**Assignment #1** – Linux Kernel Programming

**Part #1** – Hello World Kernel Module

In general, Part #1 was very entertaining to code and implement. I utilized several online resources to solve the coding assignment including YouTube and the following kernel module programming guide: https://tldp.org/LDP/lkmpg/2.4/html/book1.htm**.** My coding solution simply sets up the basic parameters of the module (license, author, etc.) and implements the corresponding functions module\_init() and module\_exit() that are to be used when creating my module. It was very interesting to learn about the kernel and its differences compared to userspace code. The use of printk() to print messages to the kernel log along with the unique KERN\_INFO macros were quite interesting to learn about and use since they differed so greatly from standard C programming. I had several challenges in completing this part of the assignment. Firstly, it was very difficult to get my virtual machine set up properly for kernel development. I tried several options including QEMU, Oracle’s VirtualBox, and VMware of which the last one ended up working for me properly. Additionally, setting up VSCode for kernel development was quite challenging since the path couldn’t find the proper header files to be able to properly lint my implementation. I simply removed the C/C++ extension and utilized Vim or plan VSCode for the majority of the assignment. Lastly, I was confused about finally removing the module since it would take a while for the “Goodbye, world!” message to appear in the kernel log after running sudo dmesg.

**Part #2** – Memory Driver Module

Part #2 was a very interesting portion of the assignment to implement. I mainly utilized this YouTube series that I found describing how to create device files and write/read to them along with a couple of StackOverFlow pages. My coding solution implements the standard code needed to get a module up and running along with opening up a device file with 512kB of buffer storage so that a user can write/read messages to it accordingly. I had several challenges with this part of the assignment. The first problem that I ran into was almost crashing my VMWare Ubuntu installation due to memory errors caused by my kernel log. I ended up solving this by clearing out these logs, but I haven’t managed to figure out the error that is causing it to fill it up so quickly. Additionally, I ran into some problems with my read function, where it would continue in a loop forever causing my code to crash. Similarly, when running my user space program, it would often result in a result that stated killed. I ended up being able to solve this by fixing what I was returning in the function as well as my use of pointers.

**Analysis of Data**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **1 Byte Test (Write)** | **1 Byte Test (Read)** | **64 Bytes Test** | **~1k Bytes Test** | **~65k Bytes Test** | **~524k Bytes Test** |
| 0.000004 | 0.000001 | 0.000003 | 0.000012 | 0.000711 | 0.004590 |
| 0.000002 | 0.000001 | 0.000002 | 0.000012 | 0.000103 | 0.003456 |
| 0.000001 | 0.000001 | 0.000002 | 0.000019 | 0.000766 | 0.005566 |
| 0.000001 | 0.000001 | 0.000002 | 0.000013 | 0.000744 | 0.004079 |
| 0.000001 | 0.000001 | 0.000002 | 0.000012 | 0.000753 | 0.002530 |
| 0.000001 | 0.000001 | 0.000002 | 0.000070 | 0.000767 | 0.004092 |
| 0.000001 | 0.000001 | 0.000002 | 0.000013 | 0.000820 | 0.005746 |
| 0.000001 | 0.000001 | 0.000002 | 0.000014 | 0.000903 | 0.003848 |
| 0.000001 | 0.000001 | 0.000002 | 0.000013 | 0.000774 | 0.001990 |
| 0.000001 | 0.000001 | 0.000002 | 0.000012 | 0.000722 | 0.005769 |
| **Average Value & Bytes/Secs** | | | | | |
| 0.0000014 | 0.000001 | 0.0000021 | 0.000019 | 0.0007063 | 0.0041666 |
| 714285.7143 | 1000000 | 30476190.48 | 53894736.84 | 92787767.24 | 125831133.3 |

This graph is very interesting as it shows just how much they different tests differ in terms of performance. As can be seen, as we increase the amount that must be read/written, the time required increases exponentially along with the file sizes. This, of course is to be expected, however what was interesting was the increasing bytes/sec as the amount to write/read increased. There is most certainly a lack of precision which can be artificially deflating the bytes/sec calculation for the smaller sizes. Additionally, some overhead can be created in getting the system ready for the smaller sizes since it doesn’t take much time to actually read/write for different sizes. So, the time it takes for the system to set everything up is definitely playing a role in making it so that in reading/writing larger files, the bytes/sec calculation was larger.

