"IMPLEMENTATION OF DEVICE DRIVER PROGRAMMING (USB PEN DRIVE)"

PROJECT REPORT

By

RAHUL SAW(17BCE1179)

ANISH DHANUKA(17BCE1256)

DIBYANSHU GAUTAM(17BCE1328)

Slot: A2

Name of faculty: **Dr. KUMAR R**

(SCSE)

VIT®

UNIVERSITY

(Estd. u/s 3 of UGC Act 1956)

VELLORE ■ CHENNAI

www.vit.ac.in

April 2019

CERTIFICATE

This is to certify that the project work entitled "**Implementation Of Device Driver Programming (USB Pen Drive)"** that is being submitted by "**Rahul Saw(17bce1179) , Anish Dhanuka(17bce1256) and Dibyanshu Gautam(17bce1328)**" for Operating Systems (CSE2005) is a record of bonafide work done under my supervision-**Prof Kumar R**. The contents of this Project work, in full or in parts, have neither been taken from any other source nor have been submitted for any other CAL course.

Place: Chennai

Date: 10/04/19

ACKNOWLEDGEMENTS

We would like to thank our respected Operating Systems teacher Prof. Kumar R for providing us the support and guidance for our project and teaching us all the important concepts involved to successfully end this task.

We would also like to thank all the lab assistants who helped us in conducting all the experiments successfully.

Also, we all are grateful to VIT UNIVERSITY,CHENNAI Management for providing us with such a good environment to work out on our topic and providing this one opportunity out of many others they are yet to provide.

ABSTRACT

As a device driver program our project aims at implementing USB Pen Drive for the working of modules and kernels and how the modules are loaded in kernel for execution of the code. It also aims at the registration and deregistration of the device in the UNIX system. The drawback with the general device driver is that it often manages an entire set of identical device controller interfaces. But with the digital UNIX we can statistically configure more device drivers into the kernel. For the implementation, a directory will be made which will have 2 files that are kernel and module file and the coding will be done in C language. The reason for using such technique is that when we isolate a device specific code in device drivers and then by having a consistent interface to the kernel, adding a new device gets easier. Since UNIX is a monolithic kernel, so kernel and module form is to be implemented together to avoid recompiling of the kernel when the driver is added. As a result, when a pen drive is connected to device driver and as soon as we remove it, it will be disconnected. So when USB pen drive and the device are registered properly then we would conclude that our kernel and module have been properly implemented. The main advantage of such model is that when a USB controller module is updated it does not require a complete recompilation, which as a result reduces the errors. Hence, adding a new device only need a new module driver rather than a new kernel, which offers much flexibility.

INTRODUCTION

A program that controls a selected form of device that could be attached to our computer is a driver. There are device drivers for printers, displays, read-only storage readers, floppy disk drives, and so on. A driver could be a software system element that lets the software system and a device communicate with one another. A driver communicates with the device to that the hardware connects through the pc bus. The driver problems commands to the device, once the job program interrupts a routine within the driver. Once the device sends information back to the ktodriver, the driver could invoke routines within the original calling program.

The purpose of a device driver is to handle requests created by the kernel with reference to a specific form of device. There's a well-defined and consistent interface for the kernel to form these requests. By using analytic device-specific code in device drivers and by having a uniform interface to the kernel, adding a replacement device is simpler. a device driver could be a software system module that resides among the Digital UNIX system kernel and is that the code 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 driver for every form of hardware device.

The importance the device driver can be seen by the functions of a driver that is encapsulation which hides low-level device protocol details from the client unification makes similar devices look the same protection with the cooperation of OS only authorized applications can use device Multiplexing such as the Multiple applications can use the device concurrently. Device drivers isolate low-level, device-specific details from the system calls, which can remain general and uncomplicated. Because each device differs so, kernels cannot practically handle all the possibilities. Instead, each configured device plugs a device driver into the kernel. To add a new device or capability to the system, just plug in its driver.

Motivation

A device driver, run as a part of the kernel software system, manages every of the device controllers on the system. Often, one device driver manages a complete set of identical device controller interfaces. With Digital OS, we will be able to statically put together a lot of device drivers into the kernel than there are physical devices within the hardware system. At boot time, the auto configuration software system determines that of the physical devices are accessible and useful and may 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 expressed previously, the kernel makes requests of a driver by job the driver's normal entry points (such because the probe, attach, open, read, write, close entry points). In case of I/O requests like read and write, it's typical that the device causes an interrupt upon completion of every I/O operation. Thus, a write call 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 segmented into many partial transfers at the driver level. .

