CSC3150 Assignment 5

Student ID: 119010249 Name: Qin Peiran

Part 1: Designing steps of my program

- Overall Description

In this project, I implement a device as well as its operations in the kernel module to control this device. This device can do the arithmetic operations like add, subtraction, division, multiplication, and finding prime numbers. What's more, to operate on this device and set the configurations of this device, several functions are also implemented including read, write(blocking, non-blocking), and some ioctl functions. In this way, users can directly use the APIs and controls/operates on my device. In the bonus part, An interrupt service routine was added to the IRQ number's action list and counts for the keyboard interrupt time. In the following parts, I will introduce my design ideas in detail.

- Design Structure

There are mainly five parts of my program: 1. Register, initialize and add the device. 2. Implementation of ioctl functions. 3. Implementation of file operations. 4. Implementation of arithmetic routine. 5. Removement of the device.

1. **Register, initialize and add the device**

- In the first step of initialization, we register range of device number and get the major and minor number of the device. (Major number would be used to identify the driver and minor number would be used to specify the device).
- Then, we allocate an cdev structure and initialize it with the file operations. The file operations are included in the following structure: (there is a mapping between user mode and kernel mode)

```
static struct file_operations fops = {
      owner: THIS_MODULE,
      read: drv_read,
      write: drv_write,
      unlocked_ioctl: drv_ioctl,
      open: drv_open,
      release: drv_release,
};
```

  After that, the cdev is ready to be added. Then `cdev_add()` is called to add the device to the system and make it live immediately.

- Except for that, we also initialize the DMA buffer, and allocate a work routine using `dma_buf = kzalloc(DMA_BUFSIZE, GFP_KERNEL);`, `work_routine = kmalloc(sizeof(typeof(*work_routine)), GFP_KERNEL);`.
- After that, we would compile the module and insert the kernel object using `sudo insmod mydev.ko` as well as create a system node using `mknod`. Also, we use `chmod` to change the access permissions of the file system object to be 666, meaning all the users can read/write it.

2. **Implementation of ioctl functions**

- After the cdev is successfully initialized, the next step is to implement the ioctl functions so that users can modify the configurations of the device. The masked labels include:
    - HW5_IOCSETSTUID: in this setting, the user (I) can write my student ID to the specific address in the DMA buffer. (the value user pass in would be the student id).

```
int value;
get_user(value, (int *) arg);
myouti(value, DMASTUIDADDR);
if (value >= 0) printk("%s:%s(): My STUID is = %d\n", PREFIX_TITLE,
__func__, value);
else return -1;
```

    - HW5_IOCSETRWOK, HW5_IOCSETIOCOK, HW5_IOCSETIRQOK: I write 1 to those corresponding addresses of DMA buffer since I complete the RW function, ioctl function and bonus part.

```
if (cmd == HW5_IOCSETRWOK){
    myouti(value, DMARWOKADDR);
    if (value == 1) printk("%s:%s(): RM OK\n", PREFIX_TITLE, __func__);
    else return -1;
}
if (cmd == HW5_IOCSETIOCOK){
    myouti(value, DMAIOCOKADDR);
    if (value == 1) printk("%s:%s(): IOC OK\n", PREFIX_TITLE, __func__);
    else return -1;
}
if (cmd == HW5_IOCSETIRQOK){
    myouti(value, DMAIRQOKADDR);
    if (value == 1) printk("%s:%s(): IRQ OK\n", PREFIX_TITLE, __func__);
    else return -1;
}
```

    - HW5_IOCSETBLOCK: with this label mask, the user can set the next write operation to be blocking or non-blocking. If user wants it to be blocking write, 1 will be written to the corresponding space of the DMA buffer.

```
if (cmd == HW5_IOCSETBLOCK){
    myouti(value, DMABLOCKADDR);
    if (value == 1) printk("%s:%s(): Blocking IO\n", PREFIX_TITLE,
__func__);
    else if (value == 0) printk("%s:%s(): Non-Blocking IO\n",
PREFIX_TITLE, __func__);
}
```

