CSC3150 Operating Systems

Project 5 report

Name: Xiao Nan

Student ID: 119010344

Date: 2021/12/07

The Chinese University of Hong Kong, Shen Zhen

0. Introduction:

In homework 5 basic part I implemented a device driver, a kernel module, to control a device which can perform elementary computations. I implemented basic driver operations including device and driver allocation and initialization, reading, writing (with arithmetic routines scheduling) and I/O controlling. In the bonus part I implemented an IRQ handler for the device driver which monitors keyboard interrupts and counts the number of keyboard-generated IRQs happening.

1. Design of the program:

The normal part of the program consists of 5 major functions for driver operations: reading, writing, initializing module, exiting module, and I/O controlling.

In the module initialization function, the program would first register the device by assigning a certain pair of device major and minor number to it using alloc_chrdev_region function. Then the device descriptor cdev is allocated (using cdev_alloc()) and is initialized with the fops file operation structure defined in order to map the user-space file operations to the driver. Then after the device is added to the descriptor using cdev_add(), the work routine (for temporarily storing job) and the DMA buffer (for storing computational inputs, results, and I/O settings) are allocated (using kmalloc()) and initialized (using memset()).

In the writing function, the program would first transfer necessary data (operator and operands) to DMA. Then, after initializing the work routine using (INIT_work()), the program would schedule works according to the I/O mode read from the DMA buffer: If the I/O mode is blocking then the work would be put into the working queue using schedule_work() and the program would wait till the routine is finished using flush_scheduled_work() (note that the readable flag in DMA is set to 1 after the routines are flushed, in order to enable reading the work routine result). If the mode is non-blocking then the program would first set the readable bit in DMA to be 0, indicating that the result in DMA is unreadable, and then return immediately so that CPU can continue doing other works when the work routine is under execution.

The arithmetic function, scheduled in the writing function, is responsible for calculating in a way the DMA suggested using operands and operator. It obtains operator and operands from DMA, judges the operator and performs corresponding calculations in a “switch” sentence. After computation is finished, the result would be stored in the DMA buffer and if the current I/O mode is blocking, the readable flag in DMA would be set 1 so that the result is readable by ioctl().

The ioctl() function controls the I/O operation settings. By using such function the user program can set the blocking/non-blocking writing flag in DMA with the value the program desires (using command HW5_IOCWAITREADABLE) so as to control writing mode. The function also provides an interface to wait for the readable signal: When receiving command HW5_SETBLOCK, the driver would wait in a while loop (by continuously checking the readable flag in DMA) till the flag is set 1 and then return to the user program.

After operations dependent on the driver are completed, the exit_modules() would be used to remove the driver and unallocated the device. The exit_modules(), registered as module_exit() function, would first unregister the device and then free the cdev device descriptor. Finally the DMA buffer and work routine would be freed and the module exits successfully.

In the bonus part I added an interrupt handler in the device driver implemented in the normal part. During the module initialization the interrupt handler is registered with IRQ number 1 (corresponding to the keyboard interrupt number) and shared mode (so that it enables multiple handlers for the keyboard interrupt) using request_irq(). In the keyboard interrupt handler function, I simply increment the count stored in DMA (DMACOUNTADDR) with 1 and then return IRQ_HANDLED to notify the OS that the IRQ is handled properly. Finally when exiting the module, the total count in DMA would be printed out. Note that since the interrupt is very sensitive to keyboard hits, the count obtained may be larger than the number of keyboard hits.

2. Runtime environment:

2.1. Linux version:

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

2.2. GCC version:

```
namshoo@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.
```

2.3. Linux kernel version:

```
namshoo@ubuntu:/$ uname -r
4.15.0
```

3. Steps to execute my program:

3.1. Inserting the driver module:

In command line, you need to first unzip the .zip file. Make sure that the .zip file is under the current directory:

```
namshoo@ubuntu:/$ unzip Assignment_5_119010344.zip
```

"cd" to the "Source" execution directory, and type "make" to compile the source code. Make sure that the Makefile, main.c, ioc_hw5.h are under the current directory:

```
namshoo@ubuntu:/$ cd Assignment_5_119010344/source/
namshoo@ubuntu:/Assignment_5_119010344/source/$ $ make
```

3.2. Creating the device file:

Then, check the device MAJOR and MINOR numbers using the command "dmesg | grep chrdev" (MAJOR and MINOR are two numbers identifying a device):

```
namshoo@ubuntu:/Assignment_5_119010344/source/$ dmesg | grep chrdev
[xxx.xxxxxx] OS_AS5:init_modules(): register chrdev(MAJOR,MINOR)
```

Then, create the device file using the script "mkdev.sh" with arguments MAJOR and MINOR. Make sure that mkdev.sh is under the current directory:

```
namshoo@ubuntu:/Assignment_5_119010344/source/$ sudo ./mkdev.sh MAJOR MINOR
```

3.3. Running the test program:

Then, run the test program, and the outputs would be displayed in terminal:

```
namshoo@ubuntu:/Assignment_5_119010344/source/$ ./test
```

3.4 Removing the module and device file:

To remove the device files, you simply need to run the script:

```
1. namshoo@ubuntu:/Assignment_5_119010344/source/$ sudo ./rmdev.sh
```

To remove the driver module and the kernel object files, you simply need to run “make clean” with the help of the Makefile:

```
2. namshoo@ubuntu:/Assignment_5_119010344/source/$ make clean
```

Then, the kernel message outputs would be printed out in the terminal screen (due to the implementation in the Makefile).

