PA2

Due Sunday by 11:59pm **Points** 100 **Submitting** a file upload
File Types zip

Programming Assignment Two

Due Date and Time: Sunday, October 4th, 2020 @11:59PM

Introduction

This assignment consists of three parts. First, you will compile, install, and run a simple Loadable Kernel Module (LKM). Second, you will write your own LKM that implements a basic device driver. Finally, you will write a user-space test program to test the functionality of your device driver.

Part 1 - Simple Loadable Kernel Module

Usually, as we saw in PA1, if you make a change to the kernel, you must recompile and then reboot before these changes become active. This approach is time-consuming and can be painstaking. In contrast, LKMs add functionality to the OS without the need to reboot or recompile the entire kernel. In short, LKMs are object files that can be used to extend a running kernel's functionality on the fly.

LKM Example

Create a directory, **/home/kernel/modules**, and edit a new file named **helloModule.c**. Populate this file with the following code:

```
#include<linux/init.h>
#include<linux/module.h>

MODULE_AUTHOR("Your Name");
```

```
MODULE_LICENSE("GPL");

int hello_init(void) {
    printk(KERN_ALERT "inside %s function\n",__FUNCTION__);
    return 0;
}

void hello_exit(void) {
    printk(KERN_ALERT "inside %s function\n",__FUNCTION__);
}

module_init(hello_init);
module_exit(hello_exit);
```

Notice a few things about the above:

- **<linux/init.h>** & **<linux/module.h>** contain the library headers for module initialization and other functions that support LKMs
- **module_init()** and **module_exit()** bind two functions, **hello_init()** and **hello_exit()**, to be executed when the module is installed and uninstalled
- As with PA1, you cannot use user space functions such as **printf()**, use **printk()** with **KERN_ALERT** to write messages to **/var/log/syslog**

We'll now use a makefile to build the module. Create a file named **Makefile** in **/home/kernel/modules** and add the following line to it:

```
obj-m:=helloModule.o
```

In this line, **obj-m** means module, and the line as a whole tells the compiler to create a module object named **helloModule.o**

To compile the module enter the following command:

```
make -C /lib/modules/$(uname -r)/build M=$PWD
```

and if the compilation is successful, your output should be something similar to:

```
make: Entering directory '/home/kernel/linux-hwe-4.15.0'
CC [M] /home/kernel/modules/helloModule.o
Building modules, stage 2.
MODPOST 1 modules
```

```
CC /home/kernel/modules/helloModule.mod.o
LD [M] /home/kernel/modules/helloModule.ko
make: Leaving directory '/home/kernel/linux-hwe-4.15.0'
```

You can now list out the contents of the **/home/kernel/modules** directory with the **ls -l** command:

```
user@csci3753-vm:/home/kernel/modules$ ls -l
total 276
-rw-rw-r-- 1 user user    331 Sep 12 13:04 helloModule.c
-rw-rw-r-- 1 user user 127232 Sep 12 13:51 helloModule.ko
-rw-rw-r-- 1 user user    603 Sep 12 13:51 helloModule.mod.c
-rw-rw-r-- 1 user user  72584 Sep 12 13:51 helloModule.mod.o
-rw-rw-r-- 1 user user  57864 Sep 12 13:51 helloModule.o
-rw-rw-r-- 1 user user     21 Sep 12 12:56 Makefile
-rw-rw-r-- 1 user user     43 Sep 12 13:51 modules.order
-rw-rw-r-- 1 user user      0 Sep 12 13:51 Module.symvers
```

Notice the file named **helloModule.ko**. This is the kernel module (.ko) object you will be inserting into the running kernel.

Install the module by executing **sudo insmod helloModule.ko**. Now when you use **lsmod**, you should see that your module is installed:

```
user@csci3753-vm:/home/kernel/modules$ sudo insmod helloModule.ko
user@csci3753-vm:/home/kernel/modules$ lsmod | grep hello
helloModule 16384 0
```

Run **dmesg** or **sudo tail /var/log/syslog** to see the message emitted by your kernel module when it initialized:

```
[ 2643.230212] inside hello_init function
```

Now remove the module by typing **sudo rmmod helloModule**. If you again enter the **lsmod** command, you will see that your module is no longer listed. Type **dmesg** to see if the expected output from unloading the module is printed:

```
[ 2657.344579] inside hello_exit function
```

If the all the above checks out, you've successfully completed the first and most basic part of the assignment. Now we can move onto the next part, where you will write your own kernel module to implement a device driver.

Part 2 - Write a Device Driver LKM

Linux Devices Overview

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 a major and a minor number, where the major number is unique to every device driver. The minor number differentiates all the devices belonging to that device driver.

