CSCI 3753: Operating Systems
Fall 2017

Programming Assignment Two
Due Date and Time: 11:55 PM, Tuesday, October 10, 2017

Goal: Using Linux Kernel Module Programming to Code a Character Device Driver

Introduction

This assignment is all about getting familiar with Linux Modules and Linux Device Drivers. At first you will learn how to code a Linux kernel module (LKM), how to install the module and how to run the module. Then you will write a module that will enable you to install a Linux device driver. This assignment write-up is structured as follows:

1. What is a Linux kernel module?
2. How Linux kernel module works.
3. How to code a Linux kernel module.
4. What are Linux device drivers?
5. How do device drivers work?
6. How to code a device driver using Linux kernel modules.

You may use the VM from the first programming assignment, or any Linux installation that runs on your laptop. It will be useful to obtain the current version of the Linux kernel running on your machine if you choose the latter option. It is recommended that you use the VM. To obtain the current version of Linux kernel, follow the steps below:

1. Go to a shell terminal and type **uname –r**
2. You will get an output like x.y.z-ab-something else
   a. X is the major number
   b. Y is the minor number. If even that means the version is stable. If it's odd that means it's still in experimental version
   c. Z is the revision number.
3. To check the source code, you can go to this folder **/usr/src/$(uname -r)**

Loadable Kernel Modules (LKM)

LKMs are object files that are used to extend the running kernel's functionalities of the current operating system. This is basically a piece of binary code that can be inserted and installed in the kernel on the fly. As you know if you want to make a change in the current OS, after you make changes, you have to reboot your computer, like what you did in the first assignment when adding a system call. After you reboot, the changes that you made are installed in the kernel.

Now as you can see, this approach is a bit painstaking. To make this approach more dynamic, LKMs are introduced where you can add extensions to the kernel on the fly without the need to reboot. This comes very handy when you are trying to work with some device and just be done with it very fast and then uninstall the device without needing to reboot, thus saving time and energy and also space, because you can uninstall the module after your work is done.

How to work with modules

1. Get the file *helloModule.c* and store it in a folder named "module"
2. Open the file
3. The init.h is required for the initialization of the module and the module.h is required to let the kernel know that this is an LKM.
4. I have coded two simple functions in the module code namely hello_init() and hello_exit(). I want hello_init to execute when the module starts to work and hello_exit when the module gets uninstalled. To make sure this happens, at the end of the code I have added these two lines.
   a. module_init (hello_init)
   b. module_exit (hello_exit)

   What this means is that when the module is getting started, the kernel follows the function pointed to by module_init() and executes that function. Similarly, when the module is uninstalled, the kernel follows the function pointed to by module_exit(), in this case, the function hello_exit().

5. As you are coding in the kernel, you cannot use the printf( ) function. Instead you have to use the function **printk**. The **KERN_ALERT** is used to let the kernel know the importance of the message you are trying to print with the printk function. If it's KERN_ALERT then the message will be written in the log file in the location **/var/log/syslog** file.   The contents of the log file can be seen from the terminal using the command **dmesg** or **sudo tail –f /var/log/syslog**. There are other kernel message importance levels too (KERN_INFO, KERN_EMERG etc).
6. To check what is happening, you can type **dmesg** or **sudo tail –f /var/log/syslog** in another terminal and check what is happening when you are trying to install the module.
7. Now you have to write a makefile. Create a file named Makefile and type the following line in it
   a. **obj-m:=helloModule.o**
   b. here m means module. You are telling the compiler to create a module object named helloModule.o
8. Now to compile the module, type in the terminal "**make –C /lib/modules/$(uname -r)/build M=$PWD modules**"
9. Now in the command prompt type the following: **ls**
10. 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.
11. Now in the terminal type "**sudo insmod helloModule.ko**".
12. Now if you type **lsmod** you will see your module is now inserted in the kernel.
13. Now type "**dmesg**" or "**sudo tail /var/log/syslog**" and you will see expected output is printed, because the hello_init() function was executed when the module was installed.

14. Now to remove the kernel type "**sudo rmmod helloModule**" and then the module will be removed as it can be ascertained by typing the lsmod command. Type dmesg to see if the expected output is printed .

