International Journal of Scientific & Engineering Research, Volume 11, Issue 10, October-2020
ISSN 2229-5518

Device driver programming: Implementation of Linux USB Device Driver

Sharanya Mukherjee

Abstract— This paper focuses on the implementation of USB Pen-drive utility program aims at handling the sensible information of operating of modules and kernels where we tend to find out how the module is loaded in the kernel for the execution of the code. The paper also jointly aims at registering the device within the UNIX system. The module where we tend to write the fundamental registration and de-registration program for the USB pen-drive ,has been written in C language . The module file is saved with .o extension within the kernel file.

Index Terms—Device Driver programming, USB device driver, kernel, UNIX, module, Operating system, Linux,

1 INTRODUCTION

The purpose of a device driver is to handle requests created by the kernel with reference to a selected style of device. There's a well-defined and consistent interface for the kernel to form these requests. By noninflected device-specific code in device drivers and by having the same interface to the kernel, adding a brand new device is less complicated. a device driver could be a software package module that resides at intervals the Digital OS kernel and is that the software package interface to a hardware device or devices. A hardware device could be a peripheral, like a controller, tape controller, or network controller device. In general, there's one utility program for every style of hardware device.

2 LITERATURE REVIEW

A device driver, run as a part of the kernel software system, manages every of the device controllers on the system. Often, one utility manages a complete set of identical device controller interfaces. With Digital OS, you'll be able to statically assemble additional device drivers into the kernel than there are physical devices within the hardware system. At boot time, the car configuration software system determines that the physical devices are accessible and useful and might turn out an accurate run-time configuration for that instance of the running kernel. Similarly, once a driver is dynamically designed, the kernel performs the configuration sequence for every instance of the physical device.

As explicit antecedence, the kernel makes requests of a driver by line of work the driver's commonplace entry points (such as the probe, attach, open, read, write, shut entry points). Within the case of I/O requests like read and write, it's typical that the device causes associate degree interrupt upon completion of every I/O operation. Thus, a write supervisor call instruction from a user program might end in many calls on the interrupt entry purpose additionally to the initial invoke the write entry purpose. This can be the case once the write request is divided into many partial transfers at the driving force level. Device drivers, in turn, create calls upon kernel support interfaces to perform the tasks mentioned earlier. The device register offset definitions giving the layout of the management registers for a tool are a part of the supply for a tool driver. Device drivers, not like the remainder of the kernel, will access and modify these registers. Digital OS provides generic CSR I/O access kernel interfaces that enable device drivers to browse from and write to those registers.

Firstly, we should be aware of the fact that whether a driver for a USB device is there or not on a Linux system, a valid USB device will always be detected at the hardware and kernel spaces of a USB-enabled Linux system, since it is designed (and detected) as per the USB protocol specifications. At the hardware level, the Hardware-space detection is done by the USB host controller which is typically a native bus device The corresponding host controller driver would pick and translate the low-level physical layer information into higher-level USB protocol-specific information. The USB protocol formatted information about the USB device is then populated into the generic USB core layer (the USB core driver) in kernel-space, thus enabling the detection of a USB device in kernel-space, even without having its specific driver available.

3 LINUX COMMANDS USED

A basic listing of all detected USB devices can be obtained using the lsusb command, as root

- lsusb (It is used to view all the usb devices connected)
- lsusb –v (Is is used to give a detailed view of the usb devices
- connected)
- cat /sys/kernel/debug/usb/devices (It also gives detail info of the
- connected devices including the details of interfaces and endpoints)
- sudo –i (in order to log in from root)
- make (to compile a module)
- insmod (inserting a module)
- rmmod (removing a module)
- modprobe (inserting back a module)
- mount (lists all mounted devices)
- fdisk –l (Gives detailed info of the mounted disks)
- mkdir (create new directory)

4 IMPLEMENTATION

The source code is very simple. The header files used are linux/module.h, linux/kernel.h and linux/usb.h. The most important structure is our usb_driver structure. It is the core of our code. It includes 4 main branches. Name, id , prob and disconnect. There are three functions defined inside the structure of usb_driver. Name is simply the name of the device driver.

