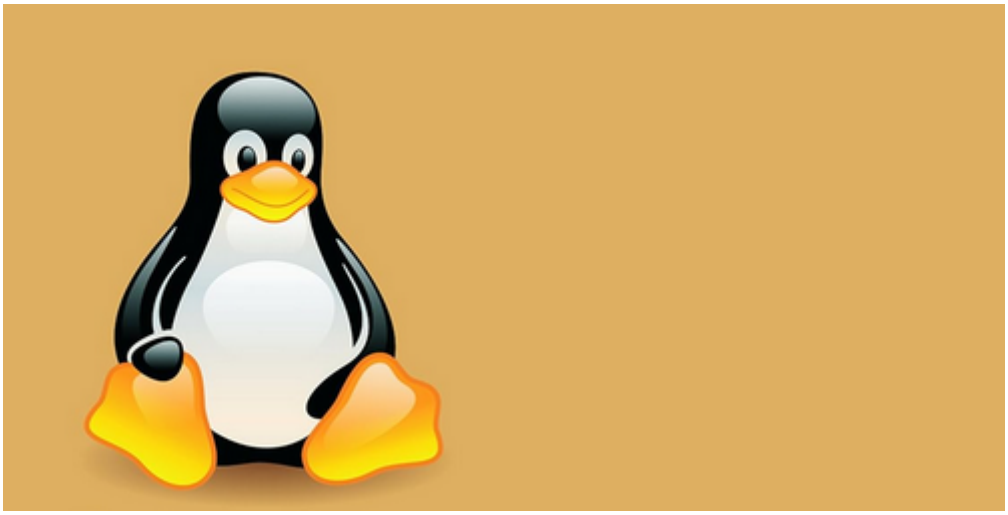
CS3003D: Operating Systems

VIVEK K P

B210473CS



1. Problem Statement & Methodology

Objective

Create a Linux kernel module that accepts two parameters during insertion using the `insmod` command.

Module Parameters

- `kernel_version`: An array parameter specifying the current kernel version.
- `time` : The time in seconds before which the tasks described below should be performed.

Module Insertion Rules

- The module insertion is successful only if the specified `kernel_version` matches the version during module compilation (using LINUX_VERSION_CODE).
- Upon successful insertion, print the major number, minor number, and timer value in kernel messages using `dmesg`.

Post Insertion Actions

After successful insertion, the following two actions should be performed in the given order before the specified time elapses:

1. **Read Operation**: Allow userspace to read from the character device using commands such as `cat` or `read`.
2. **Write Operation**: Get the username of the currently logged-in user and write it to the device using commands such as `echo` or `write`.

References

- Utilize `LINUX_VERSION_CODE` for kernel version comparison.
- Implement a kernel waitqueue mechanism for handling actions within the specified time.

Module Removal

When `rmmod` is called:

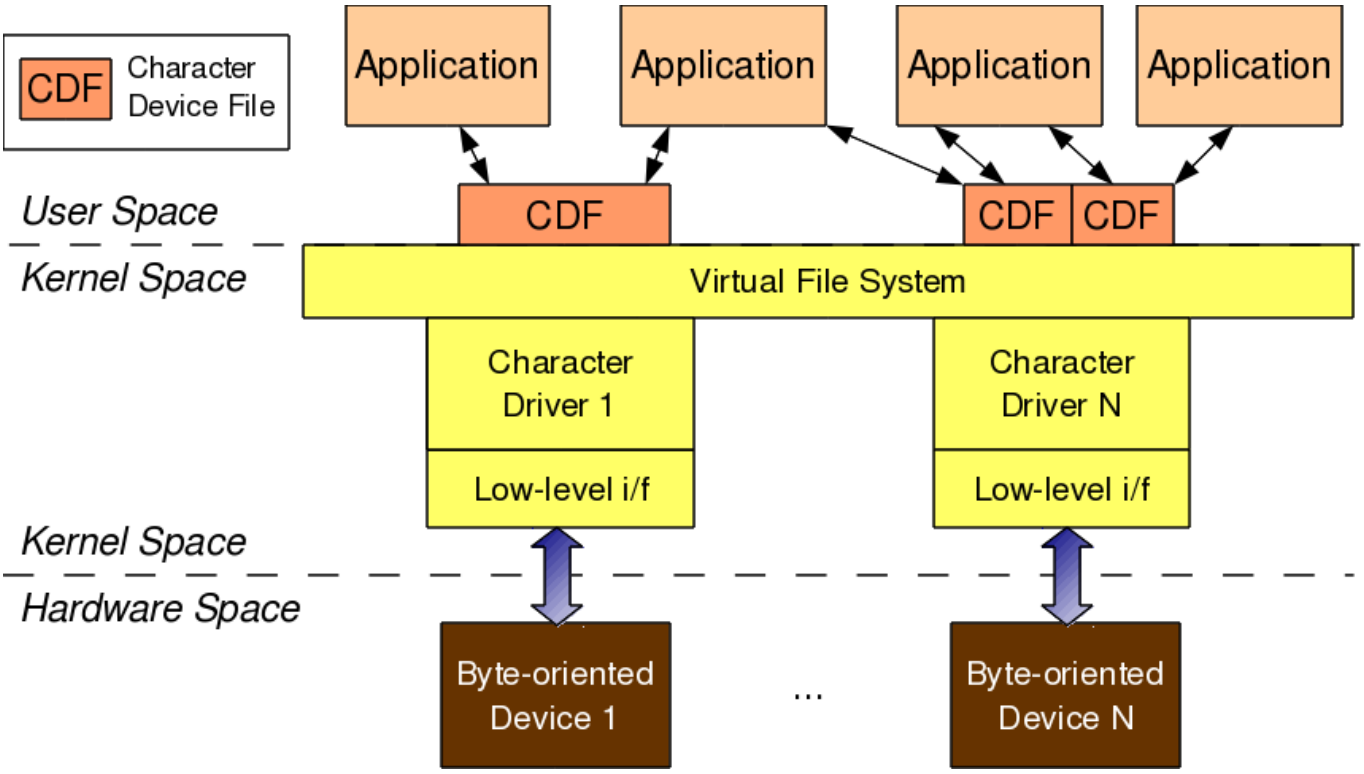
- Check whether the actions were completed in order within the given time.
- If successful, print "Successfully completed the actions within time" along with the username.
- If unsuccessful, print a failure message.

Note: The completion of actions within the specified time is crucial for successful removal.

2.Explanation

Linux Character device

A character device is one of the simplest ways to communicate with a module in the Linux kernel.
These devices are presented as special files in a */dev* directory and support direct reading and writing of any data, byte by byte, like a stream. Actually, most of the pseudo-devices in */dev* are character devices: serial ports, modems, sound, and video adapters, keyboards, some custom I/O interfaces. Userspace programs can easily open, read, write, and custom control requests with such device files.
Here I am describing how to write a simple Linux kernel module which can create one or multiple character device.



1.Passing arguments to the Linux device driver

module parameters macros
This macro is used to initialize the arguments. `module_param` takes three parameters: the name of the variable, its type, and a permissions mask to be used for an accompanying sysfs entry. The macro should be placed outside of any function and is typically found near the head of the source file. `module_param()` macro, defined in `linux/moduleparam.h`.

`module_param(name, type, perm);`

```
module_param(valueETX, int, S_IWUSR|S_IRUSR);
```

here we can use the LINUX_VERSION_CODE macros

below is the code snippet explaining how i done it,

```
printk(KERN_INFO "The kernel verion is correct");

printk(KERN_INFO "timer = %d \n", timer);

// printk(KERN_INFO "cb_valueETX = %d \n", cb_valueETX);

// printk(KERN_INFO "Kernel-Version = %s \n", kernel_version);

pr_info("Kernel version: %d.%d.%d\n",
```

```
LINUX_VERSION_CODE / 65536,

(LINUX_VERSION_CODE / 256) % 256,

LINUX_VERSION_CODE % 256);
```

along with the header file `#include <linux/version.h>`

2.Minor and Major Numbers , Device file creation

In Linux device drivers, the concepts of major and minor numbers are essential for identifying and managing different devices. Here's an explanation of these concepts and how they relate to creating a device file in a Linux device driver:

Major Number:

- **Definition:** The major number identifies the specific driver associated with a device.
- **Usage:** The kernel uses the major number to route I/O requests to the appropriate device driver.
- **Dynamic Assignment:** Traditionally, major numbers were statically assigned. However, modern systems often use dynamic major number allocation.
- **Device File Creation:** When a device driver is registered, it is assigned a major number. The driver communicates this major number to the kernel, which then uses it to create the device file in the `/dev` directory.

Minor Number:

- **Definition:** The minor number differentiates between different devices handled by the same driver.
- **Usage:** The combination of major and minor numbers uniquely identifies a specific device instance.
- **Device File Naming:** In the `/dev` directory, device files are named using a convention that includes the major and minor numbers. For example, `/dev/mydevice0` might correspond to the first instance of a device handled by the driver.
- **Multiple Devices, One Driver:** A single driver can handle multiple devices, each distinguished by a unique combination of major and minor numbers.

Creating a Device File:

1. **Driver Registration:**

- When a device driver is loaded, it typically registers itself with the kernel, specifying its capabilities and providing a set of file operations.

```
// Register the character device with the kernel
register_chrdev(major_number, "mydevice", &file_operations);
```

In this example, `major_number` is the dynamically assigned major number, and `file_operations` is a structure containing function pointers for handling file operations.

2. **Major Number Allocation:**

   - If using dynamic major number allocation, the kernel assigns a major number to the driver during registration.

```
major_number = register_chrdev(0, "mydevice", &file_operations);
```

The `register_chrdev` function returns the assigned major number.

3. **Device File Creation:**

   - The major number is used to create the device file in the `/dev` directory. The minor number is typically used to distinguish between multiple devices managed by the same driver.

```
// Create device file in /dev
device_create(my_class, NULL, MKDEV(major_number, 0), NULL, "mydevice0");
```

Here, `my_class` is the device class, `MKDEV` is a macro that combines the major and minor numbers, and `"mydevice0"` is the name of the device file.

4. **User Access:**
   - Users can interact with the device using the device file created in `/dev`. For example, reading and writing to `/dev/mydevice0` allows users to communicate with the corresponding device handled by the driver.
5. **Driver Cleanup:**

   - When unloading the driver, resources must be released, and the device file removed.

```
// Remove the device file
device_destroy(my_class, MKDEV(major_number, 0));

// Unregister the character device
unregister_chrdev(major_number, "mydevice");
```

This removes the device file and unregisters the character device.

Understanding major and minor numbers is crucial for managing devices in a Linux kernel driver, and creating a device file facilitates user-space interactions with the driver's functionality.

3.File Operations

File Operations in `struct file_operations`

####### `struct module *owner`:

- **Definition:** This field is a pointer to the module that "owns" the structure, preventing the module from being unloaded while its operations are in use.
- **Initialization:** Typically initialized to `THIS_MODULE`, a macro defined in `<linux/module.h>`.

`ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)`:

- **Purpose:** Used to retrieve data from the device.
- **Return Value:** A non-negative value represents the number of bytes successfully read.
- **Null Pointer:** If NULL, it causes the read system call to fail with -EINVAL.

`ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)`:

- **Purpose:** Sends data to the device.
- **Return Value:** A non-negative value represents the number of bytes successfully written.
- **Null Pointer:** If NULL, it causes the write system call to fail with -EINVAL.

`int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long)`:

- **Purpose:** Allows issuing device-specific commands (e.g., formatting a track of a floppy disk).
- **Return Value:** Returns an error for any request that isn't predefined (-ENOTTY, "No such ioctl for device").
- **NULL:** If NULL, ioctl system calls return -ENOTTY.

`int (*open) (struct inode *, struct file *)`:

- **Purpose:** Invoked when a file structure is being opened.
- **Return Value:** If NULL, opening the device always succeeds, but the driver isn't notified.