Device Driver

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 numbers associated with it, namely major and minor numbers. The major number is a number that is unique to every device driver and the minor number is to differentiate all the devices belonging to that device driver. So for example, for a hard disk, there are many partitions. To differentiate the hard disk device driver, major number is used whereas to differentiate the different partitions, minor number is used.

Kernel major number

| Driver 1 |
| Driver 2 |
| Driver 3 |

Device 11
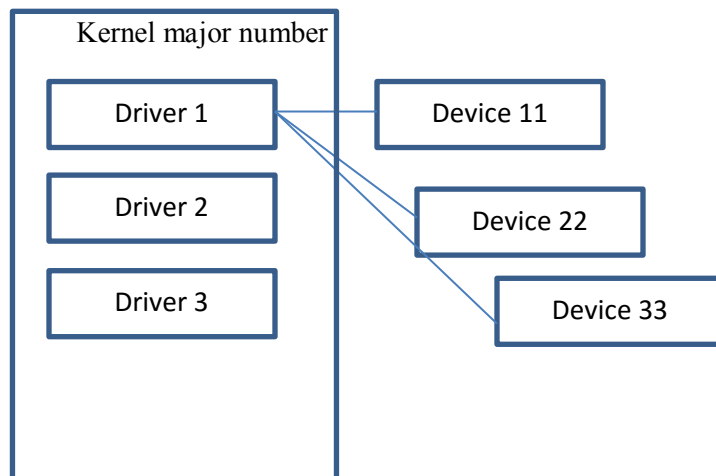
Device 22

Device 33

As it can be seen from the above diagram, there are three drivers namely Driver1, Driver2 and Driver 3. The numbers 1, 2 and 3 are the major numbers. These major numbers are associated with device drivers in the kernel to differentiate one device driver from another. To code a Device Driver, the major number has to be unique.

Also, Device Driver 1 works with Device 11, Device 22, Device 33. The numbers 11, 22 and 33 associated with the devices are called the minor numbers that are used to differentiate the devices associated with one particular device driver.

For example, if you type **ls –l /dev | grep sda**, this will give you all the device files (or the current partitions) associated with the hard disk device drivers. You will see the partitions are listed with their corresponding major and minor numbers.

There are two kinds of device drivers namely Character Device Driver and Block Device Driver.

Character Device Driver

1. Reads from the device character by character
2. Writes to the device character by character
3. Operates blocking mode, which means when a user writes info to the device, he/she must wait until the device completes execution. They are most common of all device drivers.

Block Device Driver

1. Reads large chunks of information.
2. Very CPU intensive, takes some time to finish the execution.
3. They are asynchronous, a user does not need to wait for the reading and writing to be completed.

Creating Device File for 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. If you type in the terminal "**ls /dev**" you can see all the device files in the machine. You have to create a file in this folder to work with the character device driver you will be coding. 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/simple_character_device c 240 0**" where 'c' is for creating a character driver, '777' 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 and 'simple_character_device' is the name of the device file.

The major number that you will be giving should be unique. Check **/usr/src/linux/documentation.ide/ide.txt** to check for the current devices and their major numbers in your machine. You will need to use a major number that is not taken by any of the device drivers currently installed in your machine.

The assignment

In this assignment, what you have to do is to code a simple 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 testapp that you will be creating from the user space.

So here are the 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.
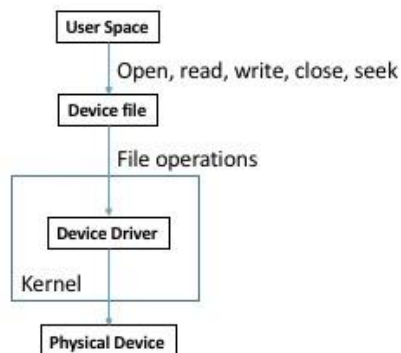
The above diagram is an overview of what is going on when you are working with a device driver. From the user-space, you will be trying to access the device file. The device file is associated with a particular device driver. When from the user-space you issue the commands open, read, write, lseek or close, e.g. when you echo hello >file.txt, the operations performed are open the file, write "hello" to the file and then close the file sequentially. Similarly, when you type cat file.txt, the operations performed are 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 (remember major number?). 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.

In this assignment, we will be only writing our data in the device file instead of the actual physical device. In your extra credit (more on extra credit section), you will be asked to create an emulated device and perform file operations for that device from the user-space using the device driver module you coded for that emulated device.

