CSC3150 Assignment 5

119010054 Dai Yulong

1. Design Ideas

In assignment 5, I write program to make a prime device in Linux and implement file operations in kernel module to control the device. The global view of my program is the same as the desription. I will introduce the detailed information of initialize and exit module and file operation functions (including bonus):

1.1 Initialize and exit module

- Module initialization is implemented in `init_modules(void)`, which uses register_chrdev, initializes the prime device, allocates DMA buffer, allocate work routine and IRQ (bonus).
    - In the first step of initialization, the program registers range of device number and gets the major and minor number of the device. (Major number would be used to identifies the driver and minor number would be used to specify the device).
    - In the second step, the program allocates a adev structure and initialize it with the file operations.
    - In the third step, the program initializes the DMA buffer, and allocates a work routine and IRQ (bonus).
- Module exit is implemented in `exit_modules(void)`, which prints the number of interrupts, frees DMA buffer, deletes character device, frees work routine and frees IRQ (in order). First, the program calls `MKDEV` to create a value to compared with the kernel devie number. Then `cdev_del()` is used to remove the device from the system. Finally, use `unregister_chrdev_region()` to unregister the range of device number. Notice that the program uses `free_irq(IRQ_NUM, (void *)&drv_interrupt_count)` to free irq

1.2 File operations

1. `drv_arithmetic_routine(struct work_struct* ws)`: Here I implement arthemetic routine. First, the program reads data from the DMA buffer, including an opcode and two operands. They are read by `myinc()`, `myini()`, `myins()` respectively. Then by checking the type of the operand, the program performs different arithmetic operations on the two operands. These arithmetic operations include `+, -, *, %, p`. Note that the operation `p` is to find a c-th prime number larger than b. I directly transferred the `prime()` function from `test.x` to `main.c`. After the arithmetic operation, the program writes back the computed answer and changes the readable settings.

2. `drv_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)`: This function is to change the device configuration. It defines many types of operations with swith case to do coordinated work. For each masked label, the program firstly use `get_user(value, (int *)arg);` to get data from user. Then store these data into the corresponding position of DMA buffer. Specially notice that when the masked label is `HW5_IOCSETBLOCK`, the program check whether it's a in blocking mode (1 represents blocking mode; 0 represents non-blocking mode). In the case of masked label `HW5_IOCWAITREADABLE`, I implement a while loop keep checking readable setting. If unreadable, use `msleep(1000)` to let this work to be sleeped (still can compute). If readable (after computation, readable setting changes), the loop will stops.
3. `drv_write(struct file *filp, const char __user *buffer, size_t ss, loff_t* lo)`: This function is used to write data into the device. It has three part:
    - Write data into DMA: Here we need to transfer data from user mode to kernel mode. Because `get_user` cannot transfer data with type DataIn. It can only transfer simple types like char and int. Therefore, I use `copy_from_user` to copy data from user space (here it's `buffer`).
    - Initialize queue work: The program calls `INIT_WORK(work, func)`. The `work` here is `work_routine`, which is already defined in the module initialization function.Then the program calls `schedule_work(work)` to put work task in global workqueue.
    - Decide IO mode: The program reads the mode in DMA buffer (blocking or nonnlocking), then perform corresponding operations. For blocking IO mode, write operation would not return until the arithmetic operation is done. Here I use `flush_scheduled_work()` . For non-blocking IO mode, just return immediately once the task is put into the global workqueue.
4. `drv_read(struct file *filp, char __user *buffer, size_t ss, loff_t* lo)`: The program gets the computation results stored in the DMA buffer and print it out. Then it clears the results in DMA buffer and sets the read flag as false. Remember use `put_user` to transmit the result from kernel mode to user mode.
5. `drv_interrupt_count(int irq, void *dev_id)` (Bonus): If the IRQ type equals IRQ_NUM (which equals to 1 and represents the keyboard interrupt), the program reads IRQ number in DMA buffer (by myini), increases one, and writes it back to DMA buffer (by myouti). Notice that IRQ_NUM represents IRQ number of keyboard. `IRQF_SHARED` means that many devices share the same interrupt line (so that the action that counter plus one would be executed when keyboard interrupt happens).