`int (*release) (struct inode *, struct file *)`:

- **Purpose:** Invoked when the file structure is being released (closed).
- **Return Value:** If NULL, the release operation is not performed.

Example Usage:

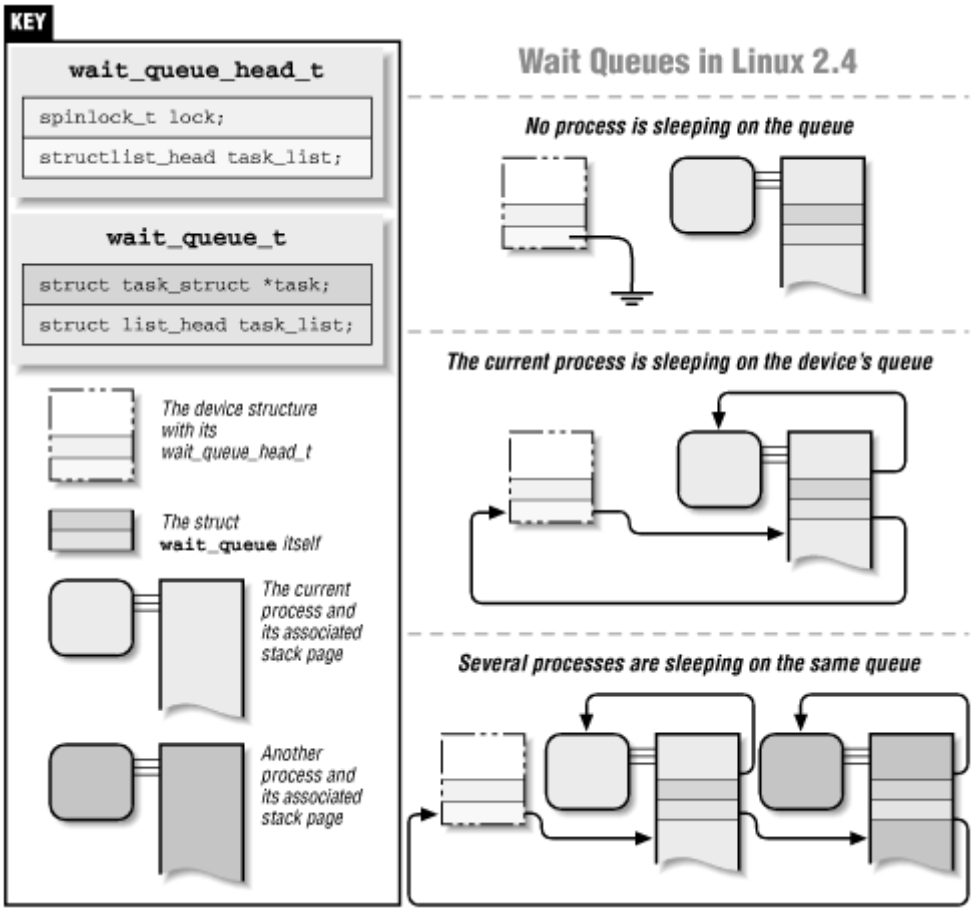
```
static struct file_operations fops = {
    .owner    = THIS_MODULE,
    .read     = etx_read,
    .write    = etx_write,
    .open     = etx_open,
    .release  = etx_release,
};
```

In this example, `etx_read`, `etx_write`, `etx_open`, and `etx_release` are functions in the driver that implement the read, write, open, and release operations, respectively. The `fops` structure is then associated with the driver to define how the kernel interacts with the driver's file operations.

` Instead of doing `echo` and `cat` command in the terminal you can also use `open()`, `read(), write(), close()` system calls from user-space applications.`

4.Wait queue Timeout

A "wait queue" in the Linux kernel is a data structure to manage threads that are waiting for some condition to become true; they are the normal means by which threads block (or "sleep") in kernel space. Over the years, the wait queue mechanism has evolved into a fairly elaborate and complicated kernel subsystem. Now, however, there is a move afoot to simplify that code, using a wait queue variant developed for the realtime tree; the result could be a fair amount of code churn in the kernel.



sleep until a condition gets true or a timeout elapses

`wait_event_timeout(wq, condition, timeout);`

`wq` – the waitqueue to wait on

`condtion` – a C expression for the event to wait for

`timeout` – timeout, in jiffies

The process is put to sleep (`TASK_UNINTERRUPTIBLE`) until the *`condition`* evaluates to true or timeout elapses. The *`condition`* is checked each time the waitqueue *`wq`* is woken up.

