OS Assignment

By Ervin Ranjan , B220027CS

1) Problem Statement :

Create a Character Device Driver with the following functionality:

1) Kernel Version Check:
    1) The driver must accept an array parameter called kernel_version, which specifies the current kernel version.
    2) The driver should only be inserted if the provided kernel version matches the version used to compile the module.
    3) For example, if the module was compiled for version 6.5.1, it should only be successfully inserted if you run

insmod <your_driver_name>.ko kernel_version=6,5,1

2) Driver Insertion:

Upon successful insertion, the driver should print the assigned major and minor numbers in the kernel log (this can be checked using the dmesg command).

3) Device Read/Write Operations:

After insertion, you should write <FIRSTNAME>_<ROLLNO> to the device and read from it in two different ways:

1) Using the 'echo' command for writing and the 'cat' command for reading.

eg:

> echo "RAMESH_B220007CS" > /dev/<device_name>

> cat /dev/<device_name>

RAMESH_B220007CS

2) Using a user program written in C or any other language.

Whenever the read and write functions of the driver are called, appropriate messages should be printed in the kernel log (e.g., Read function called! or Write function called!).

2) Methodology :

I took kernel_version parameter using the

int module_param_array(const type *array, const char *name, int *num, int flags)

I have written the following functions :

1) int chr_driver_init(void) :

    This is a function that runs on the insertion of the module, I set the function to run at module insertion using

    int module_init(int (*init)(void));

    I do the following operations in this function:

    1) Check if the given kernel version matches with current linux version using macros stored in version.h –
        - LINUX_VERSION_CODE : contains the current linux version code, encoded as a number

- KERNEL_VERSION_CODE(major,minor,patch) : encodes major, minor and patch numbers of the linux version

2) Allocate memory for the device buffer
   using
   void* kmalloc(size_t size, gfp_t flags)
3) Allocate and register char device numbers (with name 'character_device')
   using
   int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name)
4) Print major and minor numbers of the device with macros MAJOR(dev) and MINOR(dev)
5) Initialize a new character device structure and set file_operations with
   void cdev_init(struct cdev *cdev, const struct file_operations *fops)
6) Add the device using
   int cdev_add(struct cdev *p, dev_t dev, unsigned count)
7) Create a device class called 'character_device_class' using
   struct class *class_create(struct module *owner, const char *name)
8) Create a device file node called 'character_device_1' and register it using
   struct device * device_create(struct class *class, struct device *parent, dev_t devt, const char *fmt, ...)

2) int chr_driver_exit(void) :
   It runs on module exit , I set it using
   void module_exit(int (*init)(void)).
   I do the following operations in this function:
   1) Free the buffer allocated for the device using
      void kfree(const void *ptr)
   2) destroy the device class using
      void class_destroy(struct class *class)
   3) destroy the device using
      void device_destroy(struct class *class, dev_t dev)
   4) destroy the structure containing metadata about the device using
      void cdev_del(struct cdev *cdev)
   5) unregister the device that is remove the major and minor numbers using
      int unregister_chrdev_region(dev_t from, unsigned int count)
3) int open(struct inode *inode, struct file *file):
   It prints "Open function called" and "Device File Opened".
4) int release(struct inode *inode, struct file *file):
   It prints "Close function called" and "Device File Closed"
5) ssize_t read(struct file *file,char *buf,size_t len,loff_t *file_offset):
   It prints "Read function called" , then I copy data from device buffer to user buffer using int copy_to_user(void __user *to, const void *from, unsigned long n), then I update the file_offset and return the number of bytes read. I also maintain a variable (buffer_pointer) to keep track of the size of the file, and only read the remaining portion if the number of bytes required to read is greater than the file.

6) ssize_t write(struct file *file,const char *buf, size_t len, loff_t *file_offset) :
It prints “Write function called” , then I copy data from user buffer to device buffer using int copy_from_user(void *to, const void __user *from, unsigned long n) then I return the data written.

I have written a Makefile to compile the driver code. I wrote data into the device by using the ‘echo’ command and redirecting its output into the file, then I used the ‘cat’ command to view the contents, for the user program, I used standard file handling functions such as fopen (to open the file in specified mode, I opened it in “r+” which means read and write mode) , fprintf (to write into the file) , fscanf(to read from the file) to write and read from device.