First of all let's talk about our ID table which is taken in the form of an array. It is because we want to detect as many devices we want. The ID table is fully dependent upon two things. The Vendor ID and the Product ID. Whenever a device is connected this ID_Table function just matches whether the vendor ID and product ID matches to any of the stored ID's. If yes, then it takes control of the device. However , if it doesn't match then it lets go. The vendor ID and Product ID are saved in the USB_DEVICE(vendor,product) function. At the end we have a terminating condition for a situation in which our driver receives null.

Moving on towards the prob function. It is called whenever a device is connected to any of the serial ports in our system. In this function we have two arguments one is the interface pointer while the other is for the USB ID(it is the same as the ID, mentioned above). Other than the arguments I have initialized two different structures. One of type usb_host_interface while other is of type usb_endpoint_descriptor. The host interface has all the information about the interface of the device you have connected. Meanwhile the endpoint descriptor will have the detailed information about the endpoints available in our device. These both are just used in order to print the number of endpoints and to print out the attributes and details of the endpoints. After that we have stored the information of the usb_host into a pointer and then printed the number of endpoints. Then we had a loop from 1 to the number of endpoints available.

Then one by one we have printed the addresses, MXPS and the type of endpoint for all the endpoints inside the device. After that we have a function named disconnect. The name suggests it very well. Yes, the disconnect function is only called when the device is disconnected. We don't have to do anything special in this function. Therefore, I have just printed a statement on the console instead. Now talking about the init function. We all know that init function is the function that is called when you are installing a mod. Therefore, we have registered the module as a usb device driver in the init function. However, if the usb registration is not done properly then the return value will be -1 else than that it will be greater than 0.

To conclude, we can see 5 different lines. The first two lines of module_init(usb_init) is just telling the kernel that the init function for this module is usb_init. Same goes for our favourite exit function. After that we have taken license from GPL. Without this license I cannot call the functions such as usb_register or usb_deregister. It is because the ubuntu has a permission set not letting everyone use those special functions. But when you give a license of GPL then no one can stop you from taking any function from the usb.h library. Next line just shows the author and after that the module_description is showing just a small description of our code.

4.1 MODULE CODE

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/usb.h>
//probe function
static int pen_probe (struct usb_interface *interface, const struct usb_device_id *id)
{
printk(KERN_INFO "[*] OS Pen Drive (%04X:%04X) plugged\n" , id->idVendor, id->idProduct);
return 0 ;
}
//disconnect
static void pen_disconnect (struct usb_interface *interface) {
printk(KERN_INFO "[*] Pen Drive removed\n" );
}
//usb_device_id
static struct usb_device_id pen_table [] = {
//( 0781:5567)
{ USB_DEVICE( 0x0781 , 0x5567 ) }, //vendor_id,product_id
{} /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, pen_table);
// usb_driver
static struct usb_driver pen_driver =
{
.name = "MYPENDRIVE" ,
.id_table = pen_table, //usb_device_id
.probe = pen_probe,
.disconnect = pen_disconnect,
};
static int pen_init ( void ) {
int ret = -1 ; printk(KERN_INFO "[*] Constructor ofDriver" );
printk(KERN_INFO "\tRegistering Driver with Kernel" );
ret = usb_register(&pen_driver);
printk(KERN_INFO "\tRegistration is complete" );
return ret;
}
static void pen_exit ( void ) {
//deregister
printk(KERN_INFO "[*] Destructor of driver" );
usb_deregister(&pen_driver);
printk(KERN_INFO "\t unregistration complete!" );
}
module_init(pen_init);
module_exit(pen_exit);
MODULE_LICENSE( "GPL" );
MODULE_AUTHOR( "SHARANYA" );
MODULE_DESCRIPTION( "USB Pen Registration Driver" );
```

4.2 KERNEL CODE