It **returns 0** if the *`condition`* evaluated to `false` after the *`timeout`* elapsed, **1** if the *`condition`* evaluated to `true` after the *`timeout`* elapsed, or the **remaining jiffies** (at least 1) if the *condition* evaluated to `true` before the *`timeout`* elapsed.

4.Screenshots and Code snippets

driver.c

```
#include <linux/kernel.h>
```

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/err.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/slab.h>
#include <linux/version.h>
dev_t dev = 0;

static struct class * dev_class;

static struct cdev etx_cdev;
int timer, kernel_version[4];
module_param(timer, int, S_IRUSR | S_IWUSR); //integer value
module_param_array(kernel_version, int, NULL, S_IRUSR | S_IWUSR); //Array of
integers
static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);
static int etx_open(struct inode * inode, struct file * file);
static int etx_release(struct inode * inode, struct file * file);
static ssize_t etx_read(struct file * filp, char __user * buf, size_t len,
loff_t * off);
static ssize_t etx_write(struct file * filp,
const char * buf, size_t len, loff_t * off);
static struct file_operations fops = {
.owner = THIS_MODULE,
.read = etx_read,
.write = etx_write,
.open = etx_open,
.release = etx_release,
};
static int etx_open(struct inode * inode, struct file * file) {
pr_info("Driver Open Function Called...!!!\n");
return 0;
}
static int etx_release(struct inode * inode, struct file * file) {
pr_info("Driver Release Function Called...!!!\n");
return 0;
}
static ssize_t etx_read(struct file * filp, char __user * buf, size_t len,
loff_t * off) {
pr_info("Driver Read Function Called...!!!\n");
return 0;
}
```

```
char * kernel_buf;
char * temp;
static ssize_t etx_write(struct file * filp,
const char __user * buf, size_t len, loff_t * off) {
pr_info("Driver Write Function Called...!!!\n");
kernel_buf = kmalloc(len, GFP_KERNEL);
if (!kernel_buf) {
pr_err("Memory allocation failed\n");
return -ENOMEM;
}
temp = kmalloc(len, GFP_KERNEL);
copy_from_user(temp, buf, len);
if (copy_from_user(kernel_buf, buf, len)) {
pr_err("Failed to copy data from user space\n");
kfree(kernel_buf);
return -EFAULT;
}
pr_info("Username is: %s\n", kernel_buf);
kfree(kernel_buf);
return len;
}
int timer_flag = 0;
static struct timer_list my_timer;
void timer_callback(struct timer_list * t) {
timer_flag = 1;
printk("timer expired");
}
void start_timer(void) {
timer_setup( & my_timer, timer_callback, 0);
printk("The timer is started....\n");
mod_timer( & my_timer, (jiffies + msecs_to_jiffies(timer * 1000)));
}
static int __init etx_driver_init(void) {
int i;
int flag = 0;
for (i = 0; i < (sizeof kernel_version / sizeof(int)); i++) {
if (i == 0 && kernel_version[i] == (LINUX_VERSION_CODE / 65536))
flag++;
if (i == 1 && kernel_version[i] == ((LINUX_VERSION_CODE / 256) % 256))
flag++;
if (i == 2 && kernel_version[i] == (LINUX_VERSION_CODE % 256))
flag++;
}
if (flag != 3) {
printk(KERN_INFO "The given kernel_version is wrong!!!!");
printk(KERN_INFO "Kernel Module insertion is unsuccesfull!!!!!!");
```

