CSC3150 Assignment 5 Report

Name: Hu Yiyan

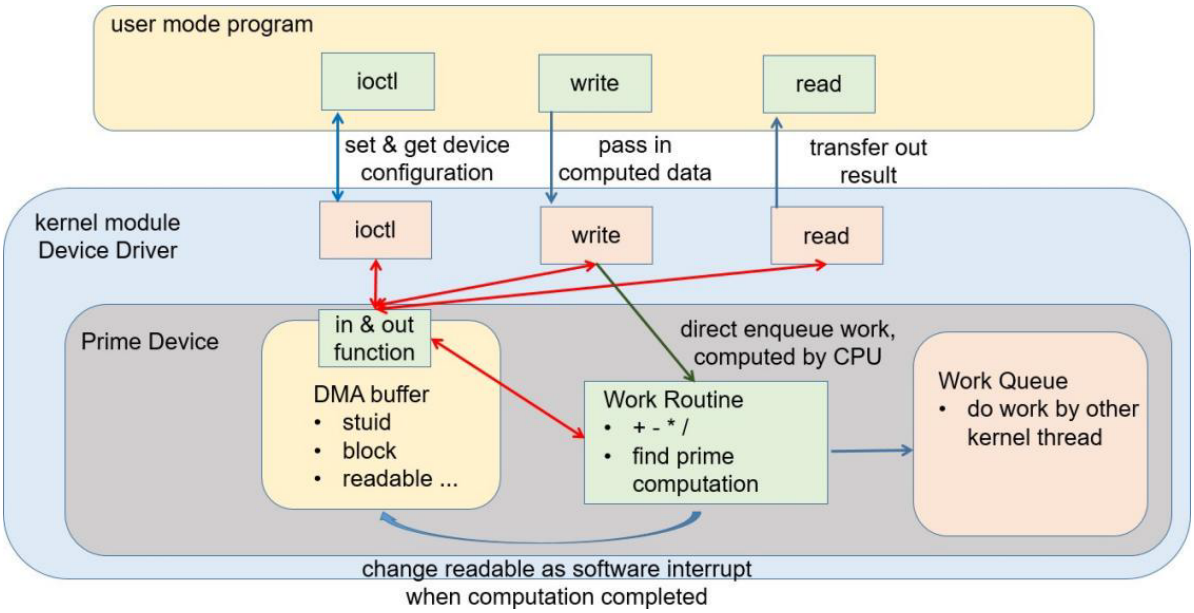
Student ID: 119010104

Design

1. Source Part

Overview

In the assignment 5, we need to make a prime device in Linux and implement file operations to control the device. The global view of the assignment is shown in the figure below.



Initializing and Exiting the Kernel Module

```
module_init(init_modules);
module_exit(exit_modules);
```

1. Initialization

```
static int __init init_modules(void);
```

We need to initialize the kernel module at the first place. In the kernel module, we should: (a) register character device and make it alive, (b) allocate space for DMA buffer, (c) allocate space for the work routine.

- **Register character device and make it alive:**

  To register the device, we need to use the `cdev_alloc()` function and `alloc_chrdev_region()` function, we also need to set file operations and the owner of the device.

- **DMA buffer**

In linux, the dma-buf subsystem provides the framework for sharing buffers for hardware (DMA) access across multiple device drivers and subsystems, and for synchronizing asynchronous hardware access. In this project, DMA buffer is simply for transferring data within the kernel module.

- **Work routine**

  Work routine is an instance of the `work` structure. We can call some functions to manage the execution of the works. In this project, the operations of blocking and ono-blocking write is implemented by controlling the work queue.

2. Exit

```
static void __exit exit_modules(void);
```

When exiting the kernel, we need to (a) free the allocated spaces, (b) delete the device. Note that in the kernel module we should call `kfree()` to free the device.

File Operations

We need to implement several file operations to control the device.

1. IOCTL

```
static long drv_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);
```

The IOCTL is for the user to change the device configuration. It can be seen as the interface for user to interact with the kernel module. The IOCTL function will perform coordinate operations based on the commands that the user inputs. In this project, the IOCTL will do the following 6 works.

```
HW5_IOCSETSTUID -> set student id
HW5_IOCSETRWOK -> set if RW OK
HW5_IOCSETIOCOK -> set if ioctl OK
HW5_IOCSETIRQOK -> set if IRQ OK
HW5_IOCSETBLOCK -> set blocking or non-blocking writing mode
HW5_IOCWAITREADABLE -> wait if readable now
```

The functions change the device configuration by modifying the DMA buffer. The following figure is the implementation of one IOCTL operation.

```
if (cmd == HW5_IOCSETSTUID){ // set student ID
    int value;
    get_user(value, (int*)arg);
    myouti(value, DMASTUIDADDR);
    printk("%s:%s(): My STUID is %d\n", PREFIX_TITLE, __func__, value);
}
```

The operation corresponding to the `HW5_IOCWAITREADABLE` command is special. Instead of simply changing the content in the DMA buffer, the operation will constantly check the DMA buffer and wait until the readable flag is set in the DMA buffer. The waiting operation is implemented by a while loop. The following code illustrates the implementation of the waitreadable ioctl operation.