The device register offset definitions giving the layout of the control registers for a device are a part of the supply for a device driver. Device drivers, not like the rest of the kernel, will access and modify these registers. Digital UNIX provides generic CSR I/O access kernel interfaces that permit device drivers to scan from and write to those registers.

To perform some useful functions in the device, processes need to be accessed to the peripherals connected to the computer, that are thus controlled by the kernels through device drivers. As device drivers are thus handled differently by each kind of the kernel design, but in each and every case, the kernel has to provide the I/O to allow the drivers to physically access their devices through some port in the device or memory location.

DESCRIPTION OF MODULES

The Modules involved are:

- Module
- Drive Insertion
- Authentication
- Kernel Code
- Display of Pen drive Details
- Registration and Deregistration

Module

A module is an object file prepared in a special way. The Linux kernel can load a module to its address space and link the module with itself. The Linux kernel is written in 2 languages: C and assembler (the architecture dependent parts). The development of drivers for Linux OS is possible only in C and assembler languages, but not in C++ language (as for the Microsoft Windows kernel). It is connected with the fact that the kernel source pieces of code, namely, header files, can contain C++ key words such as new, delete and the assembler pieces of code can contain the '::' lexeme.

The module code is executed in the kernel context. It rests some additional responsibility in the developer: if there is an error in the user level program, the results of this error will affect mainly the user program; if an error occurs in the kernel module, it may affect the whole system. But one of the specifics of the Linux kernel is a rather high resistance to errors in the modules' code. If there is a noncritical error in a module (such as the dereferencing of the null pointer), the oops message will be displayed (oops is a deviation from the normal work of Linux and in this case, the kernel creates a log record with the error description). Then, the module, in which the error appeared, is unloaded, while the kernel itself and the rest of modules continue working. However, after the oops message, the system kernel can often be in an inconsistent state and the further work may lead to the kernel panic.

Drive Insertion

The insertion of the pen drive into the module is done with the help of the module file. The module is present in the directory of the laptop and as well as the pen drive. The insertion is done with the help of the command 'lsusb' . It will show the pen drive connected to which port and it will help in understanding the different port details of the laptop or to the system into which the

program is installed into and displaying port number and to check whether the pen drive is properly connected to the port and the port number is assigned into the program for accessing the details of the pen drive.

Kernel

The kernel and its modules are built into a practically single program module. That is why it is worth remembering that within one program module, one global name space is used. To clutter up the global name space minimally, one should monitor that the module exports only the necessary minimum of global characters and that all exported global characters have the unique names (the good practice is to add the name of the module, which exports the character, to the name of the character as a prefix). Here we are using make function to implement the details of kernel.

Display of pen drive Details

The details of the USB pen drive are displayed in a sequential fashion. Showing the things needed to be displayed like vendor ID, product ID, manufacture ID etc. Thus the internal details was able to be accessed by the USB drive which is the work of the drive is accomplished. The command used is 'lsusb -v'. to get the details.

Implementation:

Code

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/usb.h>
#include<linux/uaccess.h>

#define MIN(a,b) (((a) <= (b)) ? (a) : (b))
#define BULK_EP_OUT 0x01
#define BULK_EP_IN 0x82
#define MAX_PKT_SIZE 512

static struct usb_device *device;
static struct usb_class_driver class;
static unsigned char bulk_buf[MAX_PKT_SIZE];

static int pen_open(struct inode *i, struct file *f)
{
```

```
        return 0;
}
static int pen_close(struct inode *i, struct file *f)
{
        return 0;
}
static ssize_t pen_read(struct file *f, char __user *buf, size_t cnt, loff_t *off)
{
        int retval;
        int read_cnt;

        /* Read the data from the bulk endpoint */
        retval = usb_bulk_msg(device, usb_rcvbulkpipe(device, BULK_EP_IN),
                        bulk_buf, MAX_PKT_SIZE, &read_cnt, 5000);
        if (retval)
        {
                printk(KERN_ERR "Bulk message returned %d\n", retval);
                return retval;
        }
        if (copy_to_user(buf, bulk_buf, MIN(cnt, read_cnt)))
        {
                return -EFAULT;
        }

        return MIN(cnt, read_cnt);
}
static ssize_t pen_write(struct file *f, const char __user *buf, size_t cnt,
                                                        loff_t *off)
{
        int retval;
        int wrote_cnt = MIN(cnt, MAX_PKT_SIZE);

        if (copy_from_user(bulk_buf, buf, MIN(cnt, MAX_PKT_SIZE)))
        {
                return -EFAULT;
        }