2. Environments and Run Methods

2.1 Environments

My project adopts the recommended environment:

- Linux Version

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

- Linux Kernel Version

```
dai@ubuntu:~$ uname -r
4.10.14
```

- GCC Version

```
dai@ubuntu:~$ 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.
```

2.2 Run Methods

For first time active your device, you need five steps (commends) to run my program

1. Using `cd` to enter the "source" directory, then type `make` command and enter.
2. Type `dmesg` to check the available MAJOR and MINOR number.
3. Run `sudo sh ./mkdev.sh MAJOR MINOR` to create a file node for “mydev”.
4. Run test `./test`
5. Type `make clean` and enter

If you have already active your device, you only need three steps (commends) to run my program

1. Using `cd` to enter the "source" directory, then type `make` command and enter.
2. Run test `./test`:
3. Type `make clean` and enter

3. Screenshots of Sample output

- The sample user mode output:

```
dai@ubuntu:~/Desktop/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

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

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

- The sample kernel mode output (including bonus)

```
[ 3790.118811] OS_AS5:init_modules():..............Start..............
[ 3790.118812] OS_AS5:init_modules(): register chrdev(244,0)
[ 3790.118813] OS_AS5:init_modules(): allocate dma buffer
[ 3792.241393] OS_AS5:drv_open(): device open
[ 3792.241396] OS_AS5:drv_ioctl(): My STUID is = 119010054
[ 3792.241397] OS_AS5:drv_ioctl(): RW OK
[ 3792.241397] OS_AS5:drv_ioctl(): IOC OK
[ 3792.241398] OS_AS5:drv_ioctl(): IRQ OK
[ 3792.241408] OS_AS5:drv_ioctl(): Blocking IO
[ 3792.241409] OS_AS5:drv_write(): queue work
[ 3792.241410] OS_AS5:drv_write(): block
[ 3792.241473] OS_AS5:drv_arithmetic_routine(): 100 + 10 = 110
[ 3792.241477] OS_AS5:drv_read(): ans = 110
[ 3792.241480] OS_AS5:drv_ioctl(): Non-Blocking IO
[ 3792.241481] OS_AS5:drv_write(): queue work
[ 3792.241483] OS_AS5:drv_ioctl(): wait readable 1
[ 3792.241485] OS_AS5:drv_arithmetic_routine(): 100 + 10 = 110
[ 3793.249719] OS_AS5:drv_read(): ans = 110
[ 3793.249734] OS_AS5:drv_ioctl(): Blocking IO
[ 3793.249738] OS_AS5:drv_write(): queue work
[ 3793.249739] OS_AS5:drv_write(): block
[ 3793.253136] OS_AS5:drv_arithmetic_routine(): 100 - 10 = 90
[ 3793.253154] OS_AS5:drv_read(): ans = 90
[ 3793.253176] OS_AS5:drv_ioctl(): Non-Blocking IO
```

