Storage Device Driver

CFL: Cassidy Martin, Fola Alonge, Layne Struzinski

COSC 439 Group Project
Gitlab: https://gitlab.com/falong2/cfl.git

Table of Contents

Members/Roles

Because of the current circumstances of this being a project for an online class in a pandemic, all meetings for this project were held online at times that were most convenient for each member to attend. This required more organization to maintain a consistent project workflow, because most work will be happening asynchronously outside of the occasional group meetings. To combat this, specific tasks were assigned to each member in the group.

Group members and assigned tasks breakdown:

1. Cassidy Martin:
    - Meeting Organization
    - Coding
    - Additional Research
    - Documentation

2. Fola Alonge:
    - Main Research
    - Coding
    - Driver Testing

3. Layne Struzinski:
    - Coding
    - TroubleShooting Code
    - Documentation

Abstract

The intent of this Operating Systems project is to gain hands-on experience with lower level development on a virtual machine, by creation of a device driver from scratch that will allow our virtual machine to identify a usb storage device that is currently inserted into a usb port, and then interact with the storage device. Device drivers are crucial to allowing operating systems to interpret and communicate with the hardware physically connected to it.

To accomplish the creation of a device driver that will read and interpret a usb storage device, our group will do the following:

- Maintain organized documentation throughout the course of the entire project.
- Research device drivers and learn the basic background information required to develop one.
- Obtain a usb storage device that will be used consistently for all testing.
- Develop the code for the device driver
- Test the code for functionality with the usb storage device

Objectives

The main objective of this project is to develop a device driver on a virtual machine for a USB storage device. Upon successful completion of the device driver, the usb storage device will be able to be identified by the machine and interacted with. In addition, the developed device driver will allow data to be written to and read from the USB storage device.

Problem

Because the concept of drivers that allow communication between operating systems and other hardware is essentially already solved...the main problem our group hopes to solve is the limited understanding of operating systems and device drivers we all possess going into this project. A successful completion of this project will show that we have obtained and learned the information required to create a device driver and have a better understanding of the intricacies of how they allow operating systems to communicate with hardware.

Project Components

- Tessler_drive.c (The C file containing functions/structs used by the driver)
- Makefile (compiles the C file and creates all the module files and the driver)
- Tesser_drive.ko (the driver that is created by the Makefile and registered to the kernel)
- Other module files (also created by the Makefile)

Implementation

In order to meet our device driver functionality goals described within our project objective, the following macros, and functions were utilized in our tessler_drive.c file:

MODULE_LICENSE(GPL):

- The module license macro is responsible for the copyright license of the file and giving the module access to GPL-only commands within the kernel.

module_init(tessler_drive_init)

- Constructor of the module that will be called after compilation of .ko file and successful insertion of the module into the kernel via the usage of ‘insmod’ on the command line. Registers the module to the Kernel

MODULE_DEVICE_TABLE(usb, tess_table)

- At the time of compilation, this macro uses both parameters passed to it (usb, and tess_table) to create a device table, which is then referred to by the kernel when the usb device is connected and if a match is found then the module is then loaded.

module_exit(tessler_drive_exit)

- Deconstructor of the module that will only be called to remove the module from the kernel.

tessler_driver.c functions:

Static struct usb_device *device

- The Linux kernel’s means of representing the entire USB device.

Static struct usb_driver pen_driver

- Registers the driver to the Linux USB system. Contains key information about the driver such as the device it supports, and its functionality when a compatible device is connected to the machine. Included in our usb_driver struct is the name, probe, disconnect function, and id_table.

Static int __init tessler_driver_driver_init(void)

- Registers our new USB driver with the USB subsystem with a call to usb_register().

Static void __exit tessler_driver_exit(void)

- Deregisters our USB driver from the USB subsystem by calling the usb_deregister() function.

Static struct usb_device_id tess_table[]

- Used to determine which USB device will be able to communicate with the driver via vendor id, device id etc.

Static void tess_disconnect(struct usb_interface * interface)
(Prints when the device is disconnected from the kernel)

- This function is called when the storage device is removed from the USB bus. It prints out a disconnect message to the kernel and shuts down any pending data transfer.

