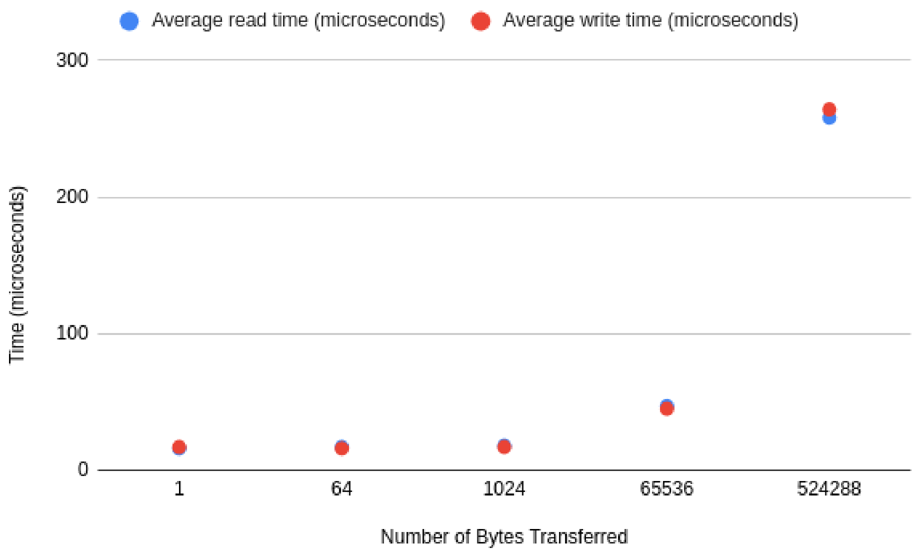
GH repo: https://github.com/evgerritz/cpsc429_labs/tree/lab1
“make run”: Build, insert kernel module and run tests

Part 1:

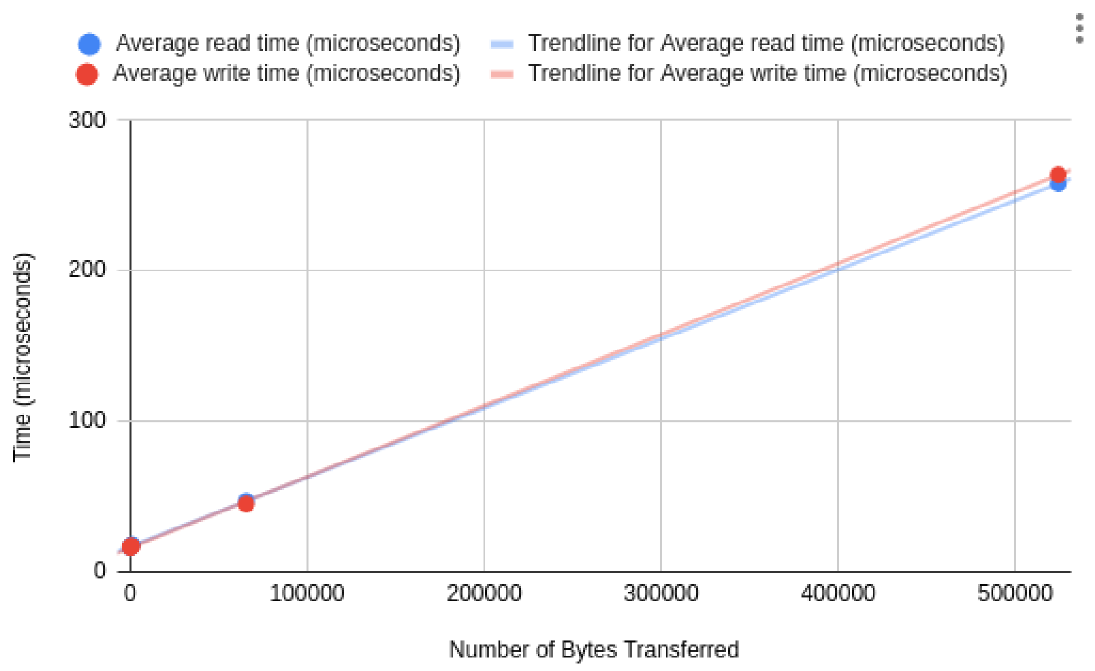
- Acknowledgments
  - Wrote module by referencing https://tldp.org/LDP/lkmpg/2.6/html/x121.html.
- Overview
  - This is a simple kernel module that prints to the kernel ring buffer upon being loaded and unloaded.
- The biggest difficulty here was simply setting up a virtual machine using qemu and cloning 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 create 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