Outline of the device driver code:

1. First some header files need to be included. With the other header files necessary for module programming, you will also need to include two more. They are **linux/fs.h** to get the functions that are related to device driver coding and **asm/uaccess.h** to enable you to get data from userspace to kernel and vice versa.

2. Declare the init and exit functions as you do for module programming and make the module_init and module_exit to point to those functions. In the init function, you have to register the character driver using the function **register_chrdev()** function. This function takes three parameters namely 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.

3. Similarly, in the exit function, you have to unregister the driver using the function **unregister_chrdev()**. This function takes the major number and the name of the character driver you want gone. Check google if you have any problems regarding these two functions.

4. Now you would want the device driver to perform some file operations. For that you need the **file_operations** structure which you can find in the (**lib/modules/$(uname – r)/build/include/linux/fs.h**) header file. Check the file_operations structure in the header file and create a similar structure with the same file_operations type and with a different name because you want your device driver to perform only a few of those operations. For this assignment, you have to perform open, close, seek, read and write operations only. So you have to include the five corresponding function pointers (.open, .close, .llseek, .read, .write) for these five operations and make them point to the five functions you will be writing for these operations.

5. Inside the device driver, the function you will be writing to open the file takes two parameters. The first one is the exact inode which represents the actual physical file on the hard disk and the second parameter is the abstract open file 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 until now. Do the same thing for the close function.

6. The seek function takes three parameters. The first one is the file pointer, the second one is offset whose values is interpreted using the third parameter, whence. The seek function changes the value of the current position of the opened file. 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. 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.

7. You will use a dynamically allocated kernel buffer 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. These functions, defined in *<linux/slab.h>*, are:

    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( ). Don't free memory not originally allocated by kmalloc( ) or you will run into trouble.

8. The data from kernel cannot be used in the userspace. So you have to use two functions inside the device driver namely **copy_to_user** and **copy_from_user**. Each of these two functions takes three parameters.

    a. **copy_to_user(destination, source, size)** to get data from kernel to userspace
    b. **copy_from_user(destination, source, size)** to get data from userspace to kernel

9. The read function takes four parameters. The first one is the file pointer, the second one is the user space buffer where you will be storing your read data, the third one is the number of space available in the userspace buffer and the last one is 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 defined

above in the code. You just have to <mark>copy the data from the device_buffer to buffer and print that in the terminal</mark>. Use the function copy_to_user() to copy data from the device_buffer to the userspace buffer that is buffer variable that's present in the function's arguments. You will also have to <mark>print the number of bytes read in the file in the log file</mark> (/var/log/syslog).

10. The write function does the same thing. What you try to write is stored in the userspace buffer and then it is copied to device_driver variable and then it is written in the device file opened. All you have to do is to use the function copy_from_user() to copy data to buffer to device_buffer. Write starts from the current position of the opened file. If the current position is before the end of the file, this operation will overwrite some of the bytes that were written earlier. <mark>In both read and write functions, you have to make sure the offset is properly set.</mark> You will also have to <mark>print the number of bytes written in the file in the log file</mark> (/var/log/syslog).

Installing the module

1. Create your makefile as described in the previous sections when working with the helloModule.c
2. Edit your Makefile
3. Compile your module using the make command you used before.
4. Install your module by the insmod command.
5. Check the log file as described in previous sections to check if it's properly installed. Check with the "**cat /proc/devices**" command.

Creating a device file and test the driver code

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

Writing an interactive test program

1. Now you have to write an interactive test program that will allow you to read from, write to and seek in the device file
2. You have to use the location of the device file you created in the /dev folder while writing the test code.
3. Your interactive program at first should give the user the following options:
    a. Press r to read from device
    b. Press w to write to the device
    c. Press s to seek into the device
    d. Press e to exit from the device
    e. Press anything else to keep reading or writing from the device
    f. Enter command:
4. 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:
    a. 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:
    a) Data read from the device:

5. If the user presses 'w' then you should ask for the data to be written from the user. The format should be like this:
    a. Enter data you want to write to the device:
6. 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):
7. If the user presses 'e' then you should quit the testapp.
8. If the user presses something else, you should continue giving the user the options like you did in step 3 in this section.

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 the 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.

2. Or, create a simple block device driver and do the same file operations like you did for character device driver.