    - HW5_IOCWAITREADABLE: To ensure the correctness of the read operation after the non-blocking write, the user can use the ioctl function with this masked label to wait for the value in the readable-address being set. In the implementation, it will use while loop and `msleep` to wait until it is readable.

```
if (cmd == HW5_IOCWAITREADABLE){
    int readable = myini(DMAREADABLEADDR);     /* check whether it is
readable */
    printk("%s:%s(): wait readable 1\n", PREFIX_TITLE, __func__);
    while (readable != 1){
        msleep(1);
        readable = myini(DMAREADABLEADDR);
    }
    put_user(readable, (int *)arg);
    if(readable == 1) return 1;
    else return 0;
}
```

3. **Implementation of file operations (read, write)**

- First of all, the parameters (IO mode, operator, operands) passed by user would be first fetched using `get_user()`. IO mode would be used to distinguish the write mode, operators and operands would be stored into the DMA buffer which can be used in the arithmetic routine to perform the operation. Moreover, the arithmetic routine would be defined to the work routine which would be scheduled and executed later.
- **Blocking write**
  - In the beginning, the work would be scheduled using `schedule_work()`, which means that the work task would be put into the global workqueue, and kernel thread would be assigned to perform the work.
  - According to the definition of blocking write, the write operation would not return until the operation is done, which means that we should use `flush_scheduled_work()` to ensure that the scheduled work has run to completion. Then return.
- **Non-Blocking write**
  - Instead of waiting for the scheduled work to be completed, non-blocking work would return immediately once the task is put into the global workqueue.
- **Read**
  - My implementation of the read operation is blocking, which means that it will keep checking until readable is set. Then it would retrieve the result in the DMA buffer, transmit it to user using `put_user` then clear the readable.

4. **Implementation of arithmetic routine**

- The arithmetic functions we implement here are +, -, *, /, prime. The basic steps are that, it first retrieves the necessary data from the DMA buffer. Then calculate the result according to the corresponding operator and operands. After getting the result, it would put the result into the DMA buffer and then set the readable to be 1.

5. **Exit the module**

- First, we use `MKDEV` to create a value that can be compared to a kernel device number. Then we use `cdev_del()` to remove the device from the system. Finally, use `unregister_chrdev_region()` to unregister the range of device number:

```
dev_t dev;
dev = MKDEV(dev_major, dev_minor);
cdev_del(dev_cdevp);
free_irq(IRQ_NUM, &dev_major);
unregister_chrdev_region(dev, 1);
```

By the way, we are also here to remove free the memory of DMA buffer and free the work routine.

- Design steps of bonus part

- In the bonus part, we want to count the interrupts number of keyboard event, which means we can add an interrupt service routine to the action list of IRQ number of keyboard. When a keyboard event happens, it would perform the function of counting would be performed.
- In the initialization of the module, we use `request_irq()` to register an interrupt handler and enbles a given interrupts line for handeling.

```
request_irq(IRQ_NUM, (irq_handler_t) irq_handler, IRQF_SHARED,
"Iterrupt_Counter_Device", &dev_major);
```

The `IRQ_NUM` is 1, which is the IRQ number of keyboard. The `irq_handler` is the routine that will be executed when the interrupts happen, its function in this project is to add 1 to the counter. `IRQF_SHARED` is set means that many devices share the same interrupt line (so that the action that counter plus one would be executed when keyboard interrupt happens). The last argument is the cookie passed back to the handler function (when free the IRQ, it would be used to identify which one should be freed).