        /* Write the data into the bulk endpoint */
        retval = usb_bulk_msg(device, usb_sndbulkpipe(device, BULK_EP_OUT),
                        bulk_buf, MIN(cnt, MAX_PKT_SIZE), &wrote_cnt, 5000);
        if (retval)
        {
                printk(KERN_ERR "Bulk message returned %d\n", retval);
                return retval;
        }
```

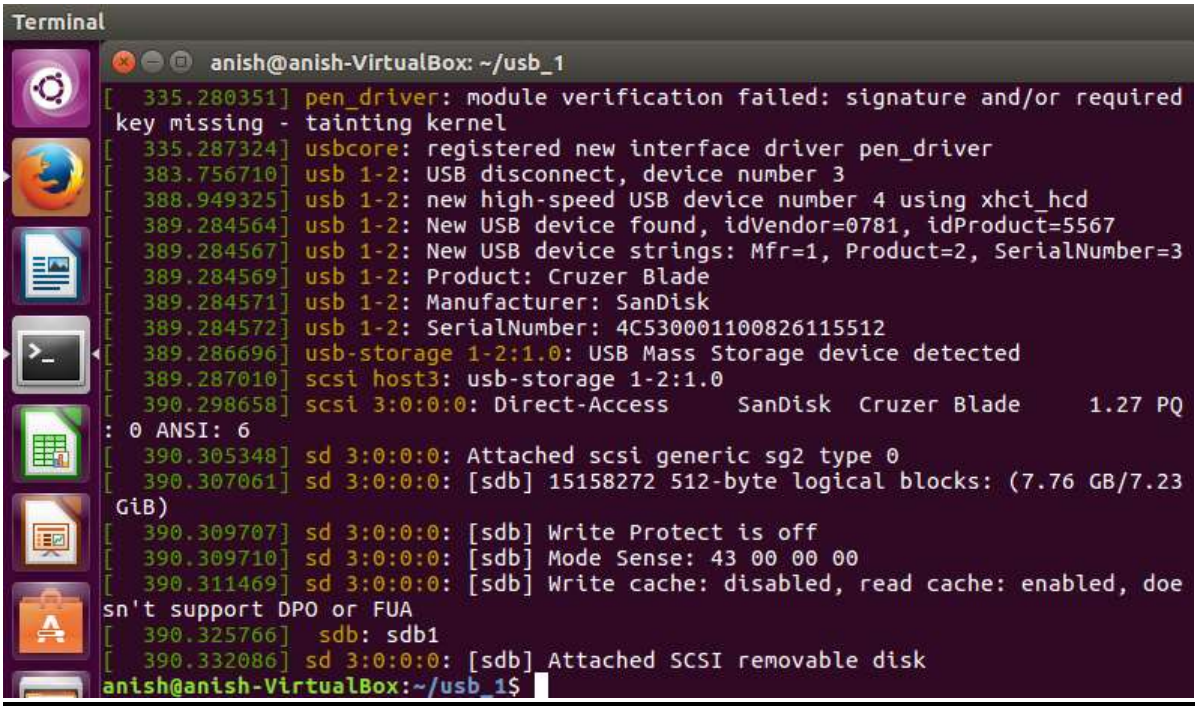
```
	return wrote_cnt;
}

static struct file_operations fops =
{
	.owner = THIS_MODULE,
	.open = pen_open,
	.release = pen_close,
	.read = pen_read,
	.write = pen_write,
};

static int pen_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
	int retval;

	device = interface_to_usbdev(interface);

	class.name = "usb/pen%d";
	class.fops = &fops;
	if ((retval = usb_register_dev(interface, &class)) < 0)
	{
		/* Something prevented us from registering this driver */
		printk(KERN_ERR "Not able to get a minor for this device.");
	}
	else
	{
		printk(KERN_INFO "Minor obtained: %d\n", interface->minor);
	}

	return retval;
}

static void pen_disconnect(struct usb_interface *interface)
{
	usb_deregister_dev(interface, &class);
}