Static int tess_probe (struct usb_interface *interface, const struct usb_device_id *id)

- The probe function starts the storage device initialization; initializing hardware, allocating resources, and registering the device with the kernel.

<u>Read and Write Functions:</u>
The read and write functions were taken from http://sysplay.github.io/books/LinuxDrivers/book/Content/Part13.html and used in our presentation as an example to show what the read and write function should look like. Hence why we added it into our tessler_drive.c function as a block comment.

Both functions are under the SCSI protocol. The SCSI protocol is used to connect the storage device to the computer to perform the transfer of data. During this transaction, the computer acts as the SCSI initiator, or the endpoint that sends a SCSI command. The storage device acts as the SCSI target that receives and processes the command to provide the requested data transfer. If completed, the read and write functions would be expected to perform the following tasks:

Static ssize_t tess_read(struct file *f, char __user *buf, size_t count, loff_t *off)

- This function transfers data from the SCSI targeted device.
- Uses the usb_bulk_msg() function to send data from the storage device. The usb_bulk_msg() function gets a timeout value and a buffer that places data that was received from the storage device.

Static ssize_t tess_write(struct file *f, char char __user *buf, size_t count, loff_t *off)

- This function transfers data to the SCSI targeted device.
- In the function, a pointer to data that the user wants to transfer to the storage device is received along with the size of the data. This function determines how much data can be sent to the device based on the size of the bulk endpoint that the device has. Then the data is copied to the kernel space, points the write urb to the data and submits the urb to the USB subsystem.

skel_write_bulk_callback()

- This function works alongside the write function. It is called by the usb_fill_bulk_urb() call in the write function and is called when the urb is finished by the USB subsystem. It returns if the write urb was successful.

Issues

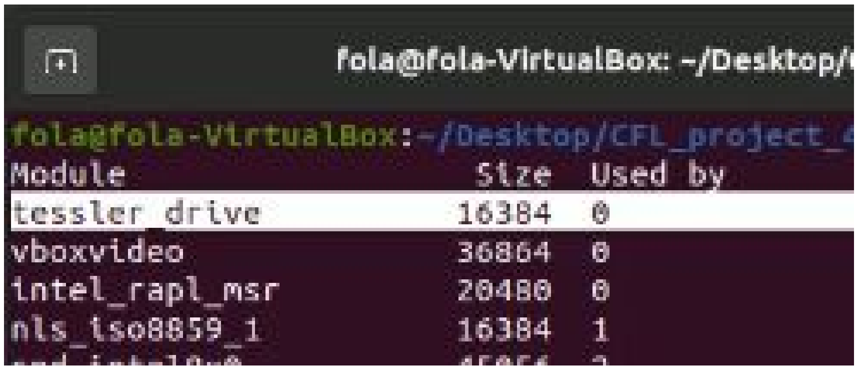
The first major issues experienced through the development of our project occurred during the creation of our makefile. Our previously created makefiles were purposed to only compile a single C file and create an executable, because we are dealing with a kernel module. Makefiles dealing with Kernel need to follow kbuild infrastructure, which requires a makefile to be named ‘Makefile’. This was later resolved allowing us to get to the point of compiling our tessler_drive.c file. In addition, our kernel headers had to be updated.

A later issue that was encountered after successful compilation and insertion of our tessler_drive.ko file was that the linux OS’ default usb storage drivers were interfering with our new driver’s ability to interact with the storage device plugged into the machine for testing. To overcome this, the default driver needed to be blacklisted. Using vim we could edit a read only file containing a list of blacklisted kernel modules. Then we added “blacklist usb_storage” to the list.

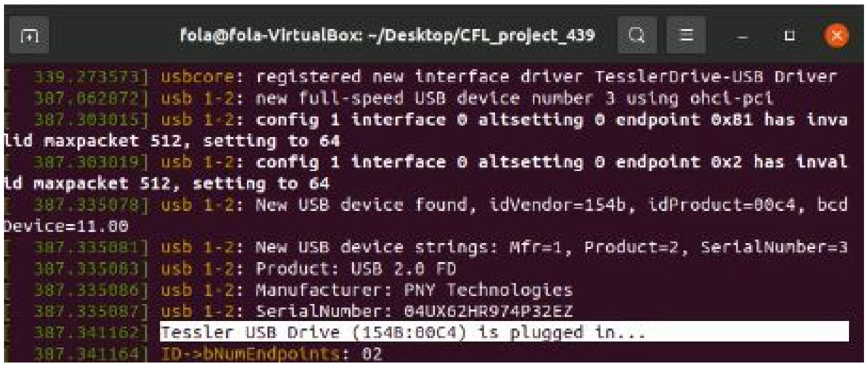
We also had an issue fully grasping the concept of using the SCSI protocol to complete the transfer of data.

