One issue is that right now, any number of user programs can access the memory in mymem. Thus, while we can avoid race conditions from different threads within a user space program by using a mutex in the userspace program, to avoid clobbering between programs we would need to ensure only one program has access to the mymem device at a time. To do this, we could add a mutex to the open and close operations in the

kernel module to ensure only one program can have an active file descriptor for the mymem device.

- Right now, if you close the file descriptor before all worker threads have completed, C will happily let you keep interacting with the file, even though all the reads and writes will result in garbage values. The calls to read/write/lseek will all result in -1 indicating failure, but it is up to the user code to handle these errors. An even better solution would be to refuse to compile code that allows a user to close a file descriptor and still try to use it.
- Well there are two ways of trying to access a byte of memory. One could try to lseek to an illegal value, or try to read/write past the end of the allocated block. Both of these are safe in that the driver code will refuse these requests by returning -EINVAL before any damage could be done. However, it is again up to the user to notice these failures and not use the garbage values they imply.

- Data/Plot



  -
  - Table
    -

Number of Workers (w):	Percent Error (n = 150000, avg of 3 trials)
1	0

2	49.5
3	62.8
4	53.7
5	61.3
6	78
7	53.7
8	56.7
9	84.3
10	87.8
15	80.2
20	78.6
25	92.9
30	81
35	71
40	94.8
45	85.72
50	77.3
100	83

GH repo: https://github.com/evgerritz/cpsc429_labs/tree/1_p4
Run the Rust mymem test using ‘cargo run’ inside mymem/test_mymem
Run the Rust multithreaded mymem test using ‘cargo run’ inside mymem/threads_rust
The rust_mymem module is in samples/rust_mymem.rs

Part 4.

- Acknowledgements
    - Referenced the provided resources, using the webinar and the rust-for-linux github documentation
    - Referenced the sample miscdev driver included with Rust-for-Linux
- Overview
    - mymem/test_mymem/src/main.rs implements the mymem timing test originally written in C in part 2.
    - mymem/threads_rust/src/main.rs implements the multithreaded test program originally written in C in part 3.
    - samples/rust_mymem.rs implements the rust_mymem Rust kernel module
- Challenges
    - Getting the kernel compilation to work was a pain. I had to mess around with the EXTRAVERSION setting in the kernel Makefile until the modules had the right versionmagic string. Otherwise, the target machine would not accept the compiled modules.
    - Debugging the rust kernel module proved to be quite difficult. I kept getting segmentation faults and was not sure why they were happening. It seems that the Rust compiler was laxer regarding the kernel module code.
- Questions
    - Is there any difference in performance compared to the program written in C? Why?
        - As can be seen in the data below, the performance was quite comparable for smaller sizes, with the C program only being significantly faster for the largest transfer size tested. This makes sense since Rust has a development goal to be close to C in performance.
    - What difference does your Rust-based implementation make regarding the data race problem you had to deal with in Part 3? Does Rust compiler help? If so, how?
        - Rust helps because Rust’s ownership system prevents multiple mutable references to a single object being shared across threads. Instead, Rust forces the programmer to use the proper tools for synchronizing shared state in the form of atomic reference counters and mutexes. This meant

that as soon as I had a Rust program without any compiler errors, the program worked exactly as intended!

- Now repeat the same experiment you did at Part 3 with a large value of N (>100000) with both the test program and Memory Driver in Rust. Is there any difference in performance compared to the program written in C? Why?
  - I measured the performance of both test programs using N=150,000 and W=50. The C test program with the Rust module took a total of 2 minutes, 27 seconds of CPU time. The Rust test program with the Rust module took a total of 2 minutes, 9 seconds. This is a little surprising, since I would have expected the C program to have slightly better performance. The difference in time is small and this method of measuring time is subject to a good deal of noise caused by background tasks, so I am not sure if this result is reliable. It is possible that the C bindings generated by bindgen, which are used by the C program to interface with the module are more inefficient or somehow introduce overhead that is not present when the Rust program is interfacing with the Rust driver.
- Now that both the test module and Memory driver are implemented in safe Rust, how does the Rust compiler help solve the data race issues from Part 3?
  - Theoretically, there could still be data race issues arising from within the execution of the kernel module. However, as before, Rust ensures that these race issues cannot occur by forcing the programmer to write code that guarantees only one function has access to the shared memory allocation at a time. In this case, I used a ref and a mutex to ensure the file operations were atomic.

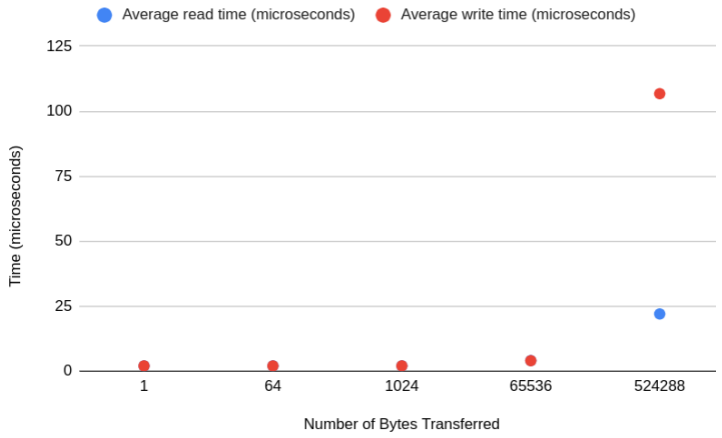
- Data/plot
  -

Size	Average Read Time (C) (n=1000) (microseconds)	Average Write Time (C) (n=1000) (microseconds)
1	2	2
64	2	2
1024	2	2
65536	3	4
524288	22	12

  -

Size	Average Read Time (Rust) (n=1000) (microseconds)	Average Write Time (Rust) (n=1000) (microseconds)
1	2.02	2.01
64	2.01	2.02

1024	2.03	2.03
65536	4.02	4.07
524288	106.72	22.01



-