**Part #3** – Multithreaded Test Module using the Memory Driver

Part #3 was quite simple to implement overall. I didn’t really run into many issues while coding this as a majority of the information I was able to find on StackOverFlow or the Linux manual. One interesting this while implementing the solution was trying to somehow get my INIT\_VAL which was equal to DEADBEEF to convert to an integer which I managed to do by setting up my read and write functions correctly to read in eight bytes and to a point to a long long int type. I also observed race conditions quite quickly. My program started to return bogus results pretty much after increasing the workload of each worker and it settled at around one-hundred and eighty-eight. The reason that these incorrect results occur is because of the fact that multiple threads are trying to edit the information that is stored in the device file. As mentioned by the homework, this is known as race conditioning, where the two threads are racing to access and then change the data.

**#define INIT\_VAL “DEADBEEF”** translates into **long long int val = 5063529506963277124**

**Documentation of W Values**

|  |  |  |
| --- | --- | --- |
| (**W**)orkers | (**N**)umber | **Valid Results** |
| 1 | 25 | Yes (5063529506963277149)  difference: |
| 1 | 250 | No (5063529506963277312)  difference: |
| 1 | 2500 | No |
| 1 | >2500 | No |
| 2 | 25 | Yes (5063529506963277174)  difference: |
| 3 | 25 | Yes (5063529506963277199)  difference: |
| 4 | 25 | Yes (5063529506963277224)  difference: |
| 5 | 25 | Yes (5063529506963277249)  difference: |
| 6 | 25 | Yes (5063529506963277274)  difference: |
| 7 | 25 | Yes (5063529506963277299)  difference: |
| 8 | 25 | No (5063529506963277312)  difference: |

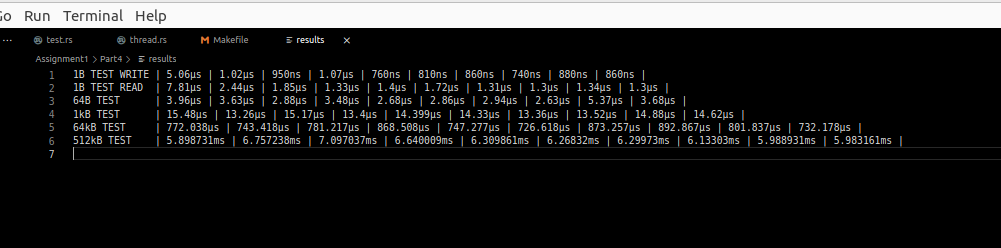
In order to resolve the multithreading issues that were occurring, I decided to use mutex (i.e. mutual exclusion locks) that lock down the information being accessed, perform the operations necessary, and unlock when it is ready to be accessed by another thread. After doing this, I was then able to run the following combination of workers and workload (W: 50 and N: 20000000) and doing anymore would result in my system crashing due to memory issues from within the log files. This solution is good for the reasons that it actually allows for the code to work at a large scale, however it hinders the programs ability to truly perform multiprocessing on whatever workload it is that the programmer wants to perform. It’s effectively locking everything done so that it’s no longer parallel but instead linear. Similarly, since the two commands needed to make this work are both system calls, it makes it a very slow approach when testing with very large numbers. For the above mentioned parameters, it took about five minutes for my computer to generate a result. You would be able to avoid this problem from within kernel space by detecting how many threads are running/trying to access the device file and enforce locks on how many are able to actually access the information. As a result, the program wouldn’t run into any issues and produce a correct output every time. There are also several risks that come to play when it comes to building Linux kernel modules. Firstly, the module has access to practically all parts of the kernel (by design of course to not have to restrict and cost efficiency and also for the purposes of it being open source), meaning that things could break very easily. A well-programmed kernel module would restrict the user from accessing this, but nothing prevents a module from calling an arbitrary function that it has the address of. Apart from that, I had attempted to close the file descriptor before all of the work threads were completed resulting in a bogus answer. Interestingly enough, trying to write/read past the allotted space by the device file was detected as a warning by my compiler. However, you could avoid both of these problems manually and within the custom module by having it detect if any of these test cases are occurring and program against them. For example, if you were to write past, the module would detect this and handle the case appropriately.