```
obj-m := usb.o
KERNEL_DIR= /lib/modules/ $(shell uname -r) /build
PWD= $(shell pwd)
all:
$(MAKE) -C $
(KERNEL_DIR) M= $(shell pwd) modules
clean:
rm -rf *.o *.ko *.mod.* *.symvers *.order *~
```

5 RESULT

For this paper, we will be using a SanDisk Pen-drive and write a driver for it. The vendor ID and Product ID can be obtained from the following command-
"Lsusb-v". We can see from the image below that the vendor id for the USB Stick is 0x0781 and Product id is 0x5567.

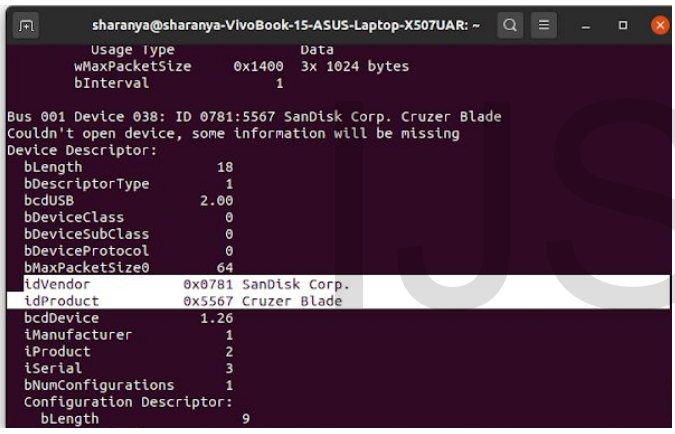

Fig 1: Finding the vendor ID and product ID of our USB pen-drive with "lsusb-v" command.

We further compile our module using the command "make".

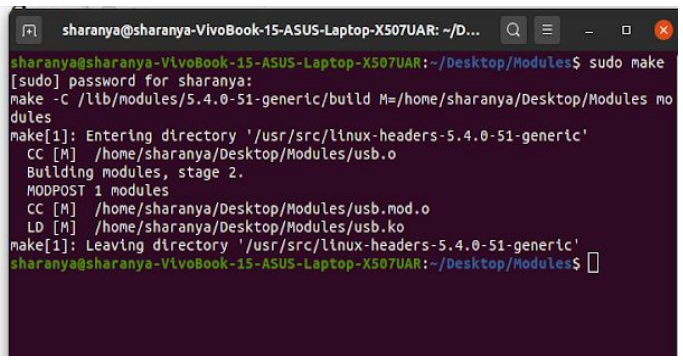

Fig 2: Compiling the module with 'make' command

After successfully building our module we need to insert it using the "insmod" command.

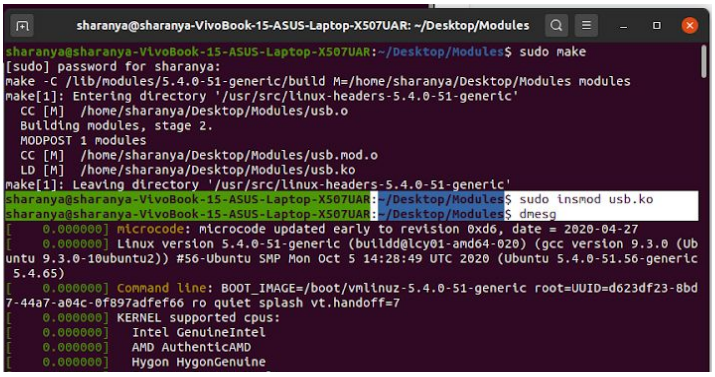

Fig 3: Inserting the module with "insmod" command.

When the pen-drive is connected it gets registered by the device driver.

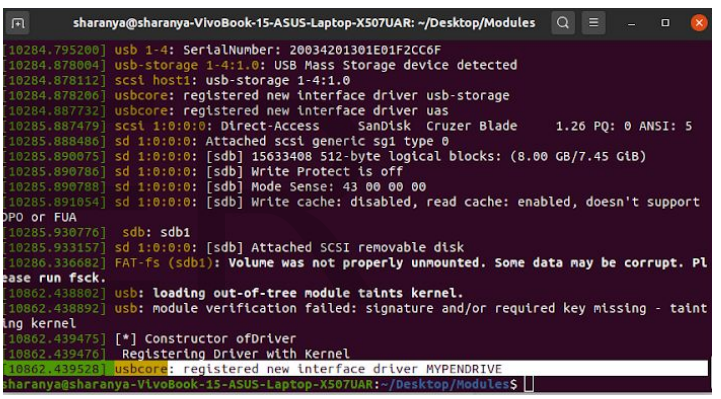

Fig 4: Pen-drive gets registered by the device driver

For Deregistering the pendrive and removing the modules we use the "rmmod" command

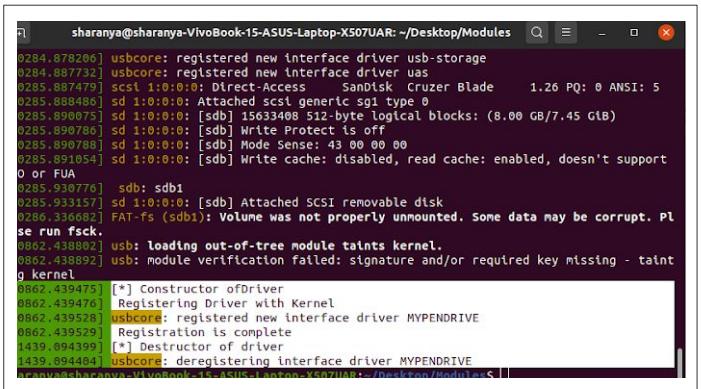

Fig 5: Removing the module with "rmmod" command

4 Conclusion

We have implemented a code for usb device drivers where we have shown registering of a usb pen drive in linux system. A directory was made which had 2 files that are Kernel and Module file. The module coding has been done in C language and Kernel helps in loading of module for device driver. When the pen drive is connected, it gets registered by the device driver and as soon as we remove it, it gets disconnected and the registration remains incomplete. We conclude that our kernel and module have been properly implemented for the formation of device driver and usb pen drive is getting registered properly.