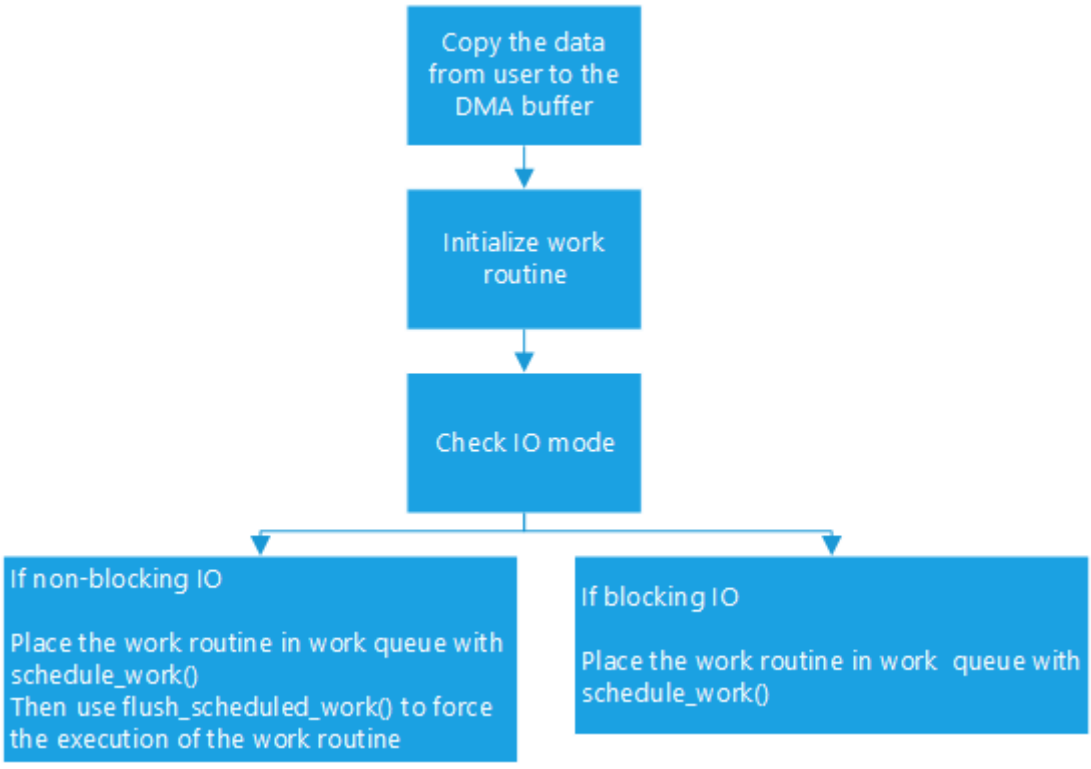
```
if (cmd == HW5_IOCWAITREADABLE) { // Wait if readable now
    while (myini(DMAREADABLEADDR)==0) { // sleep and wait
        msleep(1);
    }
    int value = myini(DMAREADABLEADDR);
    put_user(value, (int*)arg);
    printk("%s:%s(): Wait Readable %d\n", PREFIX_TITLE, __func__,
myini(DMAREADABLEADDR));
}
```

2. write

```
static ssize_t drv_write(struct file *filp, const char __user *buffer, size_t
ss, loff_t* lo)
```

The write function writes the data passed by the user into the device. The function will also check the device IO mode (blocking or non-blocking) and place the work in the work queue.

We can initialize the work routine via `INIT_WORK()`. Moreover, we can use `schedule_work()` and `flush_scheduled_work()` to control the execution of the work. The logic of the write operation is illustrated in the below flowchart.



Note: we should cannot use the `put_user()` to read the data of the type `dataIn` directly, for `put_user` only supports basic data types. To solve the issue, we can use the `copy_from_user` function.

```
copy_from_user (dataIn, buffer, sizeof(struct DataIn));
```

3. Read

```
static ssize_t drv_read(struct file *filp, char __user *buffer, size_t ss,
loff_t* lo)
```

The operation reads the computation result in the device and pass the result to the user. The read operation should also clear the readable flag to indicate that the result is just read.

Specifically, the read operation will: (a). read the result from the DMA buffer and pass the result to user, (b). clear the readable flag.

