Programming Assignment 2

Coding a Simple Character Device Driver

CSCI-3753 Operating Systems

University of Colorado at Boulder

Due Date: June 19th, 2017

Task: 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, 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 module programming

How to get the current version of kernel running in your machine?

1. Go to 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 its odd that means its 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)**

<u>Loadable Kernel Modules(LKM)</u>:

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.

<u>How to work with modules</u>:

1. Get the 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 a 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().

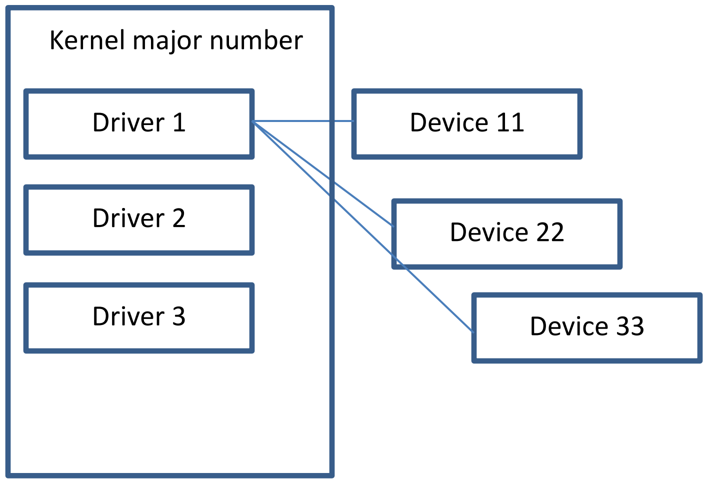
5. As you are coding in the kernel, you cannot use printf function. Instead you have to use the function **printk** function. The **KERN_ALERT** is used to let the kernel know the importance of the message you are trying to print with the printk function. If it's KERN_ALERT then the message will be written in the log file in the location **/var/log/syslog** file. the content 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 expected output is printed .

<u>**Device Driver**</u>:

Remember that in linux everything is a file. So even a device is a file and that file can be read by using the proper device driver. All device drivers have two numbers associated with it, namely major and minor number. 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.



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 Device11, Device22, Device33. The numbers 11,22 and 33 associated with the devices are called the minor numbers which are used to differentiate the devices associated with one particular device driver.

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 the user writes info to the device, must wait before the device finishes 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, the 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 major numbers in your machine.

<u>**The assignment**</u>:

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 and write to and from that file from a test file 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 file
6. From that test file, make an interactive program that will allow you to read from or write to that device file. The test program will give you a host options to choose from like do you ant to read or write or exit the program and the program will exit completely when I tell it to exit.

<u>**Skeleton of the code**</u>:

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 t 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, read and write operations only. So you have to include the four corresponding function pointers (.open, .close, .read, .write) for these four operations and make them point to the four functions you will be writing for these operations.
5. The function you will be writing to open the file takes two parameters. The first one is the exact inode and the second parameter is the file. 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. As you know, the data from kernel cannot be used in the userspace. So you have to use two functions namely copy_to_user and copy_frm_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
7. 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 read from the device will be stored in the device_buffer array defined above in the code. You just have to copy the data from the device_buffer to buffer and print that in the terminal. 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.
8. 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. Be careful of how to use the offset variable so that you don't overwrite the previous data. In both read and write functions, you have to make sure the offset is properly set. Check google if you have any trouble.

<u>Installing the module</u>:

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 its properly installed. Check with the "**cat /proc/devices**" command.

<u>Creating a device file and test the driver code</u>:

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.
3. Then create a test file that will enable the user to write and read into that file with interactive options. (like give the user the options to read and write the device and exit)

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