Arjun Bhagat ID: 917129686
Github: smeerj CSC415 Operating Systems

Assignment 6 – Device Driver

Description:

This assignment is designed to help us understand the fundamental concepts of a device driver and overall kernel development in Linux, and how to load and unload the device driver into the kernel.

My program implements a simple caesar cipher that accepts different shift values and shifts the message using the given value. In my test program, I hardcoded a message and three different shift values and it iterates through the different shifts and outputs the shift value, the encrypted, and decrypted message for all three values.

Approach:

I began by watching the recorded lectures and getting a skeleton together for my driver. I added in all the #include files I'd need, defined all the constants I'd need, and wrote out function prototypes for my device read(), write(), close(), open(), and ioctl() functions. I then created my crypt_key structure which will hold the private data when allocated in my open() function with just two fields: the message and the shift value.

I started implementing write() and read() using copy_from_user() and copy_to_user() as specified in the requirements of the assignment. I made a copy of the private data locally within the functions and made sure the data was copied to/from the user-space correctly, as well as to access the private data. I first check if the private data is NULL and exit out if it is in both functions. Lastly when writing write(), I also check that the message handed in isn't larger than the maximum message size I set of 256. I then call my encryption function to encrypt/decrypt the message accordingly. I then moved on to open() where I allocate the memory for a private crypt_key using vmalloc and initialize it using memset(). I check to make sure the data was allocated correctly, set a default shift value of 5, and assign the pointer to the file so the driver can store the data with each open file instance. close() came easily: I simply free'd the memory allocated for the crypt_key if it wasn't already NULL.

I then started on ioctl() in which I again retrieve the private data from the file structure, check if it's valid, copy the shift value from the user, and check that the copy was successful. I then needed to check if the given command was for encryption or decryption. I did this using a simple if-else statement and depending on whether the command was 3 or 4 indicating encryption and decryption respectively, I modulo the given shift value by 26 to set the range to 0-25 since a shift value of 26 wouldn't accomplish anything, and also set the value to negative if the decryption option was selected. I also added handling for invalid command options and print an error. I then defined my file operations struct and handed it the function pointers to use the implementation of my device.

For the initialization and clean-up of the driver, I did quite a bit of research on how I should be implementing this, as well as reached out to my classmates for advice. I learned that I could create a device class for my device and set the device permissions to allow read and write accessibility from any user. I began by calling alloc_chrdev_region() to allocate a major number, a device region, assign it a unique identifier, and checked for errors. I then called class_create() to create the device class. If this failed, the device would be unregistered and exit out. I set the dev_uevent of the class to my set_permissions function to set the permissions correctly. I then create the device using device_create on the device class to register the device file so it will be put in /dev. Lastly, I initialize a cdev structure with the file operations and add it to the kernel with cdev_add. If this fails, I destroy the device, the class, unregister, and exit. For cleanup_module, I just step by step undid everything I did in the initialization: I destroy the device, delete the cdev struct, destroy the device class, and the allocated major number and device region are unregistered with unregister_chrdev_region. I also logged successful initialization and removal at the end of both functions.

The testing program was quite simple. I first open the device file with read and write permissions which in turn tests the device's open function, and added error handling in case open doesn't work correctly. I set up a test message and three test shift values and in a for loop: I write the message to the device where the device encrypts/decrypts it, read the message back from the device, and then print the encrypted/decrypted message and the shift that was used. I manually set ioctl to use encryption and decryption for all three of the values to make sure everything was working correctly. Lastly I close the device file in turn testing the device close, and exit the program.

Instructions:

To run my device driver, you'll first need to cd to the Module directory within the folder of this assignment.

1. Run 'make' to create the kernel object.
2. Run 'sudo insmod cryptographer.ko'
3. You can run 'sudo lsmod | grep cryptographer" to ensure it's been loaded
4. cd into the Test directory
5. Run 'make run' and the test program will run