3) Detailed Explanation :
I have already explained the functions and operations done inside each of these functions in the Methodology section, In this section, I will focus on the steps I followed to compile the driver code , insert the module and write and read data from it.
Source code for driver (named chr_driver.c) :

```c
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/string.h>
#include <linux/version.h>

#define KERNEL_BUFFER_SIZE 512
#define MAX_PARAM_SIZE 100

dev_t dev = 0;
static struct class *dev_class;
static struct cdev chr_cdev;
uint8_t *kernel_buffer;
static int buffer_pointer;
static int kernel_version[MAX_PARAM_SIZE];
static int param_count = 0;
module_param_array(kernel_version,int,&param_count,0660);
MODULE_PARM_DESC(kernel_version,"major, minor, and patch");
static int open(struct inode *inode,struct file *file);
static int release(struct inode *inode,struct file *file);
static ssize_t read(struct file *file,char __user *buf,size_t len,loff_t *file_offset);
static ssize_t write(struct file *file,const char __user *buf,size_t len,loff_t *file_offset);
static int __init chr_driver_init(void);
static void __exit chr_driver_exit(void);

static struct file_operations fops = {
        .owner = THIS_MODULE,
        .read = read,
        .write = write,
        .open = open,
        .release = release
};
```

```c
static int open(struct inode *inode,struct file *file){
        printk(KERN_INFO "Open function called\n");
        printk(KERN_INFO "Device File Opened\n");
        return 0;
}

static int release(struct inode *inode, struct file *file){
        printk(KERN_INFO "Close function called\n");
        printk(KERN_INFO "Device File Closed\n");
        return 0;
}

static ssize_t read(struct file *file,char __user *buf,size_t len,loff_t *file_offset){
        printk(KERN_INFO "Read function called\n");
        if(len > buffer_pointer){
                len = buffer_pointer;
        }

        if(*file_offset >= buffer_pointer){
                return 0;
        }

        int to_copy = len;
        int not_copied = copy_to_user(buf,kernel_buffer,len);
        int data_read = to_copy - not_copied;
        if(not_copied){
                printk(KERN_ERR "Failed to copy data to user space\n");
                return -EFAULT;
        }


        *file_offset += data_read;
        printk(KERN_INFO "Data Read Success\n");
        return data_read;
}
```

```
static ssize_t write(struct file *file, const char __user *buf,size_t len,loff_t *file_offset){
        printk("Write function called\n");

        if(len > KERNEL_BUFFER_SIZE){
                len = KERNEL_BUFFER_SIZE;
        }

        int to_write = len;
        int not_written = copy_from_user(kernel_buffer,buf,len);
        int data_written = to_write - not_written;
        if(not_written){
                printk(KERN_ERR "Failed to copy data from user space\n");
                return -EFAULT;
        }
Terminal
        buffer_pointer = data_written;

        printk(KERN_INFO "Data Write Success\n");
        return data_written;
}

static int chr_driver_init(void){
        if(LINUX_VERSION_CODE != KERNEL_VERSION(
                kernel_version[0],kernel_version[1],kernel_version[2]
                )
                        ){
                printk(KERN_ERR "kernel version does not match\n");
                printk(KERN_INFO "linux version code: %d\n",LINUX_VERSION_CODE);
                printk(KERN_INFO "kernel version code: %d\n",
                                KERNEL_VERSION(
                kernel_version[0],kernel_version[1],kernel_version[2])
                                );
                return -1;
        }

        if((kernel_buffer = kmalloc(KERNEL_BUFFER_SIZE,GFP_KERNEL)) == 0){
                        printk(KERN_INFO "Cannot allocate memory\n");
                        return -1;
```

```
        }


        if(alloc_chrdev_region(&dev,0,1,"character_device") < 0){
                printk(KERN_INFO "Cannot allocate major number\n");
               return -1;
        }

        printk(KERN_INFO "Major: %d, Minor: %d",MAJOR(dev),MINOR(dev));
        cdev_init(&chr_cdev,&fops);

        if(cdev_add(&chr_cdev,dev,1) < 0){
                printk(KERN_INFO "Cannot add the device to the system\n");
                goto destroy_class;
        }
Terminal
        if((dev_class = class_create("character_device_class")) == 0){
                printk(KERN_INFO "Cannot create class\n");
                goto destroy_class;
        }

        if(device_create(dev_class,NULL,dev,NULL,"character_device_1") == NULL){
                printk(KERN_INFO "Cannot create device\n");
                goto destroy_device;
        }

        printk(KERN_INFO "Device Driver Insert Success\n");

        return 0;

        destroy_device:
                class_destroy(dev_class);

        destroy_class:
                unregister_chrdev_region(dev,1);
                cdev_del(&chr_cdev);

        return -1;
}
```

```
static void __exit chr_driver_exit(void){
        kfree(kernel_buffer);
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&chr_cdev);
        unregister_chrdev_region(dev,1);
        printk(KERN_INFO "Device Driver Remove Success\n");
}


module_init(chr_driver_init);
module_exit(chr_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ervin Ranjan");
MODULE_DESCRIPTION("Linux Character Device Driver");
```

FIrst we need to run the command ‘make’ to compile the module.

```
obj-m := chr_driver.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
1 change; before #1  49 seconds ago                          4,2-9         All
```