(a). Example code: read the result from FMA buffer and pass to user

```
int value;
value = myini(DMAANSADDR); // read from DMA buffer
put_user(value, buffer);
myouti(0, DMAANSADDR); // pass the value to the user
```

(b). Example code: clear the readable flag

```
myouti(0, DMAREADABLEADDR); // reset the flag by writing to DMA buffer
```

4. arithmetic

```
static void drv_arithmetic_routine(struct work_struct* ws)
```

The operation will execute the computation when the work is in queue. If the current IO mode is non-blocking, we should also set the readable flag in the DMA buffer to indicate that the result can be read.

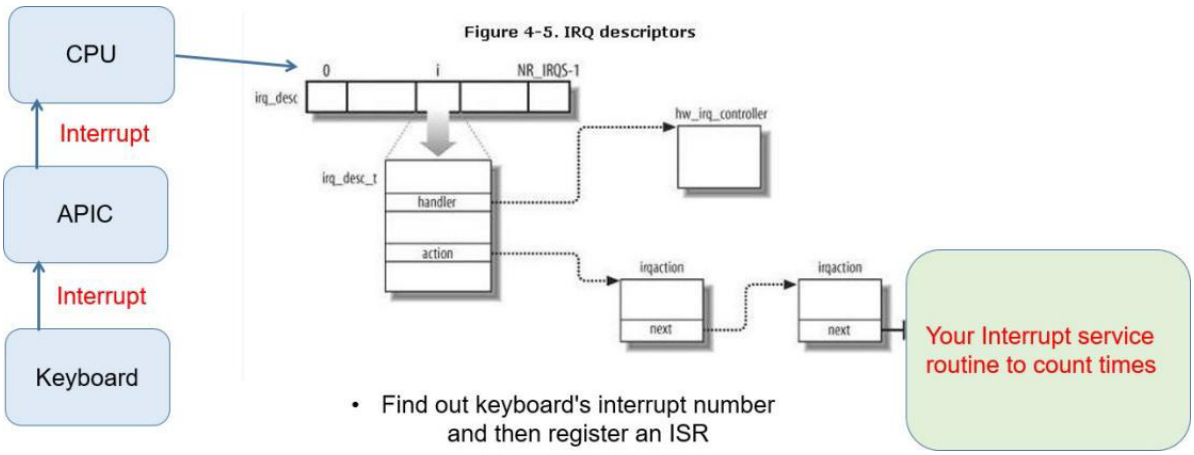
Example code: reset the readable flag

```
// rest readable flag if non-blocking write
if (!myini(DMABLOCKADDR)) myouti(1, DMAREADABLEADDR);
```

Bonus Part

Overview

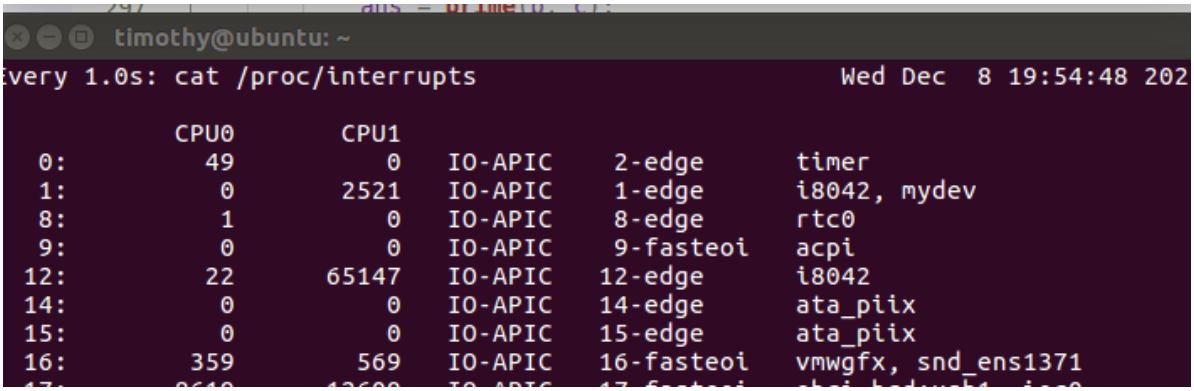
In the bonus part, we need to count the interrupt times of input device (keyboard for example). The logic of the interrupt is shown in the picture below.



Design Details

1. Get the IRQ number of the keyboard

I check the IRQ number of the keyboard interrupt using the command `watch -n 1 cat/proc/interrupts`.



I try to type something in the terminal, and I find that i8042 (IRQ number: 1) is increasing. Then I conclude that the IRQ number of keyboard input should be 1.

