The Chinese University of Hong Kong, Shenzhen

CSC 3150

Operating Systems

Assignment 5

Author:
Shi Wenlan

Student Number:
119010265

December 25, 2021

Contents

1 Task 1

1.1 How did I design the program

How to register a character device when module initialized: In the implementaion of *init_modules*, I use function *alloc_chrdev_region* to allocate a range of character device numbers.

How to initialize a cdev and add it to make it alive: First of all, define a global variable *dev_cdev* of type *cdev* pointer. Then, call the following functions in sequence in the implementaion of *init_modules*: *cdev_alloc* in the purpose of allocating a cdev structure; *cdev_init* in the purpose of initializing cdev, remembering fops, and making it ready to add to the system; *cdev_add* in the purpose of adding the device to the system, making it live immediately. Note that the *owner* of *dev_cdev* should be set to $THIS\_MODULE$ before calling *cdev_add*.

How to allocate DMA buffer: I can use *kmalloc* to allocate a space in the core area for the DMA buffer. However, considering that *kmalloc* does not process the contents of the allocated space, in order to avoid bugs caused by illegal reading (e.g. read before the first write), I used *kzalloc*, which combines the functions of *kmalloc* and *memset*, to complete the initialization of clearing while applying for space.

How to allocate work routine: Because the work routine has a its own initialization function $INIT\_WORK$, I can simply use *kmalloc* to allocate space for it. Note that the parameter flag should be GFP_KERNEL (on behalf of a process running in the kernel space).

How to implement read operation for my device: In the implementaion of *drv_read*, I use *myini* to get the data stored in the port DMAANSADDR in the DMA buffer. Then I use *put_user* to write it into user space.

How to implement write operation for my device: The implementaion of *drv_write* is mainly divided into three parts.

Part 1: use *kmalloc* to allocate kernel space for *dataIn* (a DataIn pounter). Then use $raw\_copy\_from\_user$ to copy a block (of size *ss*) of data from user space (maybe I should use $copy\_from\_user$, but the compiler will report an implicit declaration of function error for this function. I think I do not have the permission to modify the includes of the template, so I change the function to $raw\_copy\_from\_user$, but fortunately this function can achieve the desired effect). And use *myoutc*, *myouti*, *myouts* copy operator, operand 1, and operand 2 from *dataIn* to the port

DMAOPCODEADDR, DMAOPERANDBADDR, DMAOPERANDCADDR in the DMA buffer respectively.

Part 2: use $INIT\_WORK$ to initialize work (*drv_arithmetic_routine*) from an allocated buffer (*work_routine*). Then *schedule_work* to put work task in global workqueue. If the IO mode is blocking, then we also need to use $flush\_scheduled\_work$ to flush work on work queue.

Part 3: use *myini* to read IO mode (blocking or non-blocking) from the port DMABLOCKADDR in the DMA buffer. The IO mode decide whether to call $flush\_scheduled\_work$.

How to implement ioctl settings for my device: *drv_ioctl* has 6 different branchs: set student ID, set if RW OK, set if ioctl OK, set if IRQ OK, set IO mode, wait if readable. Each time this function is called, one of the branches will be executed according to the parameter *cmd*. The first five branches are very similar, they all use *get_user* to get the data from user space first, and then use *myouti* to store the data into the corresponding port of the DMA buffer. In the last branch, use *myini* in a while loop to keep checking the flag stored in the port DMAREADABLEADDR of the DMA buffer, only when the flag is true, the loop can be broke. In order to prevent checks from being too frequent, I added *msleep* to the while loop to reduce the frequency of checks.

How to implement arithmetic routine for my device: First of all, use *myinc*, *myini* and *myins* to read operator, operand 1 and operand 2 from corresponding port of DMA buffer respectively. Secondly, use *myouti* to set readable flag in the port DMAREADABLEADDR of DMA buffer to unreadable (false or 0). Thirdly, calculate the formula and store the result to the port DMAANSADDR of the DMA buffer. Finally, use *myouti* to set readable flag in the port DMAREADABLEADDR of DMA buffer to readable (true or 1).

How to implement module exit functions: Use $kfree$ to free the allocated space for DMA buffer and work routine. Use *unregister_chrdev_region* to unregister a range of device numbers. Use *cdev_del* to remove *dev_cdev* from the system, and free the structure itself.