- Then, when exiting the module, it would be removed by calling:

```
free_irq(IRQ_NUM, &dev_major);
```

Part 2: The environment of running my program

- My project adopts the recommended environment:
  - version of OS:

```
qpr@ubuntu:~/Desktop$ cat /etc/issue
Ubuntu 16.04.5 LTS \n \l
```

  - version of kernel:

```
qpr@ubuntu:~/Desktop$ uname -r
4.10.14
```

  - gcc version:

```
root:/ $ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Part 3: The steps to execute my program

The steps are:

```
$ cd source
$ make
$ dmesg
$ sudo ./mkdev.sh MAJOR MINOR
$ ./test
$ make clean
$ sudo ./rmdev.sh
```

Remark: for `$ sudo ./mkdev.sh MAJOR MINOR`, MAJOR and MINOR should be the ones print out in `$dmesg`.

Part 4: The screenshot of my program output

- For the user Usermode output it is:

```
qpr@ubuntu:~/Desktop/3150-p5/source$ ./test
..............Start..............
100 + 10 = 110

Blocking IO
ans=110 ret=110

Non-Blocking IO
Queueing work
Waiting
Can read now.
ans=110 ret=110

100 - 10 = 90

Blocking IO
ans=90 ret=90

Non-Blocking IO
Queueing work
Waiting
Can read now.
ans=90 ret=90

100 * 10 = 1000

Blocking IO
ans=1000 ret=1000

Non-Blocking IO
Queueing work
Waiting
Can read now.
ans=1000 ret=1000

100 / 10 = 10

Blocking IO
ans=10 ret=10

Non-Blocking IO
Queueing work
Waiting
Can read now.
ans=10 ret=10

100 p 10000 = 105019

Blocking IO
ans=105019 ret=105019

Non-Blocking IO
Queueing work
Waiting
Can read now.
ans=105019 ret=105019

100 p 20000 = 225077

Blocking IO
ans=225077 ret=225077
```

For the kernel mode output:

```
[ 1196.482630] OS_AS5:init_modules():..............Start..............
[ 1196.482631] OS_AS5:init_modules(): register chrdev(245,0)
[ 1196.482636] OS_AS5:init_modules(): request_irq 1 return 0
[ 1196.482636] OS_AS5:init_modules(): allocate dma buffer
[ 1214.043796] OS_AS5:drv_open(): device open
[ 1214.043799] OS_AS5:drv_ioctl(): My STUID is = 119010249
[ 1214.043799] OS_AS5:drv_ioctl(): RM OK
[ 1214.043799] OS_AS5:drv_ioctl(): IOC OK
[ 1214.043800] OS_AS5:drv_ioctl(): IRQ OK IO
[ 1214.043808] OS_AS5:drv_ioctl(): Blocking IO
[ 1214.043809] OS_AS5,drv_write(): queue work
[ 1214.043809] OS_AS5:drv_write(): block
[ 1214.043846] OS_AS5:drv_arithmetic_routine(): 100 + 10 = 110
[ 1214.043865] OS_AS5:drv_read(): ans = 110
[ 1214.043868] OS_AS5:drv_ioctl(): Non-Blocking IO
[ 1214.043869] OS_AS5,drv_write(): queue work
[ 1214.043870] OS_AS5:drv_ioctl(): wait readable 1
[ 1214.043872] OS_AS5:drv_arithmetic_routine(): 100 + 10 = 110
[ 1214.055624] OS_AS5:drv_read(): ans = 110
[ 1214.055628] OS_AS5:drv_ioctl(): Blocking IO
[ 1214.055629] OS_AS5,drv_write(): queue work
[ 1214.055629] OS_AS5:drv_write(): block
[ 1214.055672] OS_AS5:drv_arithmetic_routine(): 100 - 10 = 90
[ 1214.055676] OS_AS5:drv_read(): ans = 90
[ 1214.055679] OS_AS5:drv_ioctl(): Non-Blocking IO
[ 1214.055680] OS_AS5,drv_write(): queue work
[ 1214.055681] OS_AS5:drv_ioctl(): wait readable 1
[ 1214.055682] OS_AS5:drv_arithmetic_routine(): 100 - 10 = 90
[ 1214.068650] OS_AS5:drv_read(): ans = 90
[ 1214.068656] OS_AS5:drv_ioctl(): Blocking IO
[ 1214.068656] OS_AS5,drv_write(): queue work
[ 1214.068656] OS_AS5:drv_write(): block
[ 1214.068683] OS_AS5:drv_arithmetic_routine(): 100 * 10 = 1000
[ 1214.068711] OS_AS5:drv_read(): ans = 1000
[ 1214.068715] OS_AS5:drv_ioctl(): Non-Blocking IO
[ 1214.068716] OS_AS5,drv_write(): queue work
[ 1214.068717] OS_AS5:drv_ioctl(): wait readable 1
[ 1214.068718] OS_AS5:drv_arithmetic_routine(): 100 * 10 = 1000
[ 1214.079837] OS_AS5:drv_read(): ans = 1000
[ 1214.079843] OS_AS5:drv_ioctl(): Blocking IO
[ 1214.079844] OS_AS5,drv_write(): queue work
[ 1214.079844] OS_AS5:drv_write(): block
[ 1214.079890] OS_AS5:drv_arithmetic_routine(): 100 / 10 = 10
[ 1214.079894] OS_AS5:drv_read(): ans = 10
[ 1214.079898] OS_AS5:drv_ioctl(): Non-Blocking IO
[ 1214.079899] OS_AS5,drv_write(): queue work
[ 1214.079900] OS_AS5:drv_ioctl(): wait readable 1
[ 1214.079901] OS_AS5:drv_arithmetic_routine(): 100 / 10 = 10
[ 1214.092063] OS_AS5:drv_read(): ans = 10
[ 1214.891073] OS_AS5:drv_ioctl(): Blocking IO
[ 1214.891075] OS_AS5,drv_write(): queue work
[ 1214.891075] OS_AS5:drv_write(): block
[ 1215.491788] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[ 1215.491946] OS_AS5:drv_read(): ans = 105019
[ 1215.491959] OS_AS5:drv_ioctl(): Non-Blocking IO
[ 1215.491960] OS_AS5,drv_write(): queue work
[ 1215.491962] OS_AS5:drv_ioctl(): wait readable 1
[ 1216.082970] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[ 1216.091944] OS_AS5:drv_read(): ans = 105019
[ 1219.442020] OS_AS5:drv_ioctl(): Blocking IO
[ 1219.442022] OS_AS5,drv_write(): queue work
[ 1219.442023] OS_AS5:drv_write(): block
[ 1221.979319] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
[ 1221.979472] OS_AS5:drv_read(): ans = 225077
[ 1221.979484] OS_AS5:drv_ioctl(): Non-Blocking IO
[ 1221.979485] OS_AS5,drv_write(): queue work
[ 1221.979487] OS_AS5:drv_ioctl(): wait readable 1
[ 1224.504808] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
```

