CSCI 3753: Operating Systems
Spring 2020

Programming Assignment Two

Due Date and Time: 11 PM, Monday, February 17, 2020
(Bonus for completed assignments turned in by 6 PM, Friday, February 14, 2020)

Goals

- Gain experience and familiarity with Linux kernel module (LKM) programming
- Gain experience and familiarity with Linux character device drivers

Introduction

This assignment consists of two major steps. First, you will write, install, and run a linux kernel module (LKM). Second, you will write a module to install a linux device driver. To help you complete these goals, this write-up contains the following sections:

Part I: Helpful Background Information

1. Environment Setup
2. Loadable Kernel Module (LKM) Overview
3. LKM Example
4. Device Driver Overview
5. Creating a Device Driver

Part II: Assignment Specifics

6. Assignment Overview
7. Device Driver Design Outline
8. Install the Module
9. Create the Device File
10. Write an Interactive Test Program
11. Submission
12. References
13. Extra Credit

Part I: Helpful Background Information

Environment Setup

You may use the VM from PA1, or any other Linux installation. It is highly recommended that you use a VM for this assignment and take snapshots as you go. Whatever VM you use, you need to install Linux source code. To do this, follow the instructions in PA1 step B.

It is helpful to know the version of the Linux kernel running on your VM. To get this information open a shell terminal and type `uname -r` The output will look like `x.y.z-ab-somethingelse`. `X` is the major number

and **Y** is the minor number. To check the version of the installed source code, you can go to the folder the source code is installed in. If you follow step B of PA1, the path will be `/home/kernel/src/$(uname -r)`

Loadable Kernel Module (LKM) Overview

LKMs are object files that are used to extend the running kernel's functionalities, and that can be inserted and installed on the fly. If you want to make a change in the running kernel itself, after you make changes, you have to reboot your computer (like what you did in the first assignment when adding a system call) before the changes you made are installed. This approach is time-consuming and can be painstaking.

In contrast, LKMs can add functionality to OS without the need to reboot. Additionally, using LKMs can store space because they can be easily uninstalled after use.