1.2 The environment of running my program

The code running environment is shown in the following figure:

```
main@ubuntu:~/Desktop/source$ uname -r
4.15.0-29-generic
main@ubuntu:~/Desktop/source$ cat /etc/issue
Ubuntu 16.04.5 LTS \n \l

main@ubuntu:~/Desktop/source$ 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.
```

Figure 1: The environment of running my program

1.3 The steps to execute my program

Enter *source* folder in terminal and do as following:

- Run `make`
- Run `dmesg` to check available device number
- Run `sudo ./mkdev.sh MAJOR MINOR` to build file node (MAJOR and MINOR are the available device number checked from previous step)
- Run `./test` to start testing
- Run `make clean` to remove the module and check the messages
- Run `sudo ./rmdev.sh` to remove the file node

Figure 2: The steps to execute my program

1.4 Screenshots of my program output

I have done the bonus, so just go to Bonus Section to check the screenshots (see Figure 3 and 4).

1.5 What did I learn from the task

I learn how to use functions about work routine to implement blocking and non-blocking write. I also learned how to transfer data between user space and kernal space. I have a deeper understanding of DMA buffer and how to implement it.

2 Bonus

2.1 How did I design the program

How to implement interrupt service routine:

First of all, I need to find out keyboard's interrupt number. By searching on the Internet, I found that i8042 represents the keyboard controller that controls the keyboard and mouse. Then I typed "watch -n 1 cat/proc/interrupt" in the terminal, and found that the interrupt number corresponding to this interrupt name is 1.

In the implementaion in *init_modules*, use *request_irq* to allocate a interrupt line numbered 1, and named *keyboard_irq_state*, with function *irq_handler* which will be called when the interrupt service routine occurs, and interrupt flag $IRQF\_SHARED$ which indicates that multiple interrupt handlers can share the interrupt number. Note that when using the $IRQF\_SHARED$ flag, the *dev_id* parameter is used as a unique identifier to identify the identity in the execution interrupt. In order to find the specified handle, it should be set to $(void*)(irq\_handler)$.

In the implementaion in *irq_handler*, add global variable *counter* by 1, where *counter* is used to count interrupt times.

In the implementaion in *exit_modules*, use $free\_irq$ to free the allocated interrupt service routine (numbered 1).

2.2 The environment of running my program

The code running environment is the same as task 1, see Figure 1.

2.3 The steps to execute my program

The steps to execute my program is the same as task 1, see Figure 2.

2.4 Screenshots of my program output

Test case: arithmetic(fd, 'p', 100, 20000);

```
..............Start..............
100 p 20000 = 225077

Blocking IO
ans=225077 ret=225077

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

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

Figure 3: User mode output

```
[36329.289658] OS_AS5:init_modules():..............Start..............
[36329.289660] OS_AS5:init_modules(): register chrdev(242,0)
[36329.289664] OS_AS5:init_modules(): request_irq 1 return 0
[36329.289665] OS_AS5:init_modules(): allocate dma buffer
[36351.383125] OS_AS5:drv_open(): device open
[36351.383129] OS_AS5:drv_ioctl(): My STUID is = 119010265
[36351.383130] OS_AS5:drv_ioctl(): RW OK
[36351.383131] OS_AS5:drv_ioctl(): IOC OK
[36351.383132] OS_AS5:drv_ioctl(): IRQ OK
[36354.250699] OS_AS5:drv_ioctl(): Blocking IO
[36354.250702] OS_AS5:drv_write(): queue work
[36354.250702] OS_AS5:drv_write(): block
[36356.430664] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
[36356.430708] OS_AS5:drv_read(): ans = 225077
[36356.430722] OS_AS5:drv_ioctl(): Non-Blocking IO
[36356.430724] OS_AS5:drv_write(): queue work
[36356.430727] OS_AS5:drv_ioctl(): wait readable 1
[36358.605014] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225077
[36358.717625] OS_AS5:drv_read(): ans = 225077
[36358.717762] OS_AS5:drv_release(): device close
[36375.779295] OS_AS5:exit_modules(): interrupt count = 33
[36375.779299] OS_AS5:exit_modules(): free dma buffer
[36375.779301] OS_AS5:exit_modules(): unregister chrdev
[36375.779301] OS_AS5:exit_modules():..............End..............
```

Figure 4: Kernal mode output

2.5 What did I learn from the task

I have improved my understanding of interrupt service routine. And I learned how to use related functions.