```
[ 1224.508365] OS_AS5:drv_read(): ans = 225077
[ 1224.508464] OS_AS5:drv_release(): device close
[ 1228.200482] OS_AS5:exit_modules(): interrupt count = 92
[ 1228.200483] OS_AS5:exit_modules(): free dma buffer
[ 1228.200484] OS_AS5:exit_modules(): unregister chrdev
[ 1228.200484] OS_AS5:exit_modules():.............End.............
```

Part 5: The things I learned from this Assignment

The gains I get from this project are mainly two: 1. More comprehensive understanding of concepts. 2. Many implementing details. I will explain them briefly.

1. For the gaining of understanding of concepts. I learn the basic structure of registering a device, initializing it and then making it alive as well as how to remove it. Several APIs like `alloc_chrdev_region()`, `cdev_alloc()`, `cdev_init()` I learned from it are very useful if I want to do create some new device and add it to system in the future.

- Except for the knowledge about how to make a device, I become clear about the global view of how the user mode program can operate on the devices. At first, I get a quite vague understanding of the figure copied from the description of Homework 5. During the process of finishing this project, I get a gradually comprehensive understanding of this figure. First, I know that there is a mapping between the functions in user mode and kernel mode, I think this is a vital connection for users' manipulation on the device. Then I start from the implementation of ioctl, which is used to set or get the configurations of the device (quite important for the user to recognize what's the state of the device now especially before read/write). As for the implementation of read, write, it also help me to gain fresh knowledge about the workqueue and how to call to schedule a work in kernel. In addition, I also learn about the usage of DMA buffer, which plays an important role in passing parameters in the kernel space.

Global View:

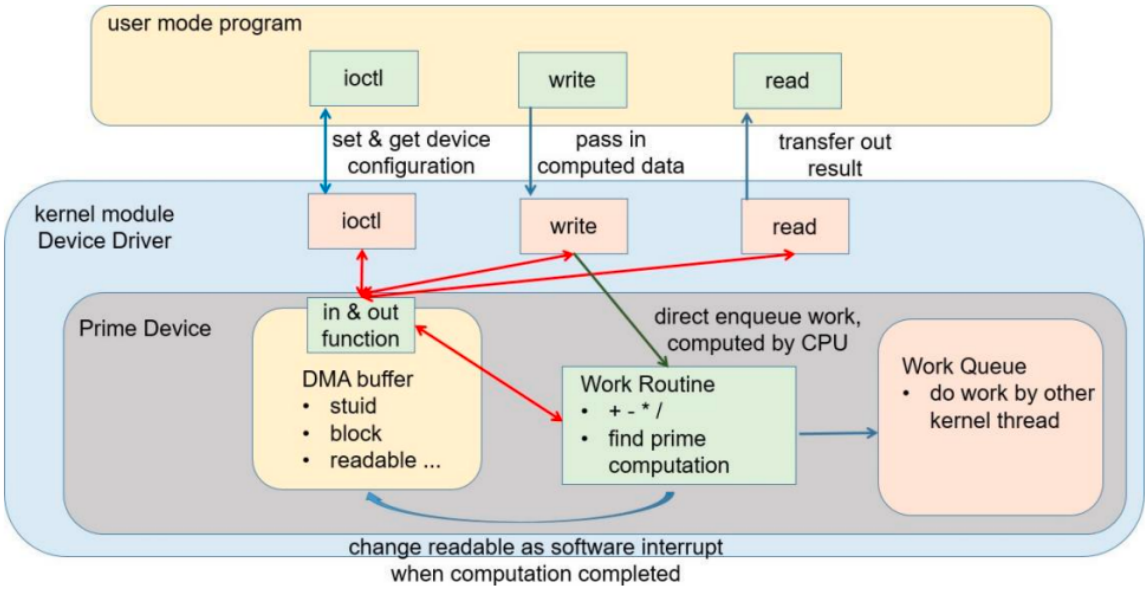

Figure: Copied from Homework 5 description, the glaobal view of this assignment

- For the Bonus part, I learned the concepts of Linux's steps of handling an interrupt. I acknowledge how to put a new device to the IRQ number I want and register an ISR. Also, implementing the interrupt handler is a very fresh and practical experience for me and help me to learn many things about how the system deals with interrupts in practice.

2. For the implementing details, I learned several things about how the details of APIs. For example, for the function `INIT_WORK();` I initially try to find out how to pass parameters through this function. Because I searched that INIT_WORK used to have three parameters like `INIT_WORK(struct work_struct *work, void (*function)(void *), void *data)` to pass the data to the function. However, in our kernel version, there are only two, so the passing of parameter may via some other methods like using `container_of` to find the pointer of the data struct we want to pass in. (However, DMA buffer is still used to pass the parameters in my program)

- In the bonus part, I learned some details of `request_irq()` and some details of its arguments like the difference between IRQF_DISABLED and IRQF_SHARED. Because I used to pass NULL as the last parameter and failed, which results from IRQF_SHARED that enables the interrupt to be shared. If the cookie is not passed, when free the IRQ, the system may not identify the target to be freed.