Results

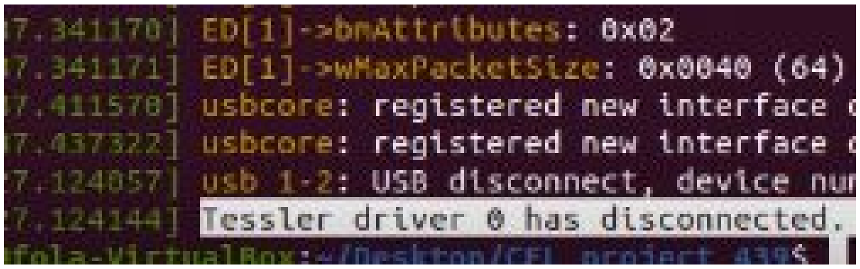
An aspect of our objective was to have a working device driver that enables our machine to interact with a USB storage device and this was accomplished and demonstrated by creating a driver that can register a module to the kernel, and handle when the usb connected/disconnected from the computer. We were able to accomplish this as shown below:



```
fola@fola-VirtualBox: ~/Desktop/
fola@fola-VirtualBox:~/Desktop/CFL_project_
Module                  Size  Used by
tessler_drive          16384  0
vboxvideo              36864  0
intel_rapl_msr         20480  0
nls_iso8859_1          16384  1
```

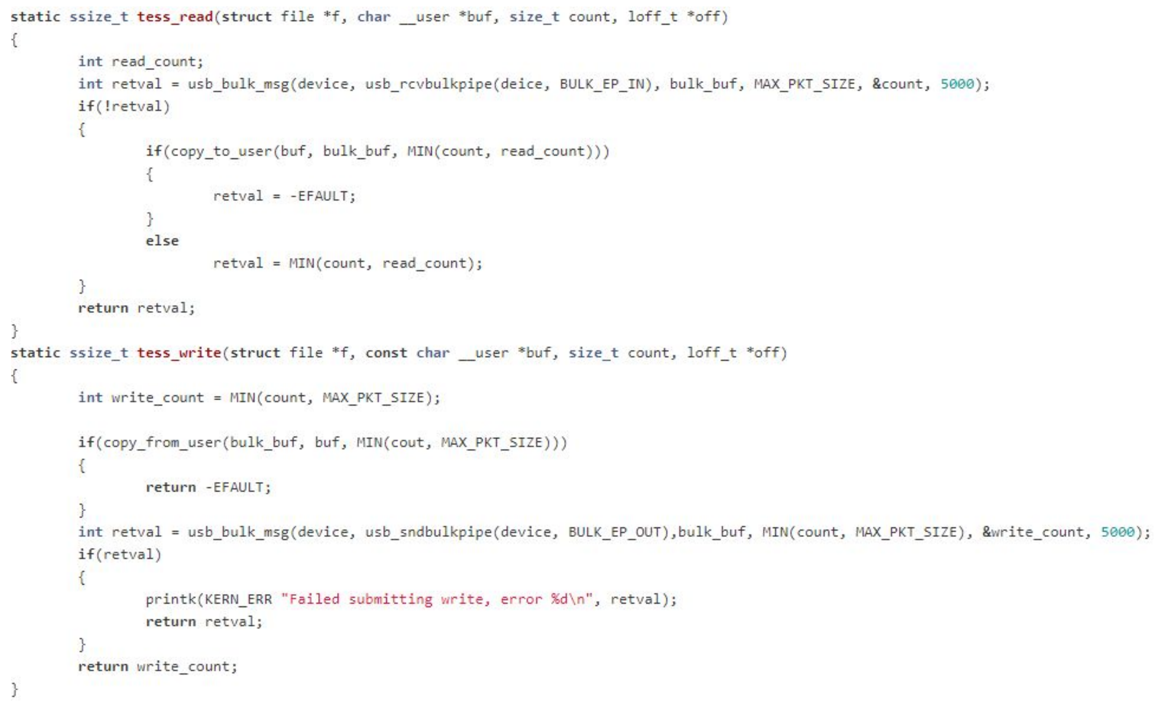


```
fola@fola-VirtualBox: ~/Desktop/CFL_project_439
[  339.273573] usbcore: registered new interface driver TesslerDrive-USB Driver
[  387.062872] usb 1-2: new full-speed USB device number 3 using ohci-pci
[  387.303015] usb 1-2: config 1 interface 0 altsetting 0 endpoint 0x81 has inva
lid maxpacket 512, setting to 64
[  387.303019] usb 1-2: config 1 interface 0 altsetting 0 endpoint 0x2 has inval
id maxpacket 512, setting to 64
[  387.335078] usb 1-2: New USB device found, idVendor=154b, idProduct=00c4, bcd
Device=11.00
[  387.335081] usb 1-2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[  387.335083] usb 1-2: Product: USB 2.0 FD
[  387.335086] usb 1-2: Manufacturer: PNY Technologies
[  387.335087] usb 1-2: SerialNumber: 04UX62HR974P32EZ
[  387.341162] Tessler USB Drive (154B:00C4) is plugged in...
[  387.341164] ID->bNumEndpoints: 02
```



```
7.341170] ED[1]->bmAttributes: 0x02
7.341171] ED[1]->wMaxPacketSize: 0x0040 (64)
7.411570] usbcore: registered new interface
7.437322] usbcore: registered new interface
7.124057] usb 1-2: USB disconnect, device nu
7.124144] Tessler driver 0 has disconnected.
fola-VirtualBox:~/Desktop/CFL_project_439$
```