Figure 1 : Contents of Makefile

Make: A build automation tool that manages the compilation and linking of programs by reading instructions from a Makefile.
Makefile: A file containing rules and dependencies that define how to build and compile a project using make.

```
ervin@ervin-VirtualBox:~$ make
make -C /lib/modules/6.5.0-35-generic/build M=/home/ervin modules
make[1]: Entering directory '/usr/src/linux-headers-6.5.0-35-generic'
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0
  You are using:           gcc-12 (Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0
  CC [M]  /home/ervin/chr_driver.o
  MODPOST /home/ervin/Module.symvers
  CC [M]  /home/ervin/chr_driver.mod.o
  LD [M]  /home/ervin/chr_driver.ko
  BTF [M] /home/ervin/chr_driver.ko
Skipping BTF generation for /home/ervin/chr_driver.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-6.5.0-35-generic'
```

Figure 2 : Output after running 'make'

Next, we need to insert the module with the command 'insmod' with the current kernel version , first I will show what happens if we enter the wrong kernel version,

```
ervin@ervin-VirtualBox:~$ sudo insmod chr_driver.ko kernel_version=6,5,0
insmod: ERROR: could not insert module chr_driver.ko: Operation not permitted
ervin@ervin-VirtualBox:~$ sudo dmesg | tail -n 3
[36422.803720] kernel version does not match
[36422.803772] linux version code: 394509
[36422.803774] kernel version code: 394496
```

Figure 3 : Output upon entering wrong kernel version

I printed out the kernel message buffer using the dmesg command . We can see that the encodings of the two don't match.

Now we show the output upon correct input,

```
ervin@ervin-VirtualBox:~$ sudo insmod chr_driver.ko kernel_version=6,5,13
ervin@ervin-VirtualBox:~$ sudo dmesg | tail -n 2
[36707.057716] Major: 239, Minor: 0
[36707.062044] Device Driver Insert Success
```

Figure 4 : Output upon entering current kernel version

We now know that the major number is 239, Now we can see the registered device number with name 'character_device' in '/proc/devices' , Note that this need not be same as the device name (I created the device with name 'character_device_1').

```
ervin@ervin-VirtualBox:~$ cat /proc/devices | grep 239
239 character_device
```

Figure 5 : Displaying device registered with major number 239

Now we can also see the device with name 'character_device_1' in '/dev',

```
ervin@ervin-VirtualBox:~$ ls -l /dev/ | grep character_device_1
crw-------  1 root  root    239,   0 Oct 28 23:49 character_device_1
```

Figure 6 : displaying device in /dev

We need to change the ownership from root to ervin, to read and write data to and fro the device,so I used the chown command

```
ervin@ervin-VirtualBox:~$ sudo chown ervin /dev/character_device_1
ervin@ervin-VirtualBox:~$ ls -l /dev | grep character_device_1
crw-------  1 ervin root    239,   0 Oct 28 23:49 character_device_1
```

Figure 7 : changing ownership of device

Now we can write some data into it using the 'echo' command and read some data using 'cat' command to see the output,

```
ervin@ervin-VirtualBox:~$ echo "ERVIN_B220027CS" > /dev/character_device_1
ervin@ervin-VirtualBox:~$ cat /dev/character_device_1
ERVIN_B220027CS
ervin@ervin-VirtualBox:~$ sudo dmesg | tail -n 12
[38773.736978] Device File Opened
[38773.736994] Write function called
[38773.736998] Data Write Success
[38773.737078] Close function called
[38773.737153] Device File Closed
[38778.458409] Open function called
[38778.458414] Device File Opened
[38778.458424] Read function called
[38778.458427] Data Read Success
[38778.458503] Read function called
[38778.458522] Close function called
[38778.458523] Device File Closed
```

Figure 8 : writing and reading from device with 'echo' and 'cat' commands respectively

Now we can do the same with a c program (called read_write.c)

Source code (read_write.c) :

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
        FILE* fptr;
        char buf[16];

        fptr = fopen("/dev/character_device_1","r+");
        fprintf(fptr,"ERVIN_B220027CS");
        fscanf(fptr,"%s",buf);
        printf("%s\n",buf);
        fclose(fptr);

        return 0;
}
```

We create the executable using gcc and then execute it , the output is shown below,

```
ervin@ervin-VirtualBox:~$ gcc -o read_write read_write.c
ervin@ervin-VirtualBox:~$ ./read_write
ERVIN_B220027CS
ervin@ervin-VirtualBox:~$ sudo dmesg | tail -n 9
[38911.488675] Open function called
[38911.488680] Device File Opened
[38911.488696] Write function called
[38911.488697] Data Write Success
[38911.488699] Read function called
[38911.488700] Data Read Success
[38911.488705] Read function called
[38911.490177] Close function called
[38911.490179] Device File Closed
```

Figure 9 : executing user program to test the driver code

With this we have written and tested our device driver.