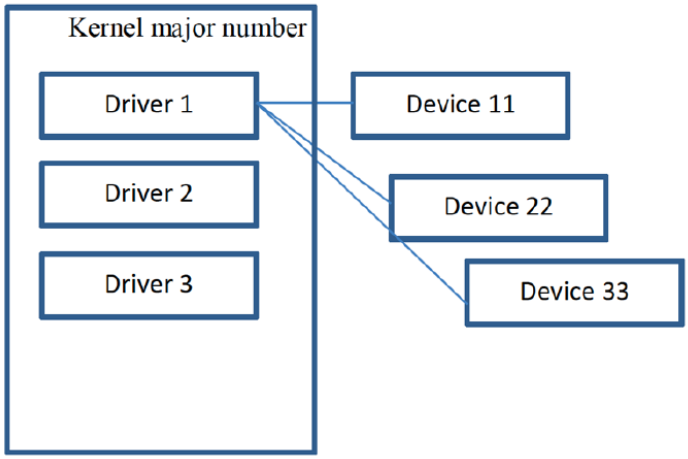
LKM Example

1. Get the file `helloModule.c` included with this assignment and store it in a folder named "modules." Open the file and look at the contents:
   a. Including `init.h` is required for module initialization
   b. Including `module.h` lets the kernel know that this is an LKM.
2. There are two functions defined: `hello_init()` and `hello_exit`. `hello_init()` will execute when the module is installed, and `hello_exit()` will execute when the module is uninstalled. To make sure this happens, at the end of the code I have added these two lines. The kernel will execute the function pointed to by either **module_init()** (at installation) or `module_exit()` (at uninstallation).
   a. `module_init(hello_init)`
   b. `module_exit(hello_exit)`
3. When writing kernel code (such as LKM code) you cannot use many user functions including `printf()`. Instead, use the function `printk()` function. For an example of the `printk()` function, see PA1. As a reminder, if you print with `KERN_ALERT` the message will be written in the log file in the location `/var/log/syslog` file and can be viewed by running the `dmesg` command or `sudo tail -f /var/log/syslog`
4. We'll use a makefile to build the module. Create a file named `Makefile` and type the following line in it: `obj-m:=helloModule.o` In this line, `m` means module, and the line as a whole tells the compiler to create a module object named `helloModule.o`
5. To compile the module, run: `sudo make -C /lib/modules/$(uname -r)/build M=$PWD modules` Now in the command prompt type the following: `ls` You will see there is a file named `helloModule.ko`. This is the kernel module (.ko) object you will be using to insert in the basic kernel image.
6. Install the module by entering `sudo insmod helloModule.ko` in the terminal. If you type `lsmod` you will see your module is now installed.
7. Run `dmesg` or `sudo tail /var/log/syslog` and you will see the expected output printed, as the `hello_init()` function was executed when the module was installed.

8. Remove the kernel by typing `sudo rmmod helloModule`. If you enter the `lsmod` command, you will see that your module is no longer listed. Type `dmesg` to see if the expected output is printed.

Device Driver Overview

Remember that in Linux, device I/O is modeled using files. Reading from and writing to a file will invoke the associated device driver to do the actual reading and writing. All device drivers have two associated numbers: major and minor numbers. The major number is unique to every device driver and the minor number differentiates all the devices belonging to that device driver. For example, in a hard disk there are many partitions. A major number is used to specify the hard disk device driver and each partition has a different minor number.



In the diagram above, there are three drivers (Driver 1, Driver 2 and Driver 3). 1, 2 and 3 are major numbers associated with device drivers in the kernel. Driver 1 works with three devices with minor numbers Devices 11, 22 and 33.

If you type `ls –l /dev | grep sda`, this will show all the device files (or current partitions) associated with the hard disk device drivers. The partitions are listed with their corresponding major and minor numbers.

There are two kinds of device drivers described below:

1. A **character device driver** reads from the device character by character, and writes to the device character by character. Character device drivers operates in blocking mode, which means that when a user writes data to the device, he/she must wait until the write operation completes. This is the most common of all device drivers.
2. A **block device driver r**eads large chunks (blocks) of information with a single read/write operation. Block device drivers are very CPU intensive (an operation takes some time to finish the execution) and are asynchronous (a user does not need to wait for the reading and writing to be completed).

Creating A Device Driver

To work with device drivers, you have to work with the corresponding device files. These files are stored in the

`/dev` folder. To see all device files on a machine, type `ls /dev`. You will have to create a file in this directory corresponding to your character device driver. The command to do that is `sudo mknod -m <permission> <device_file_location> <type of driver> <major number> <minor number>`. For

example, `sudo mknod -m 777 /dev/pa2_character_device c 240 0` where:

- `c` specifies the type of driver, in this case a character driver
- `777` sets the permission of the file so that the creator, the group the creator belongs to, and all the others can read, write and execute the file
- `240` is the major number of the driver that will be associated with this device file
- `0` is the minor number of the device
- `pa2_character_device` is the name of the device file.

The major number you choose must be unique. Inside your Linux source tree, in the file `<Linux_source_path>/Documentation/admin-guide/devices.txt` check for the current devices and their major numbers in your machine. (Note - if you followed the instructions in PA1, this will be `/home/kernel/linux-<your_versions>`) Use a major number that is not taken by any of the device drivers currently installed on your machine.

