

Lab 1 Report | Hao Li | 2-22-21

Part 1

This was the simplest part of all. For this part, it was mainly getting used to the environment.

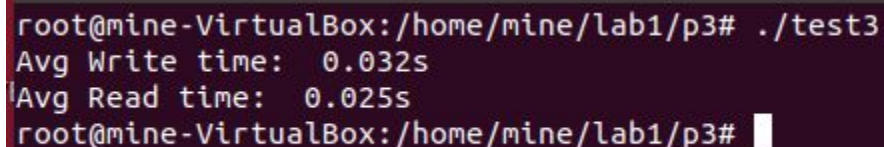
Part 2

This part took me a few hours. I did a lot of research on what a sysfs entry is and how that would interact with our kernel module. I was confused on the different ways to properly make a sysfs entry. Then to actually implement `double_me`, it was when I noticed that I needed another method which was then that I found the macro `module_param_cb`.

Part 3

This part took me the longest. It took me nearly 3 full days. Starting from absolutely nothing, I had to learn what a device is and all the details regarding how to implement it. Then, I was puzzled at first how to implement a character device given the spec since I thought my code had to do exactly what the spec says without knowing that some behavior was happening already. The part of the open function returning a file descriptor confused me for a while until I realized this was already a default behavior of open and we didn't need to implement it. Other details that took me a long time included understanding what "regions" meant, properly making the regions sysfs entry, and the copy to/from user functions.

Base Module Experiment

A terminal window with a dark background and light-colored text. The text shows a command being executed and its output.

```
root@mine-VirtualBox:/home/mine/lab1/p3# ./test3
Avg Write time:  0.032s
Avg Read time:  0.025s
root@mine-VirtualBox:/home/mine/lab1/p3#
```

Part 4

This was a relatively easier part given that part 3 was finally implemented. The time for this part was spent altering the right parts of code from part 3 and then running the experiment.

Smart Module Experiment

```
root@mine-VirtualBox:/home/mine/lab1/p3# ./test4
Chunk Size: 1
Avg Write Time: 0.04964270s
Avg Read Time: 0.04988805s
Chunk Size: 64
Avg Write Time: 0.00001395s
Avg Read Time: 0.00001180s
Chunk Size: 1024
Avg Write Time: 0.00000025s
Avg Read Time: 0.00000015s
Chunk Size: 65536
Avg Write Time: 0.00000020s
Avg Read Time: 0.00000025s
Chunk Size: 1048576
Avg Write Time: 0.00000020s
Avg Read Time: 0.00000025s
```

There is a difference in throughput. As chunk size increased to 1024, the throughput was larger while as the chunk size increased above 1024, the throughput stayed constant. The main difference could be due to the fact that the buffer used to do the read and writes is only 1024 bytes long. Therefore anything more is limited by the size of this buffer and the less sized chunks simply wasn't utilizing the buffer well enough.

Part 5

This part was relatively straightforward. There were not many problems implementing it.

Using $N = 100,000$, as soon as $W > 1$, then we see incorrect results. This is due to the fact that once there are more than one thread trying to read and write to the same part of memory, we get a race condition. This causes the values read to be inaccurate and therefore giving an incorrect result since some thread1 can read, then at the same time thread2 also reads, then thread1 and thread2 both write the same value back which gives an incorrect result. In short, this is a problem of concurrency.

My solution to this concurrency problem was relatively simple. Basically, I had a mutex in the form of an atomic counter where the one thread that read a zero only gets to read and consequently write to the memory, while all other threads do a busy wait. By doing so, we were able to easily guarantee that each read and write combination is atomic and so no overwriting occurs and values read is accurate. Obviously, a con to this approach is that it defeats the purpose of having a multithreaded program since having the mutex basically transforms the read/writes into a single thread.

Perhaps a kernel space solution is to keep a counter to whether there is already a thread trying to do a read/write to a part of memory and either rejecting or delaying the read/writes of other threads until the first thread has finished.

My program truncates the amount read/writes if it sees that the offset plus the length of the data will overflow the allocated size. My smart module does not check if the memory that it is trying to access is valid before reading or writing to memory. When I deallocated the region before the worker threads have completed, nothing abnormal happened other than the fact that obviously the result will be wrong. My module also does not check if the pointers are null and whether we are dereferencing null pointers. It takes for granted the pointers we get are valid.

Part 6

The implementation for this part was not hard, however it took a lot of time because there were a bunch of details/specifics that were not made clear in any tutorial and it all eventually came down to trial and error. A very good example is just to let us know ahead of time that we will need about 40GB of memory to do this part of the lab. I wasted several hours (~3-4 hours) first trying to resize my VM and then giving up on it and setting up an entirely new one before even proceeding on the assignment.

The Linux kernel probably took into account many security considerations for not allowing loadable kernel modules to modify/add syscalls. This is because if this was allowed, then the kernel can be compromised simply by changing a commonly used syscall to run some malicious code which can be a great vulnerability. Also, the purpose of syscalls is to provide access to the kernel. So this is a way to limit unwanted interactions with the kernel.

When you call `syscall()` with a number that does not exist, then nothing happens in userspace and you just get back a -1 as a return value from `syscall` since the syscall number doesn't even exist. To prevent such errors in userspace, the best approach is probably referring to the syscall table before actually making the syscall. Practically, it is impossible to prevent this in userspace due to malicious users.

The files I changed were as follows:

1. I created a directory called `capitalize_syscall`, implemented the syscall in `capitalize_syscall.c`, and a `Makefile`.
2. Added that new directory to the `core-y` line of the kernel's `Makefile`.
3. Added the new entry to the syscall table located at:
 - a. `arch/x86/entry/syscalls/syscall_64.tbl`
4. Added the syscall header to the syscall header file located in:
 - a. `include/linux/syscalls.h`
5. Then recompiled and rebooted.

As a safety condition, I check that the pointer passed into the syscall is not NULL to prevent dereferencing a NULL or freed pointer. An additional argument added was the length of the string so that in the case where a NULL terminator is not added to the end of the string, it will still stop at the right place. However, this can also be a flaw if malicious users target the memory region after the string by giving the syscall a very large length.

Part 7

This was also pretty straightforward given part 6 was implemented. It only took some searching before stumbling across the right lines of registers and inline assembly to modify and write. The max number of parameters to a syscall is 6 because the parameters of syscalls have special dedicated registers and there happens to be six of them. If we need to pass more arguments, we can pass a pointer to an array where we can have multiple arguments from there.