2. Allocate and free the interrupt request

We can use the `request_irq` function to allocate the IRQ in the kernel.

```
request_irq (
    unsigned int irq, // input line to interrupt, 1 in this assignment
    irq_handler_t handler, // interrupt handler function
    unsigned long irqflags, // interrupt flag, we can use IRQF_SHARED
    const char * devname, // device name, from the above figure, it is "mydev"
    void * dev_id // a cookie passed back to the handler function
    );
```

The function should be called in the `init_modules`. The following figure shows the function in the main.c.

```
ret = request_irq(1, countInterrupt, IRQF_SHARED, "mydev", dev_id);
```

We need to free the IRQ request once we delete the device in the `exit_modules()`, otherwise the kernel panic.

```
free_irq(1, dev_id);
```

3. Interrupt handler

We should implement the interrupt handler for the keyboard interrupt to count the keyboard interrupt times. I use the DMA buffer to store the interrupt count number. The following figure is the code of the interrupt handler.

```
// I think pasting code is more clear than explaining by text
static irqreturn_t countInterrupt(int irq, void *dev_id){
    int temp = myini(DMAIRQCOUNTADDR);
    temp ++;
    myouti(temp, DMAIRQCOUNTADDR);
    //printk("%s:%s(): Add up interrupt count: %d\n", PREFIX_TITLE, __func__,
temp);
    return IRQ_HANDLED;
}
```

Environment

Linux version

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

GCC version

```
timothy@ubuntu:~$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.10) 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.
```

Program Execution

To execute the program, you can follow the following steps:

```
root@ubuntu:/home/timothy/Projects/Ass5/source# make
...
sudo insmod mydev.ko
gcc -o test test.c
root@ubuntu:/home/timothy/Projects/Ass5/source# sudo sh ./mkdev.sh 243 0
crw-rw-rw- 1 root root 243, 0 Dec  8 17:34 /dev/mydev
root@ubuntu:/home/timothy/Projects/Ass5/source# ./test
...
root@ubuntu:/home/timothy/Projects/Ass5/source# rmmod mydev.ko
root@ubuntu:/home/timothy/Projects/Ass5/source# sh ./rmdev.sh
ls: cannot access '/dev/mydev': No such file or directory
```

Note:

> "..." indicates the terminal output.
>
> You may change the device number

Program Output

I only demo the output corresponding to `arithmetic(fd, 'p', 100, 20000)` in the report.

- Terminal

```
root@ubuntu:/home/timothy/Projects/Ass5/source# ./test
..............Start..............
100 p 20000 = 225077

Blocking IO
ans=225077 ret=53

Non-Blocking IO
Queueing work
Waiting
Can read now.
ans=225077 ret=53

..............End..............
```

- Kernel

```
root@ubuntu:/home/timothy/Projects/Ass5/source# dmesg | tail -23
[10915.639377] OS_AS5:init_modules(): register chrdev(243,0)
[10915.639386] OS_AS5:init_modules(): request_irq 1 returns 0
[10915.639387] OS_AS5:init_modules(): allocate dma buffer
[10993.450157] OS_AS5:drv_open(): device open
[10993.450189] OS_AS5:drv_ioctl(): My STUID is 119010104
[10993.450190] OS_AS5:drv_ioctl(): RM OK
[10993.450191] OS_AS5:drv_ioctl(): IOC OK
[10993.450192] OS_AS5:drv_ioctl(): IRQ OK
[11005.897004] OS_AS5:drv_ioctl(): Blocking IO
[11005.897013] OS_AS5:drv_write(): queue work
[11005.897022] OS_AS5:drv_write(): block
```

```
[11018.165271] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
[11018.165416] OS_AS5:drv_read(): ans = 225077
[11018.165448] OS_AS5:drv_ioctl(): Non-Blocking IO
[11018.165451] OS_AS5:drv_write(): queue work
[11030.526596] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
[11030.535462] OS_AS5:drv_ioctl(): Wait Readable 1
[11030.535495] OS_AS5:drv_read(): ans = 225077
[11030.535622] OS_AS5:drv_release(): device close
[11044.488486] OS_AS5:exit_modules(): interrupt count = 101
[11044.488490] OS_AS5:exit_modules(): free dma buffer
[11044.488515] OS_AS5:exit_modules(): unregister chrdev
[11044.488516] OS_AS5:exit_modules(): ..............End..............
```

Note: the interrupt count may vary. It depends on how many times you type on the keyboard & different machines (my computer seems treat pressing space as 2 interrupts).

Conclusion

In the assignment 5, I:

- Learn more about the IO structure of the operating system.
- Make a device in Linux, study how to allocate a device.
- Implement file operations to control the device, get more insights into how the user and kernel module device driver interacts.
- Have better understanding on blocking write and non-blocking write.
- Learn about the IRQ, practice on using the IRQ to count keyboard interrupt.

This is the last CSC3150 assignment, I feel that my programming skills improve a little bit through implementing these five assignments.

Thanks for being patient to grade my code and read the report!

Have a nice day and happy new year! (Though it may be a little earlier to say "happy new year")