Part II: Assignment Specifics

Assignment Overview

In this assignment, you will need to write character device driver, install it and then create a device file in `/dev` folder associated with that device driver. Then read from and write to that file from a userspace test application. There are a lot of moving parts, so it's helpful to break this assignment into steps:

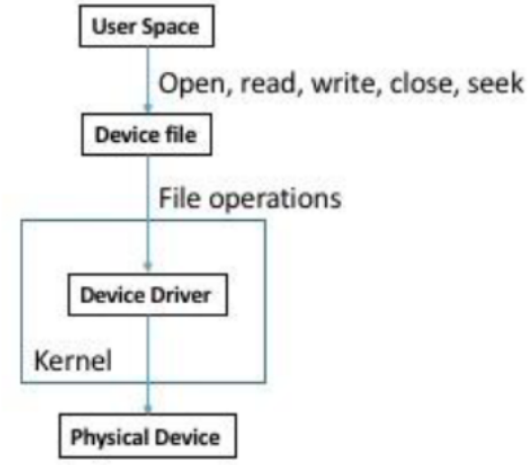
1. Create the skeleton of your device driver module
2. Code your file operations
3. Make and Install the module
4. Create a device file for this device
5. Create a test app that is an interactive program that will allow you to read from or write to that device file.

The below diagram is an overview of what is going on when you are working with a device driver. From the user-space program, you will issue the commands to **open, read, write, lseek or release(close)**. Those commands will be run through accessing your device file, which is associated with your device driver.

For example, when you run `echo hello >file.txt`, the operations performed are to open the file, write "hello" to the file, and then close the file. Similarly, when you run the command `cat file.txt`, the operations performed are to open the file, read the file content, and then close the file. The kernel sees that you are trying to perform file operations on a device file that is associated with a particular device driver. It then invokes the corresponding file operations for that device driver and then the device driver talks to the physical device to perform those file operations on the physical device.

For the purposes of this assignment, we will only write our data in the device file instead of the actual physical

device. For the extra credit section, you will be asked to create an emulated device.



Device Driver Design Outline

1. **Includes**: Along with the header files necessary for module programming, also include
   a. `linux/fs`: Contains the functions that are related to device driver coding
   b. `linux/uaccess.h`: Enables you to access data from user-space in the kernel and vice versa.
2. **Init & Exit:** Declare the init and exit functions and make `module_init()` and `module_exit()` to point to those functions.
   a. In the `init()` function, register the character driver using the `register_chrdev()` function. This function takes three parameters: the major number of the driver, the name of the driver, and a pointer to the file operations structure you want this driver to execute.
   b. In the `exit()` function, unregister the driver using the function `unregister_chrdev().` This function takes the major number and the name of the character driver.

   Check google for questions regarding `register_chrdev()` and `unregister_chrdev()`
3. **Internal Data:** Use a dynamically allocated kernel buffer (hereby referred to as a `device_buffer`) with a fixed (or constant) size to store the data written by the user. You should allocate memory for this buffer at initialization time and free this memory before exiting. There are two core functions to manage memory in the Linux kernel defined in `<linux/slab.h>`:

   a. `void* kmalloc(size_t size, gfp_t flags)` allocates memory for objects smaller than page size in the kernel. Use `GFP_KERNEL` for flags in this assignment.

   b. `void kfree(const void* kptr)` frees memory previously allocated using `kmalloc()`.
4. **File Operations:** To perform file operation in your device driver you need the `file_operations` structure found in `lib/modules/$(uname -r)/build/include/linux/fs.h` in the linux-hwe-4.15.0 kernel folder. Create a similar structure with the same `file_operations` type and with a different name with the open, close, seek, read and write operations only. You will have to implement these five functions, and set the function pointers in your `file_operations` struct to point to your

implementations. **Note:** there is no 'close' function. Instead, use the `release()` function signature instead.

a. **Open & Release(Close):** The **open** function takes two parameters: the inode (which represents the actual physical file on the hard disk), and the abstract file representation that contains all the necessary file operations in the `file_operations` structure. You don't have to do anything extra in this function - just print the number of times the device has been opened. Do the same thing for the **release** function.