/* Table of devices that work with this driver */
static struct usb_device_id pen_table[] =
{
	{ USB_DEVICE(0x0781, 0x5567) },
	{} /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, pen_table);
```

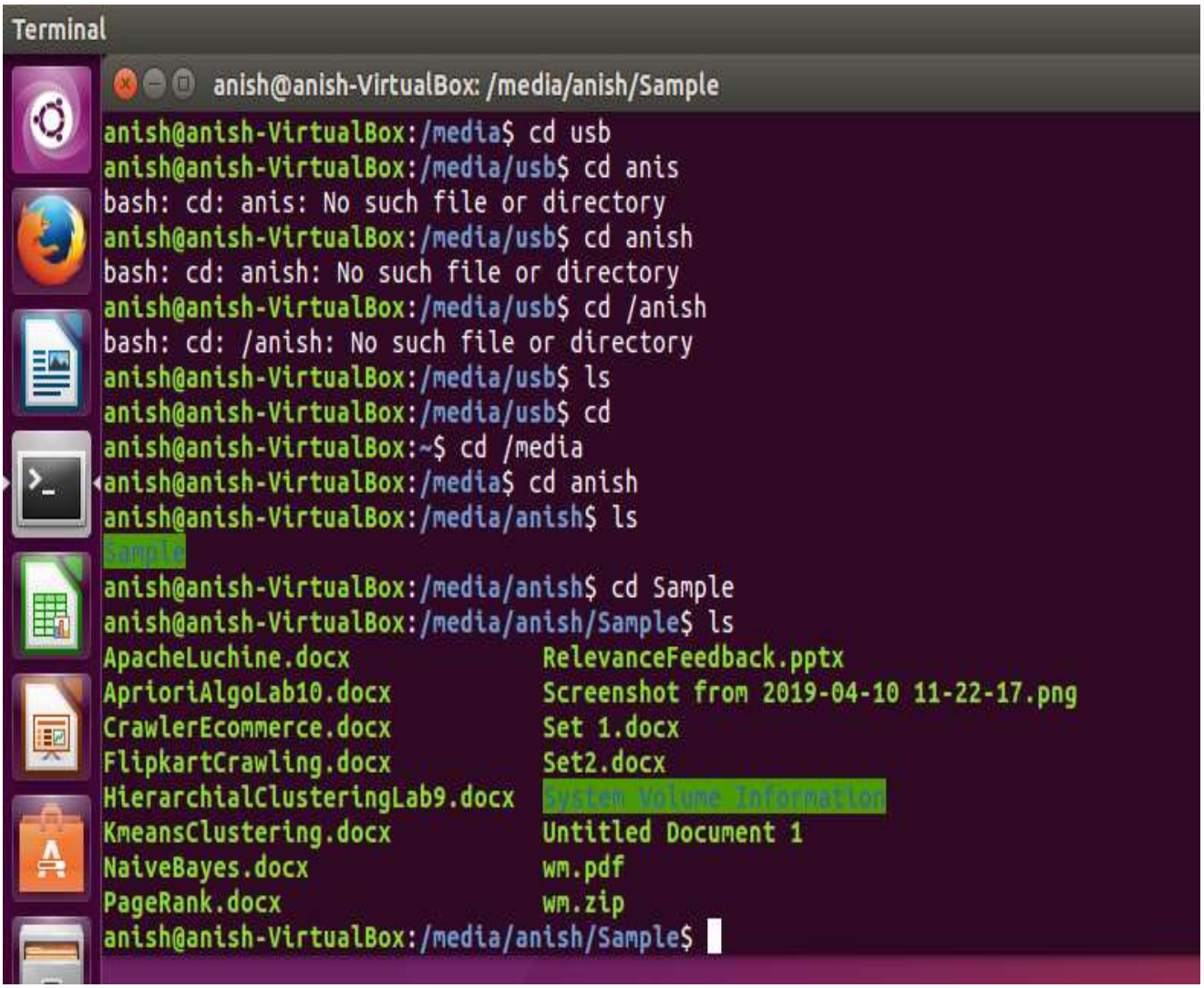
```
static struct usb_driver pen_driver =
{
        .name = "pen_driver",
        .probe = pen_probe,
        .disconnect = pen_disconnect,
        .id_table = pen_table,
};

static int __init pen_init(void)
{
        int result;

        /* Register this driver with the USB subsystem */
        if ((result = usb_register(&pen_driver)))
        {
                printk(KERN_ERR "usb_register failed. Error number %d", result);
        }
        return result;
}

static void __exit pen_exit(void)
{
        /* Deregister this driver with the USB subsystem */
        usb_deregister(&pen_driver);
}

module_init(pen_init);
module_exit(pen_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Anish <email@sarika-pugs.com>");
MODULE_DESCRIPTION("USB Pen Device Driver");
```

Makefile:

```
obj-m := pen_driver.o
KERNEL_DIR = /lib/modules/$(shell uname -r)/build
PWD   = $(shell pwd)
all:
	$(MAKE)        -C $(KERNEL_DIR) SUBDIRS=$(PWD) modules

clean:
	rm -rf *.o *.ko *.mod.* *.symvers *.order *~
```

RESULTS


Terminal
anish@anish-VirtualBox: ~/usb_1
[335.280351] pen_driver: module verification failed: signature and/or required key missing - tainting kernel
[335.287324] usbcore: registered new interface driver pen_driver
[383.756710] usb 1-2: USB disconnect, device number 3
[388.949325] usb 1-2: new high-speed USB device number 4 using xhci_hcd
[389.284564] usb 1-2: New USB device found, idVendor=0781, idProduct=5567
[389.284567] usb 1-2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[389.284569] usb 1-2: Product: Cruzer Blade
[389.284571] usb 1-2: Manufacturer: SanDisk
[389.284572] usb 1-2: SerialNumber: 4C530001100826115512
[389.286696] usb-storage 1-2:1.0: USB Mass Storage device detected
[389.287010] scsi host3: usb-storage 1-2:1.0
[390.298658] scsi 3:0:0:0: Direct-Access SanDisk Cruzer Blade 1.27 PQ: 0 ANSI: 6
[390.305348] sd 3:0:0:0: Attached scsi generic sg2 type 0
[390.307061] sd 3:0:0:0: [sdb] 15158272 512-byte logical blocks: (7.76 GB/7.23 GiB)
[390.309707] sd 3:0:0:0: [sdb] Write Protect is off
[390.309710] sd 3:0:0:0: [sdb] Mode Sense: 43 00 00 00
[390.311469] sd 3:0:0:0: [sdb] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
[390.325766] sdb: sdb1
[390.332086] sd 3:0:0:0: [sdb] Attached SCSI removable disk
anish@anish-VirtualBox:~/usb_1$


Terminal
anish@anish-VirtualBox: /media/anish/Sample
anish@anish-VirtualBox:/media$ cd usb
anish@anish-VirtualBox:/media/usb$ cd anis
bash: cd: anis: No such file or directory
anish@anish-VirtualBox:/media/usb$ cd anish
bash: cd: anish: No such file or directory
anish@anish-VirtualBox:/media/usb$ cd /anish
bash: cd: /anish: No such file or directory
anish@anish-VirtualBox:/media/usb$ ls
anish@anish-VirtualBox:/media/usb$ cd
anish@anish-VirtualBox:~$ cd /media
anish@anish-VirtualBox:/media$ cd anish
anish@anish-VirtualBox:/media/anish$ ls
Sample
anish@anish-VirtualBox:/media/anish$ cd Sample
anish@anish-VirtualBox:/media/anish/Sample$ ls
ApacheLuchine.docx
AprioriAlgoLab10.docx
CrawlerEcommerce.docx
FlipkartCrawling.docx
HierarchialClusteringLab9.docx
KmeansClustering.docx
NaiveBayes.docx
PageRank.docx
RelevanceFeedback.pptx
Screenshot from 2019-04-10 11-22-17.png
Set 1.docx
Set2.docx
System Volume Information
Untitled Document 1
wm.pdf
wm.zip
anish@anish-VirtualBox:/media/anish/Sample$

Conclusions and Future Work

Our project shows how a module is loaded into kernel and aims at registering of the device in the Linux system. This module has been written in C language where we write the basic connecting and disconnecting program for the USB PEN DRIVE is being implemented. The module file is saved with .o extension in the kernel file. USB has the following USBPcap with root hub, control and device in the user mode helps every device to connect. The driver for this USB drive should be compiled with the kernel and the module together or implemented to avoid the recompiling of the kernel with driver when the adding is needed. The Linux kernel is written in C and assembler languages. The development of these drivers for Linux is possible only in C and assembler languages. It is connected with the fact that the kernel source pieces of code and header files containing the C++ key words such as new, delete and the assembler pieces of code can contain the lexeme.

We have implemented a code for USB device driver 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, the device driver registers it and as soon as we remove it, it gets disconnected.

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.

We can also proceed further in this area by connecting IOT or wireless devices by creating device driver for them.