```
}

if (flag == 3) {

start_timer();

printk(KERN_INFO "The kernel verion is correct");

printk(KERN_INFO "timer = %d \n", timer);

pr_info("Kernel version: %d.%d.%d\n",

LINUX_VERSION_CODE / 65536,

(LINUX_VERSION_CODE / 256) % 256,

LINUX_VERSION_CODE % 256);



if ((alloc_chrdev_region( & dev, 0, 1, "etx_Dev")) < 0) {

pr_err("Cannot allocate major number\n");

return -1;

}

pr_info("Major = %d Minor = %d \n", MAJOR(dev), MINOR(dev));

cdev_init( & etx_cdev, & fops);

if ((cdev_add( & etx_cdev, dev, 1)) < 0) {

pr_err("Cannot add the device to the system\n");

goto r_class;

}
if (IS_ERR(dev_class = class_create(THIS_MODULE, "etx_class"))) {

pr_err("Cannot create the struct class\n");

goto r_class;

}
```

```
if (IS_ERR(device_create(dev_class, NULL, dev, NULL, "etx_vivek"))) {
pr_err("Cannot create the Device 1\n");
goto r_device;
}
pr_info("Device Driver Insert...Done!!!\n");
return 0;
}
r_device:
class_destroy(dev_class);
r_class:
unregister_chrdev_region(dev, 1);
return -1;
}

static void __exit etx_driver_exit(void) {

device_destroy(dev_class, dev);

class_destroy(dev_class);

cdev_del( & etx_cdev);

unregister_chrdev_region(dev, 1);

if (timer_flag == 1) {

pr_info("Not completed, timer expired...!!!\n");

} else {

pr_info("Device Driver Remove...Done!!!\n");

pr_info("Successfully completed the actions within time, username=%s\n",
temp);

kfree(temp);

}

}

module_init(etx_driver_init);

module_exit(etx_driver_exit);
```

```
MODULE_LICENSE("GPL");

MODULE_AUTHOR("vivek <vivek_b210473cs@nitc.ac.in>");

MODULE_DESCRIPTION("Simple Linux device driver (File Operations)");

MODULE_VERSION("1.3");
```

1.Create the Makefile

```
obj-m += driver.o

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

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

- run make

```
make
```

```
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ make
make -C /lib/modules/5.15.0-83-generic/build M=/home/vivek/programming/kernel_programming/pro_4 modules
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-83-generic'
  CC [M]  /home/vivek/programming/kernel_programming/pro_4/driver_1.o
  MODPOST /home/vivek/programming/kernel_programming/pro_4/Module.symvers
  LD [M]  /home/vivek/programming/kernel_programming/pro_4/driver_1.ko
  BTF [M] /home/vivek/programming/kernel_programming/pro_4/driver_1.ko
Skipping BTF generation for /home/vivek/programming/kernel_programming/pro_4/driver_1.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-83-generic'
```

- loading device driver

```
sudo insmod driver.ko timer=30 kernel_version=5,15,116
```

```
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ vim Makefile
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ vim Makefile
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ sudo insmod driver_1.ko timer=30 kernel_version=5,15,116
```

- give the permission to /dev/etx_vivek

```
chmod 777 /dev/etx_vivek
```

```
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ chmod 777 /dev/etx_vivek
```

- write into user space by using echo

```
echo $(whoami) > /dev/etx_vivek
```

```
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ echo $(whoami) > /dev/etx_vivek
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$
```

- use dmesg to see the the log things and printk outputs

```
dmesg | tail -n 12
```

```
[41580.769959] The kernel verion is correct
[41580.769961] timer = 30
[41580.769965] Kernel version: 5.15.116
[41580.769968] Major = 508 Minor = 0
[41580.770170] Device Driver Insert...Done!!!
```

```
[48188.069358] Driver Open Function Called...!!!
[48188.069380] Driver Write Function Called...!!!
[48188.069383] The value is: vivek

[48188.069387] Driver Release Function Called...!!!
```

- read from kernel space by using cat

```
cat /dev/etx_vivek
```

```
[49000.633207] Driver Open Function Called...!!!
[49000.633232] Driver Read Function Called...!!!
[49000.633243] Driver Release Function Called...!!!
```