b. **Seek:** The seek function takes three parameters: the file pointer, the offset, and a value (called whence) which describes how to interpret the offset. If the value of whence is **0** (`SEEK_SET`), the current position is set to the value of the offset. If the value of whence is **1** (`SEEK_CUR`), the current position is incremented by offset bytes (note that offset can be negative). Finally, if the value of whence is **2** (`SEEK_END`), the current position is set to offset bytes before the end of the file. Use `printk()` to record the offset after seeking. If a user attempts to seek before the beginning of the file or beyond the end of the file, an error should be returned with the current position being unchanged.
Note: For the seek function, change the prototype to: `loff_t llseek(struct file *, loff_t, int)`.

c. **Read:** This function takes four parameters: the file pointer, the userspace buffer to store the data read, the space available in the userspace buffer, and the current position of the opened file. The data is read from the current position of the file and is stored in the `device_buffer` array. Use the function `copy_to_user()` to copy data from the `device_buffer` to the userspace buffer. Use `printk()` to record the number of bytes read, and the offset at the end of read.

d. **Write:** The write function is similar. Copy data stored in the userspace buffer to the `device_buffer` using `copy_from_user()`. The write starts from the current position of the opened file, and the new offset should be recorded. If the current position is before the end of the file, this operation will overwrite some of the bytes that were written earlier. Use `printk()` to record the number of bytes written, and the offset at the end of write.

Install the Module

Follow the instructions in the previous sections to create and edit your makefile, compile your module, and install your module. Check that your module was installed by checking the kernel log and by running `cat /proc/devices`

Create the Device File

1. Create a device file for this device by the command as described in the previous section.
2. Try to echo and cat that particular file. Check if your device is working by examining the log file.

CSCI 3753: Operating Systems
Spring 2020

Write an Interactive Test Program

Write an interactive test program that will allow you to read from, write to and seek in the device file. Use the location of the device file you created in the `/dev` folder while writing the test code. Your interactive program should give the user the following options:

1. Press r to read from device
2. Press w to write to the device
3. Press s to seek into the device
4. Press e to exit from the device
5. Press anything else to keep reading or writing from the device
6. Enter command:

Option 'r': If the user presses 'r' then you should ask the number of bytes the user wants to read and create a buffer of that size (use `malloc()`) in which the data will read. Use the prompt below:

Enter the number of bytes you want to read:

Print the data read from the device file starting from the current position. The format of the output should be like this:

Data read from the device:

Option 'w': If the user presses 'w' then you should ask for the data to be written from the user. The format should be like this:

Enter data you want to write to the device:

Option 's': If the user presses 's' then you should ask for the values of the second and third parameters:

a. *Enter an offset value:*
b. *Enter a value for whence (third parameter):*

Option 'e': If the user presses 'e' then you should quit the testapp.

Other: If the user presses something else, you should continue giving the user the options like you did in step 3 in this section.

Submission

You are required to submit the following files in a zipped folder:

1. README. This file should include your contact information, descriptions of the files created as part of the assignment, instructions on how to install/user/configure your module and device file, and instructions on how to build and run your test program.
2. Device Driver Implementation
3. Makefile
4. Test program

CSCI 3753: Operating Systems
Spring 2020

Grading

Implementation: **40%** of your grade will be based on the performance of the best implementation that you provide. We will test your implementation on a number of test cases.

Interview: **60%** of your grade will be based on your interview. You will be expected to explain your work and answer questions regarding. You are also expected to know all concepts pertaining to the assignment. Note that we will use the test program you write to test your program, so it's very important for you to complete both parts of the assignment.

References

1. You can use the Linux manual pages to check the functions and their functionalities.
2. http://www.fsl.cs.sunysb.edu/kernel-api/re941.html
3. http://lxr.free-electrons.com/ident?i=unregister_chrdev
4. http://www.fsl.cs.sunysb.edu/kernel-api/re256.html
5. http://www.fsl.cs.sunysb.edu/kernel-api/re257.html

Extra credit

1. As we have discussed in the earlier sections, we are actually writing to and reading from the device file we created in the `/dev` folder. For extra credit, you are required to create an emulated physical device and read and write from and to that emulated device using the character device driver module you coded.

OR

2. Create a block device driver and do the same file operations like you did for character device driver.