**Part #4 –** Everything in Rust

Overall, this part of the assignment was very challenging and relatively time consuming, but nevertheless it was very rewarding. I ran into several issues while coding up the different files that were required. Firstly, while implementing the single threaded code, I ran into a simple issue where I would be testing multiple times without clearing out the device file. Since I was using the read function and reading in the entire file, I ended up with a weird result where a lot of the times, it would end up taking in the entire string thus increasing the size of the string and resulting in the same times when running the test code. Additionally, actually compiling rust code was very tedious, especially when writing the threaded program since it required several wrappers to ensure that the mutexes were working properly which required me to move my variables around in scope and into the thread which took a bit of research and planning. Lastly, coding up the module was rather challenging since there is very minimal documentation. I ran into the trap (as mentioned in class) of coding up the seek command in Rust where I ended up simply keeping track of the offset in my storage structure.

**Is there any difference in performance compared to the program written in C? Why?**

There is very minimal difference between the two programs written in C and Rust. Although as can be seen by the table below, Rust generally performed a tiny bit slower than C. This, of course, is to be expected since Rust is a “safer” version of C, meaning that it will use/require more checks to ensure that there is nothing unsafe about the code. However, according to statements from the Rust developers, it is aimed at being as fast as C. This explains the comparable speed of both programs rather than what I had initially expected, that being Rust being slower than C.



**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?**

The difference between my Rust-based implementation in how it handles the race problem is quite interesting relative to how it’s done in C. In C, a lock is created that then is put onto some sort of memory to lock it down and have it not be used by another thread. However, in Rust it is implemented so that the Mutex actually owns the data that is inside, and so the programmer has to find a way to determine atomically how to ensure that it is thread safe. Therefore, we wrap it once again in an Arc type (Atomically Reference Counted) which ensures that the information is locked and unlocked accordingly to be used as needed by the threads. The compiler helps in ensuring that this becomes the case. For example, when testing out the code solely using the Mutex object, it would not allow since everything is moved into the closure of the thread by value. Therefore, it was necessary that the reference structure be created such that this wasn’t the case. After doing so, everything compiled correctly, and thread safety was properly ensured by Rust’s compiler.

**Questions #1 - #3 in the Specifications Document**

Performing the final tests on my compiled Rust module and test programs, I found that the performance was comparable to C, and that the program handled the race conditions quite well. With very large values of N, Rust did just fine in locating and updating the value without causing any issues. I did have some issues when it came to making sure that it was reading and writing everything correctly, however it was not possible to get Rust to compile without having it use a Mutex and Arc around the actual data so as to protect it from race conditions. All of this is of course because Rust is built so that it is comparable in speed to C and is very safe at the same time. Now that everything is implemented in Rust, the compiler helps significantly in helping to solve data race issues that we saw in Part 3. The way that Rust implements Mutexes is very interesting and different from C, and the compiler does a good job at catching that all movement through scope is handled accordingly with a copy or accounted for atomically.

**Additional Information**

Note: I utilized the following resource very heavily: https://tldp.org/LDP/lkmpg/2.4/html/book1.htm in the development of my kernel code. Significant parts for the implementation of the main module functions were utilized (namely, the general setup of the code).