For example, a typical system will have multiple hard disks, or at least multiple partitions on a single disk. A single major number is used to specify the hard disk device driver, but each partition has a different minor number. If you type **ls –l /dev/sda*** on your VM, this will show all the device files associated with all the hard disk partitions. You should see the partitions listed with their corresponding major and minor numbers:

```
user@csci3753-vm:/home/kernel/modules$ ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 Sep 12 12:13 /dev/sda
brw-rw---- 1 root disk 8, 1 Sep 12 12:13 /dev/sda1
brw-rw---- 1 root disk 8, 2 Sep 12 12:13 /dev/sda2
brw-rw---- 1 root disk 8, 3 Sep 12 12:13 /dev/sda3
```

It's worth noting that there are two kinds of device drivers:

- A **character device driver** reads and writes from/to the device character by character. This is the most basic type of device driver and usually the simplest to implement.
- A **block device driver** reads or writes large chunks (blocks) of data with a single read/write operation. These types of drivers are more complex, but usually more efficient in their use of system resources. Network interfaces and disk controllers generally prefer to use a block driver.

Many devices will have both a character and a block driver available. It's then up to the application programmer to decide which is the most appropriate or convenient to use.

Creating A Device File

To access your device driver, you will need to create a corresponding device file in the **/dev** directory, using the command **mknod**:

```
sudo mknod -m <permission> <device_file_location> <type of driver> <major number> <minor number>
```

For example, **sudo mknod –m 777 /dev/simple_character_device c 240 0** creates a device file where:

- **-m 777** sets the permission so that all users can read, write and execute the file
- **simple_character_device** is the name of the device file
- **c** specifies the type of driver, in this case a character driver
- **240** is the major number of the driver that will be associated with this device file
- **0** is the minor number of the device

The major number you choose for your driver must be unique. Inside your Linux source tree, in the file **/home/kernel/linux-hwe-4.15.0/Documentation/admin-guide/devices.txt**, check for the current device drivers and their associated major/minor numbers.

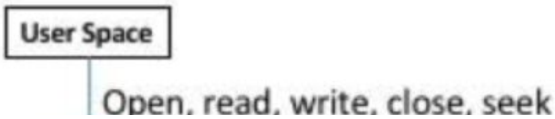
Device Driver Overview

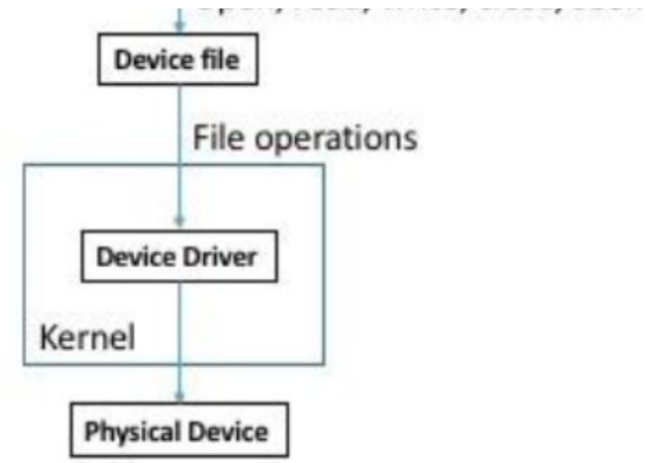
The diagram below summarizes what is going on when you are working with a device driver. From a user-space program, you will issue calls to **open(), read(), write(), seek() or close()**. These calls will access the device file in /dev which is associated with your device driver.

For example, when you run **echo hello >file.txt,** the operations performed are: 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: open the file, read the file content, and then close the file.

The device file, by way of its major and minor numbers, indicates to the kernel that you are trying to perform file operations on a device. The kernel will invoke the corresponding file operations in the device driver. The device driver then executes it's implementation of these file operations against the physical device.





For the purposes of this assignment, our device driver will only read and write data to a region of memory instead of an actual physical device.

Device Driver Implementation

Along with the header files necessary for module programming, you'll also need to include:

- **linux/fs.h** - contains the functions that used to manipulate files
- **linux/uaccess.h** - enables you to access data from user-space in the kernel and vice versa

Declare your **init()** and **exit()** functions and use **module_init()** and **module_exit()** to bind these functions

- In your **init()** function, register your character driver using **register_chrdev()**
- In your **exit()** function, unregister the driver using **unregister_chrdev()**

register_chrdev() takes three parameters: major number, a unique name, and a pointer to a file operations struct (see below). Check google or the references section of this writeup for questions regarding **register_chrdev()** and **unregister_chrdev()**.

We will use a dynamically allocated kernel buffer (hereby referred to as **device_buffer**) with a fixed size to store the data written to our device. 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>**:

- **void* kmalloc(size_t size, gfp_t flags)** allocates memory for use in the kernel, use the macro **GFP_KERNEL** as the flags argument in this case
- **void kfree(const void* kptr)** frees memory previously allocated using **kmalloc()**

Make sure you use a constant or macro to set the size of this buffer to 1**KB**.