References

[1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In Symposium on Principles of Programming Languages (POPL), pages 1–3, Portland, Oregon, January 2002.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), volume 35 of Operating System Review, pages 73–88, Banff, Alberta, Canada, October 2001.

[3] K. Crary and G. Morrisett. Type structure for low-level programming languages. In International Colloquium on Automata, Languages, and Programming (ICALP) 1999, volume 1644 of Lecture Notes in Computer Science, pages 40–54, Prague, Czech Republic, July 1999. Springer Verlag.

[4] R. DeLine and M. F ahndrich. Enforcing high-level protocols in low-level software. In Proceedings of the ACM Conference on Programming Laguage Design and Implementation (PLDI), pages 59–69, Snowbird, Utah, June 2001.

[5] G. J. Holzmann. Software analysis and model checking. In Computer Aided Verification, 14th International Conference (CAV), volume 2404 of Lecture Notes on Computer Science, pages 1–16, Copenhagen, Denmark, July 2002. Springer Verlag.

[6] F. M´erillon, L. R´eveill`ere, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI), pages 17–30, San Diego, California, October 2000.

[7] M. O'Nils and A. Jantsch. Operating system sensitive device driver synthesis from implementation independent protocol specification. In Proceedings of Design
Automation and Test in Europe (DATE), pages 562–567, Munich, Germany, March 1999.

[8] S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: from design to implementation—application to video device drivers generation. IEEE Transactions on Software Engineering, 25(3):363–377, May-June 1999.

[9] S. Wang and S. Malik. Synthesizing operating system based device drivers in embedded systems. In Proceedings of the First International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Newport Beach, CA, October 2003.