4. Program output screenshots:

The results for test program (the case with all arithmetic functions tested) in terminal and in kernel dmesg are shown in figure 4.1, 4.2, respectively. The kernel dmesg output corresponding to the demo case is shown in figure 4.3:

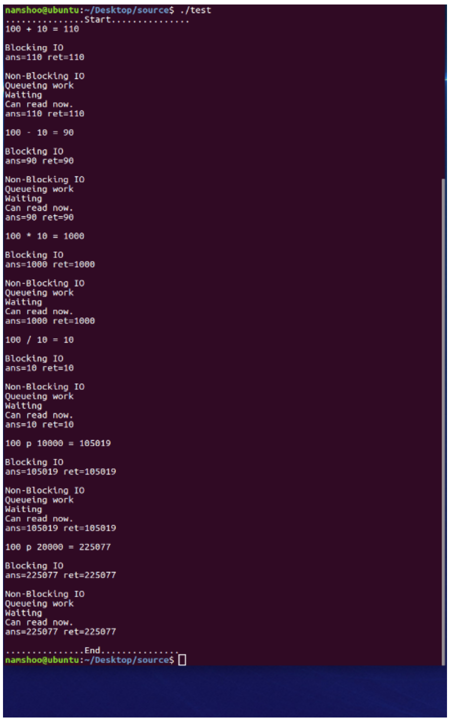

Figure 4.1: Terminal results

```
[ 5705.253089] OS_AS5:init_modules():...............Start...............
[ 5705.253092] OS_AS5:init_modules(): register chrdev(243,1)
[ 5705.253113] OS_AS5:init_modules(): request_irq 1 returns 0
[ 5705.253113] OS_AS5:init_modules(): allocate dma buffer
[ 5725.585245] OS_AS5:drv_open(): device open
[ 5725.585250] OS_AS5,drv_ioctl(): My STUID is = 123456789
[ 5725.585252] OS_AS5,drv_ioctl(): RW OK
[ 5725.585253] OS_AS5,drv_ioctl(): IOC OK
[ 5725.585255] OS_AS5,drv_ioctl(): IRQ OK
[ 5725.585297] OS_AS5,drv_ioctl(): Blocking IO
[ 5725.585300] OS_AS5:drv_write(): queue work
[ 5725.585304] OS_AS5:drv_write(): block
[ 5725.589596] OS_AS5:drv_arithmetic_routine(): 100 + 10 = 110
[ 5725.589633] OS_AS5,drv_read(): ans = 110
[ 5725.589695] OS_AS5,drv_ioctl(): Non-Blocking IO
[ 5725.589701] OS_AS5:drv_write(): queue work
[ 5725.589729] OS_AS5:drv_arithmetic_routine(): 100 + 10 = 110
[ 5725.589741] OS_AS5,drv_ioctl(): wait readable 1
[ 5725.589745] OS_AS5,drv_read(): ans = 110
[ 5725.589754] OS_AS5,drv_ioctl(): Blocking IO
[ 5725.589756] OS_AS5:drv_write(): queue work
[ 5725.589758] OS_AS5:drv_write(): block
[ 5725.589762] OS_AS5:drv_arithmetic_routine(): 100 - 10 = 90
[ 5725.589794] OS_AS5,drv_read(): ans = 90
[ 5725.589833] OS_AS5,drv_ioctl(): Non-Blocking IO
[ 5725.589838] OS_AS5:drv_write(): queue work
[ 5725.589842] OS_AS5:drv_arithmetic_routine(): 100 - 10 = 90
[ 5725.589848] OS_AS5,drv_ioctl(): wait readable 1
[ 5725.589874] OS_AS5,drv_read(): ans = 90
[ 5725.589883] OS_AS5,drv_ioctl(): Blocking IO
[ 5725.589885] OS_AS5:drv_write(): queue work
[ 5725.589887] OS_AS5:drv_write(): block
[ 5725.589890] OS_AS5:drv_arithmetic_routine(): 100 * 10 = 1000
[ 5725.589907] OS_AS5,drv_read(): ans = 1000
[ 5725.589938] OS_AS5,drv_ioctl(): Non-Blocking IO
[ 5725.589941] OS_AS5:drv_write(): queue work
[ 5725.589946] OS_AS5:drv_arithmetic_routine(): 100 * 10 = 1000
[ 5725.589972] OS_AS5,drv_ioctl(): wait readable 1
[ 5725.589977] OS_AS5,drv_read(): ans = 1000
[ 5725.590007] OS_AS5,drv_ioctl(): Blocking IO
[ 5725.590009] OS_AS5:drv_write(): queue work
[ 5725.590011] OS_AS5:drv_write(): block
[ 5725.590014] OS_AS5:drv_arithmetic_routine(): 100 / 10 = 10
[ 5725.590029] OS_AS5,drv_read(): ans = 10
[ 5725.590058] OS_AS5,drv_ioctl(): Non-Blocking IO
[ 5725.590062] OS_AS5:drv_write(): queue work
[ 5725.590066] OS_AS5:drv_arithmetic_routine(): 100 / 10 = 10
[ 5725.590093] OS_AS5,drv_ioctl(): wait readable 1
[ 5725.590098] OS_AS5,drv_read(): ans = 10
[ 5726.392087] OS_AS5,drv_ioctl(): Blocking IO
[ 5726.392090] OS_AS5:drv_write(): queue work
[ 5726.392093] OS_AS5:drv_write(): block
[ 5726.995059] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[ 5726.995136] OS_AS5,drv_read(): ans = 105019
[ 5726.995153] OS_AS5,drv_ioctl(): Non-Blocking IO
[ 5726.995156] OS_AS5:drv_write(): queue work
[ 5726.995159] OS_AS5,drv_ioctl(): wait readable 1
[ 5727.627251] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[ 5727.635886] OS_AS5,drv_read(): ans = 105019
[ 5731.072716] OS_AS5,drv_ioctl(): Blocking IO
[ 5731.072720] OS_AS5:drv_write(): queue work
[ 5731.072723] OS_AS5:drv_write(): block
[ 5733.674038] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
[ 5733.674136] OS_AS5,drv_read(): ans = 225077
[ 5733.674155] OS_AS5,drv_ioctl(): Non-Blocking IO
[ 5733.674158] OS_AS5:drv_write(): queue work
[ 5733.674914] OS_AS5,drv_ioctl(): wait readable 1
[ 5736.253867] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
[ 5736.266833] OS_AS5,drv_read(): ans = 225077
[ 5736.266954] OS_AS5:drv_release(): device close
[ 5758.486527] OS_AS5,exit_modules(): interrupt count = 209
[ 5758.486542] OS_AS5,exit_modules(): free dma buffer
[ 5758.486542] OS_AS5,exit_modules(): unregister chrdev
[ 5758.486575] OS_AS5:exit_modules():.............End.............
namshoo@ubuntu:~/Desktop/source$
```