```
[ 3793.253181] OS_AS5:drv_write(): queue work
[ 3793.253184] OS_AS5:drv_ioctl(): wait readable 1
[ 3793.253190] OS_AS5:drv_arithmetic_routine(): 100 - 10 = 90
[ 3794.273652] OS_AS5:drv_read(): ans = 90
[ 3794.273667] OS_AS5:drv_ioctl(): Blocking IO
[ 3794.273670] OS_AS5:drv_write(): queue work
[ 3794.273672] OS_AS5:drv_write(): block
[ 3794.273797] OS_AS5:drv_arithmetic_routine(): 100 * 10 = 1000
[ 3794.273808] OS_AS5:drv_read(): ans = 1000
[ 3794.273818] OS_AS5:drv_ioctl(): Non-Blocking IO
[ 3794.273822] OS_AS5:drv_write(): queue work
[ 3794.273826] OS_AS5:drv_ioctl(): wait readable 1
[ 3794.273830] OS_AS5:drv_arithmetic_routine(): 100 * 10 = 1000
[ 3795.297705] OS_AS5:drv_read(): ans = 1000
[ 3795.297720] OS_AS5:drv_ioctl(): Blocking IO
[ 3795.297723] OS_AS5:drv_write(): queue work
[ 3795.297724] OS_AS5:drv_write(): block
[ 3795.297790] OS_AS5:drv_arithmetic_routine(): 100 / 10 = 10
[ 3795.297796] OS_AS5:drv_read(): ans = 10
[ 3795.297803] OS_AS5:drv_ioctl(): Non-Blocking IO
[ 3795.297806] OS_AS5:drv_write(): queue work
[ 3795.297810] OS_AS5:drv_ioctl(): wait readable 1
[ 3795.297815] OS_AS5:drv_arithmetic_routine(): 100 / 10 = 10
[ 3796.321174] OS_AS5:drv_read(): ans = 10
[ 3797.033573] OS_AS5:drv_ioctl(): Blocking IO
[ 3797.033579] OS_AS5:drv_write(): queue work
[ 3797.033580] OS_AS5:drv_write(): block
[ 3797.570348] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[ 3797.570419] OS_AS5:drv_read(): ans = 105019
[ 3797.570432] OS_AS5:drv_ioctl(): Non-Blocking IO
[ 3797.570434] OS_AS5:drv_write(): queue work
[ 3797.570436] OS_AS5:drv_ioctl(): wait readable 1
[ 3798.098857] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[ 3798.593867] OS_AS5:drv_read(): ans = 105019
[ 3801.605237] OS_AS5:drv_ioctl(): Blocking IO
[ 3801.605239] OS_AS5:drv_write(): queue work
[ 3801.605240] OS_AS5:drv_write(): block
[ 3803.897140] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
[ 3803.897223] OS_AS5:drv_read(): ans = 225077
[ 3803.897240] OS_AS5:drv_ioctl(): Non-Blocking IO
[ 3803.897241] OS_AS5:drv_write(): queue work
[ 3803.897244] OS_AS5:drv_ioctl(): wait readable 1
[ 3806.160371] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
[ 3806.976909] OS_AS5:drv_read(): ans = 225077
[ 3806.977066] OS_AS5:drv_release(): device close
[ 3814.443502] OS_AS5:exit_modules(): interrupt count = 36
[ 3814.443503] OS_AS5:exit_modules(): free dma buffer
```

```
[ 3814.443504] OS_AS5:exit_modules(): unregister chrdev
[ 3814.443508] OS_AS5:exit_modules():.............End.............
dai@ubuntu:~/Desktop/source$
```

4. Learning

After finishing the assignment 5, I learned several things about kernel mode and user mode. And I also have a deep understanding of kernel programming

Comprehensive understanding of transportation between kernel mode and user mode

- Learn about device driver: Linux Device Drivers include character devices (such as printers, terminals, and mice), block devices*(including all disk drives), and network interface devices. And I know the difference about them. In this assignment, I wrote program to implement a prime device in Linux.
- Be faimilar with how kernel module and user module interact with each other. The DMA buffer is convent and quick tool to store data. I learn the structure of prime device, how to implement a work routine and complete many arithmetic operations. I also know how to transfer data between kernel mode and user mode.
- Non-blocking IO and blocking IO: In this assignment, I also learn what's the difference of blocking IO and non-blocking IO. Blocking write need to wait computation completed. Non-blocking write just return after queueing work.

Familiar with many kernel programming tools (functions)

- Create file system node: I use `mknod` to make a directory entry and corresponding i-node for a special file. The major number identifies the driver associated with the device. The minor number is used only by the driver specified by the major number, which provides a way for the driver to differentiate among them.
- Initialize and remove a CDEV: `alloc_chrdev_region( dev_t * dev, unsigned baseminor, unsigned count, const char * name)` is to allocates a range of char device numbers. `cdev_alloc()` is to allocate a cdev structure. `cdev_init(struct cdev * cdev, const struct file_operations * fops)` is to initialize cdev, remembering fops
- Functions about work routine: `INIT_WORK()` and `schedule_work()` are used to queue work to system queue.