Device File Operations

To perform file operations in your device driver you need to populate a **file_operations** structure. The system defined struct is found in **/lib/modules/$(uname –r)/build/include /linux/fs.h**. Create a similar structure with the same **struct file_operations** type but with a different name.   Define 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 that there is no 'close' function in the file_operations struct, use **release()** instead.

You are free to use the example below, the comments describe the interface your functions must implement:

```
struct file_operations my_file_operations = {
    .owner   = THIS_MODULE,
    .open.   = my_open,       // int my_open  (struct inode *, struct file *);
    .release = my_close,      // int my_close (struct inode *, struct file *);
    .read    = my_read,       // ssize_t my_read  (struct file *, char __user *, size_t, l
off_t *);
    .write   = my_write,      // ssize_t my_write (struct file *, const char __user *, siz
e_t, loff_t *);
    .llseek  = my_seek        // loff_t  my_seek  (struct file *, loff_t, int);
};
```

Open - The open function takes two parameters, a pointer to an inode struct (which represents the physical file on the hard disk), and a pointer to a file struct (represents the state of a file), and returns an integer indicating success or failure. In this function, you don't need to do anything other than log the number of times the device has been opened.

Release - The release function takes the same two parameters as open() and again returns an integer indicating success or failure. Use printk() to output the number of times the device has been closed.

Read - The read function expects four parameters: a file pointer, a pointer to a user-space buffer, the size of that buffer, and a pointer to the current position. Use the function **copy_to_user()** to copy data from the device_buffer, starting at the current position, to the user-space buffer. If successful, make sure to update the current position, and then return the number of bytes read. Use printk() to log the number of bytes read.

Write - The write function is similar to read. Copy data stored in the user-space buffer into the device_buffer using **copy_from_user()**. The write starts from the current position, and if successful, the position should be updated. Use printk() to log the number of bytes written. Finally, return the number of bytes written to the caller. If the user sets the current position back to the beginning of the file (by using seek), this operation may overwrite previously written data.

Seek - The seek function takes three parameters, a file pointer, an offset, and the value **whence**. Whence describes how to interpret the offset (note that offset can be negative). If the value of whence is **0** (**SEEK_SET**), the position is set to the value of the offset. If the value of whence is **1** (**SEEK_CUR**), the current position is incremented (or decremented, if negative) by offset bytes. Finally, if the value of whence is **2** (**SEEK_END**), the position is set to offset bytes from the end of the file.

If a user attempts to read, write or seek before the beginning or beyond the end of the device_buffer, an error should be indicated by returning a -1, and the current position should be left unchanged. You will need to implement some sort of bounds checking to ensure this behavior.

To get you started, we've provided a skeleton of these functions **here**.

Install and test the Module

Follow the instructions in Part 1 to compile and install your module. Verify that it's installed by checking the kernel log for printk() output or by looking in **/proc/devices**. Try to echo into (write) and cat from (read) your device file. Verify that your device is working by examining the output and checking the log file.

Part 3 - Write a small User-Space Test Program

Write an interactive test program that will allow you to read from, write to and seek in the device file. Your interactive program should give the user the following options by pressing the following keys:

Option ‘r’ - Your test program should ask for the number of bytes to read using the prompt:

Enter the number of bytes you want to read:

Making sure you create a large enough buffer using **malloc()**, read from the device file starting from the current position. Finally, print this data out to the console.

Option ‘w’ - Your program should ask for the data to be written from the user, using the prompt:

Enter data you want to write:

The user then enters the desired data terminated by a carriage return.

Option ‘s’ - Your program should ask for the values of the second and third parameters:

Enter an offset value:

Enter a value for whence:

Option ‘e’ - If the user presses ‘e’ then you should quit the program.

Other - If the user presses something other than the keys listed above, ignore it and wait for the user to enter a valid option.

Submission

You are required to submit the following as a single .zip file:

- 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.

- Device Driver Implementation
- Makefile
- Test program

References

http://www.fsl.cs.sunysb.edu/kernel-api/re941.html (http://www.fsl.cs.sunysb.edu/kernel-api/re941.html)

http://lxr.free-electrons.com/ident?i=unregister_chrdev (http://lxr.free-electrons.com/ident?i=unregister_chrdev)

http://www.fsl.cs.sunysb.edu/kernel-api/re256.html (http://www.fsl.cs.sunysb.edu/kernel-api/re256.html)

http://www.fsl.cs.sunysb.edu/kernel-api/re257.html (http://www.fsl.cs.sunysb.edu/kernel-api/re257.html)

File Upload | Studio | Code Embed | Google Drive | More

Upload a file, or choose a file you've already uploaded.

File: Browse... No file selected.

+ Add Another File

Click here to find a file you've already uploaded

Comments...

Cancel | Submit Assignment