Unfinished and Incomplete Aspects of Functionality



```
static ssize_t tess_read(struct file *f, char __user *buf, size_t count, loff_t *off)
{
        int read_count;
        int retval = usb_bulk_msg(device, usb_rcvbulkpipe(deice, BULK_EP_IN), bulk_buf, MAX_PKT_SIZE, &count, 5000);
        if(!retval)
        {
                if(copy_to_user(buf, bulk_buf, MIN(count, read_count)))
                {
                        retval = -EFAULT;
                }
                else
                        retval = MIN(count, read_count);
        }
        return retval;
}
static ssize_t tess_write(struct file *f, const char __user *buf, size_t count, loff_t *off)
{
        int write_count = MIN(count, MAX_PKT_SIZE);

        if(copy_from_user(bulk_buf, buf, MIN(cout, MAX_PKT_SIZE)))
        {
                return -EFAULT;
        }
        int retval = usb_bulk_msg(device, usb_sndbulkpipe(device, BULK_EP_OUT),bulk_buf, MIN(count, MAX_PKT_SIZE), &write_count, 5000);
        if(retval)
        {
                printk(KERN_ERR "Failed submitting write, error %d\n", retval);
                return retval;
        }
        return write_count;
}
```

Although we were not able to transfer data we wanted to incorporate this aspect in our presentation and final implementation to uphold our initial goal of being able to transfer data. As stated previously in our implementation portion of this paper, this portion of code was only used as a demo because we were unable to write our own functioning read and write functions. This was because we were not familiar with the SCSI protocol, how to implement it to build these functions, and were not able to find proper documentations specifying what SCSI commands we needed to incorporate in order to build these functions. With that said, the image above is in no way our own work and is only used for demonstration purposes.

References

https://www.kernel.org/doc/html/latest/driver-api/usb/writing_usb_driver.html

https://www.youtube.com/playlist?list=PL2GL6HVUQAuksbptmKC7X4zruZlIl59is

https://www.youtube.com/playlist?list=PLM8zRjaI08aQKKdUIqObqLTp4o5A67pOy

https://linuxtechlab.com/disable-usb-storage-linux/

https://developer.ibm.com/technologies/linux/tutorials/l-scsi-api/

http://sysplay.github.io/books/LinuxDrivers/book/Content/Part13.html