Figure 4.2: dmesg results (full)

```
[ 2007.524329] OS_AS5:init_modules():...............Start...............
[ 2007.524331] OS_AS5:init_modules(): register chrdev(243,1)
[ 2007.524336] OS_AS5:init_modules(): request_irq 1 returns 0
[ 2007.524336] OS_AS5:init_modules(): allocate dma buffer
[ 2020.075013] OS_AS5:drv_open(): device open
[ 2020.075020] OS_AS5,drv_ioctl(): My STUID is = 123456789
[ 2020.075022] OS_AS5,drv_ioctl(): RW OK
[ 2020.075024] OS_AS5,drv_ioctl(): IOC OK
[ 2020.075025] OS_AS5,drv_ioctl(): IRQ OK
[ 2023.471090] OS_AS5,drv_ioctl(): Blocking IO
[ 2023.471095] OS_AS5:drv_write(): queue work
[ 2023.471098] OS_AS5:drv_write(): block
[ 2026.070887] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
[ 2026.070995] OS_AS5,drv_read(): ans = 225077
[ 2026.071012] OS_AS5,drv_ioctl(): Non-Blocking IO
[ 2026.071015] OS_AS5:drv_write(): queue work
[ 2026.071017] OS_AS5,drv_ioctl(): wait readable 1
[ 2028.725280] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
[ 2028.730283] OS_AS5,drv_read(): ans = 225077
[ 2028.730363] OS_AS5:drv_release(): device close
[ 2044.102066] OS_AS5,exit_modules(): interrupt count = 134
[ 2044.102105] OS_AS5,exit_modules(): free dma buffer
[ 2044.102105] OS_AS5,exit_modules(): unregister chrdev
[ 2044.102113] OS_AS5:exit_modules():.............End.............
namshoo@ubuntu:~/Desktop/source$
```

Figure 4.3: dmesg results (part, corresponding to demo)

5. Things I've learnt from the project:

In this project I learnt about how Linux manages the devices by mapping device into kernel space data structures and performing operations based on kernel functions. I learnt how Linux can read and write data with the help of the device and how DMA can accelerate data reading and writing and help improve the CPU utilization rate. I also learnt that Linux uses the mechanism of working queue to schedule the works for devices and for synchronizing blocking and non-blocking behaviors. I learnt what a device driver is and how that can be implemented in Linux kernel (especially how the logical structure of the device and its corresponding setting can be represented in the kernel). Moreover, from the bonus part, I learnt how the interrupt handler works when an interrupt is taking place. I learnt the way to register an interrupt handler for an IRQ and make it work in kernel.