- currently active charactor block devices

```
cat /proc/devices
```

```
vivek@vivek-HP-Pavilion:/dev$ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  5 ttyprintk
  6 lp
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
 81 video4linux
 89 i2c
 99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
202 cpu/msr
204 ttyMAX
216 rfcomm
226 drm
234 ttyDBC
```

```
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ sudo insmod driver_1.ko timer=10 kernel_version=5,15,116
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ chmod 777 /dev/etx_vivek
chmod: changing permissions of '/dev/etx_vivek': Operation not permitted
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ sudo !!!
sudo chmod 777 /dev/etx_vivek !
chmod: cannot access '!': No such file or directory
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ sudo chmod 777 /dev/etx_vivek
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ echo $(whoami) > /dev/etx_vivek
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ cat /dev/etx_vivek
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ dmesg | tail -n 10
[ 5911.935539] Device Driver Insert...Done!!!
[ 5922.178641] timer expired
[ 5965.141929] Driver Open Function Called...!!!
[ 5965.141970] Driver Write Function Called...!!!
[ 5965.141973] Username is: vivek

[ 5965.141979] Driver Release Function Called...!!!
[ 5971.782514] Driver Open Function Called...!!!
[ 5971.782529] Driver Read Function Called...!!!
[ 5971.782539] Driver Release Function Called...!!!
```

unsuccessful completion of task

```
make
sudo insmod driver_1.ko timer=100 kernel_version=5,15,116
sudo chmod /dev/etx_vivek
echo $(whoami) > /dev/etx_vivek
cat /dev/etx_vivek
sudo rmmod driver_1.ko
dmesg | tail -n 15
```

```
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ sudo rmmod driver_1.ko
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ dmesg | tail -n 15
[ 5911.935304] timer = 10
[ 5911.935306] Kernel version: 5.15.116
[ 5911.935310] Major = 508 Minor = 0
[ 5911.935539] Device Driver Insert...Done!!!
[ 5922.178641] timer expired
[ 5965.141929] Driver Open Function Called...!!!
[ 5965.141970] Driver Write Function Called...!!!
[ 5965.141973] Username is: vivek

[ 5965.141979] Driver Release Function Called...!!!
[ 5971.782514] Driver Open Function Called...!!!
[ 5971.782529] Driver Read Function Called...!!!
[ 5971.782539] Driver Release Function Called...!!!
[ 6062.680183] nouveau 0000:01:00.0: sec2: unhandled intr 00000010
[ 6089.480938] Not completed, timer expired...!!!
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$
```

- Successful completion of task


```
make
sudo insmod driver_1.ko timer=100 kernel_version=5,15,116
sudo chmod /dev/etx_vivek
echo $(whoami) > /dev/etx_vivek
cat /dev/etx_vivek
sudo rmmod driver_1.ko
dmesg | tail -n 15
```

```
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ sudo insmod driver_1.ko timer=100 kernel_version=5,15,116
[sudo] password for vivek:
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ chmod 777 /dev/etx_vivek
chmod: changing permissions of '/dev/etx_vivek': Operation not permitted
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ sudo !!
sudo chmod 777 /dev/etx_vivek
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ echo $(whoami) > /dev/etx_vivek
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ cat /dev/etx_vivek
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ sudo rmmod driver_1.ko
vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$ dmesg | tail -n 15
[  168.114779] timer = 100
[  168.114780] Kernel version: 5.15.116
[  168.114781] Major = 508 Minor = 0
[  168.115012] Device Driver Insert...Done!!!
[  192.204144] Driver Open Function Called...!!!
[  192.204198] Driver Write Function Called...!!!
[  192.204202] Username is: vivek

[  192.204214] Driver Release Function Called...!!!
[  198.300605] Driver Open Function Called...!!!
[  198.300620] Driver Read Function Called...!!!
[  198.300628] Driver Release Function Called...!!!
[  209.208969] Device Driver Remove...Done!!!
[  209.208970] Successfully completed the actions within time, username=vivek

vivek@vivek-HP-Pavilion:~/programming/kernel_programming/pro_4$
```