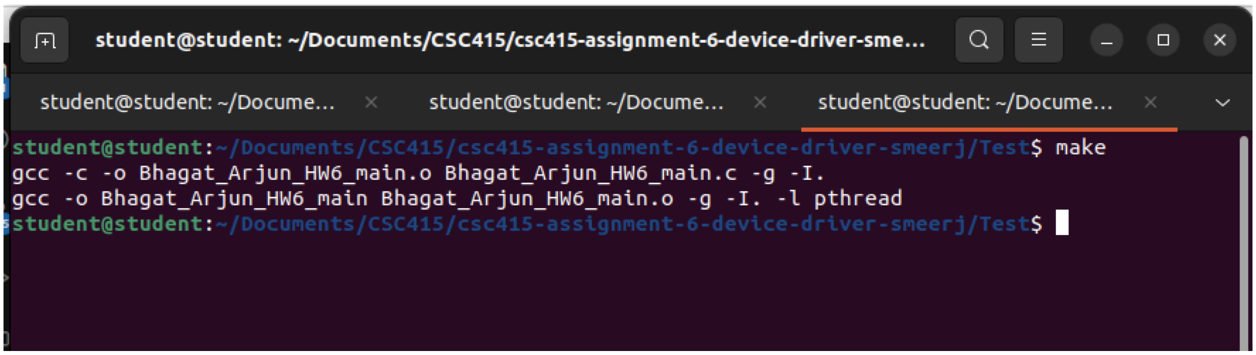
Issues and Resolutions:

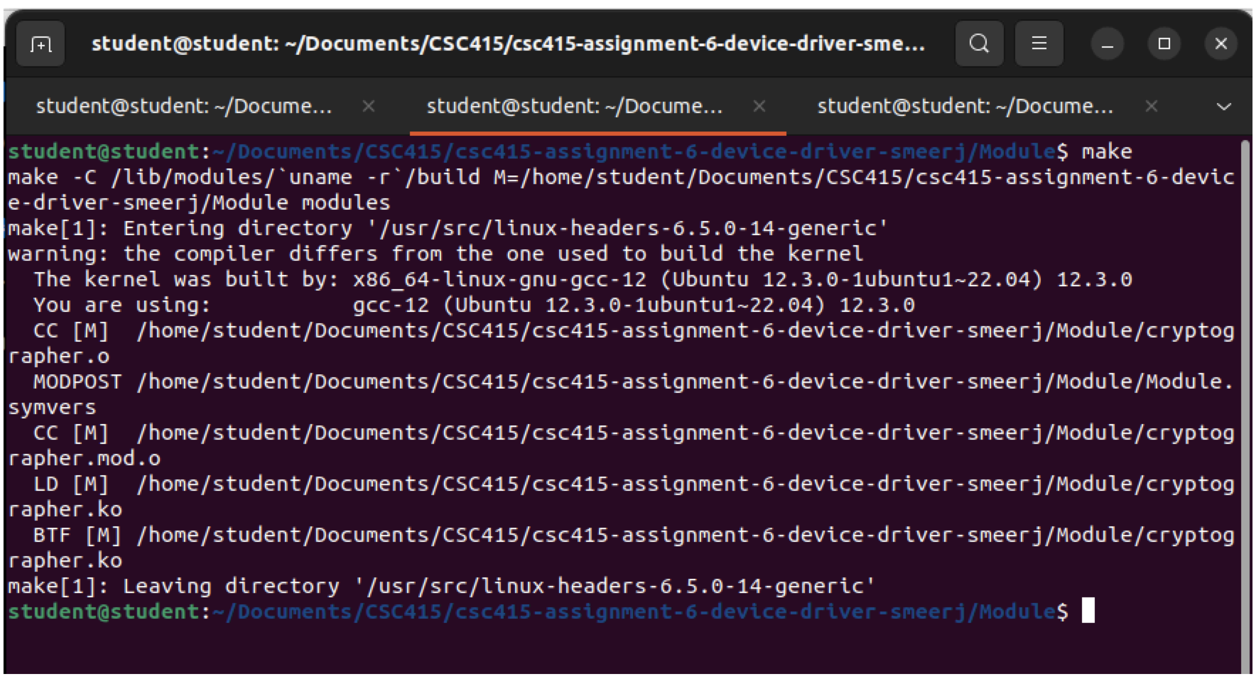
My first issue had to do with null termination of the string. I first found that in my write function, I had to manually add the null termination of the 'string' since I can't guarantee that the raw data in the buffer will contain one. I then also had to account for this in my read and write calls within my testing of the function. I call strlen(mssg) + 1 in write() and sizeof(mssg) - 1 in read() to account for the null terminator and ensure the buffer was a valid C string after reading/writing.

Arjun Bhagat
Github: smeerj
ID: 917129686
CSC415 Operating Systems

My second issue was when reworking my device driver, I was doing a make run everytime without realizing that I wasn't remounting the new kernel object file to the kernel. It took me a few more tries before I realized what a silly mistake I was making. After removing the kernel object with rmmod, making a new kernel object, and then adding it with insmod, I was then able to see the changes I was making.

Screenshot of compilation:





Arjun Bhagat
Github: smeerj

ID: 917129686
CSC415 Operating Systems

Screen shot(s) of the execution of the program: