香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

CSC3150 Operating System

Assignment 5

I/O Device System

Ziqi Gao (高梓骐)

School of Data Science

The Chinese University of Hong Kong, Shenzhen

118010077@link.cuhk.edu.cn

1. Introduction

Versions of OS: Ubuntu 16.04.2 LTS (Xenial)

Kernel Version: 4.10.14

In this assignment, we need to implement a I/O system that allows the users to set the status of a character device, and write data into or read from the char device. That is, we need to design a device driver as an interface to link the kernel space and the user space together. All of those instructions on the device are performed via this interface. As a result, we can divide this assignment into three parts: Interface used in the user mode, a device driver in the kernel mode to send requests to the device controller, and a device controller to finish the work.

We have three system call methods here:

ioctl: This method can let the user to set or get the device configuration.

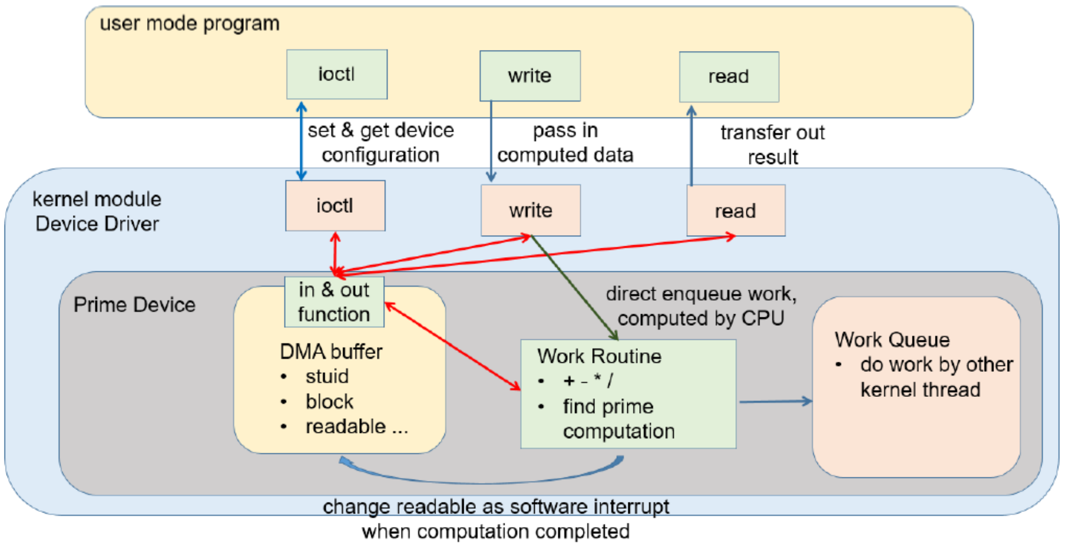
write: This method will pass some parameters to the device and let the device execute some work according to the inputs from the users.

read: This method will read the result from the device.

After the user call the system via the interface, we come to the kernel space to send requests to the device controller. Here we can save the corresponding setting configuration into our DMA (direct memory access) buffer. Also, we have a structure called file_opeartions given by the Linux kernel, which can map the interface to the device controller methods (like drv_write, drv_read).

Finally, we need to implement the device controller methods mentioned (drv_open, drv_write, …) before.

Global View:



2. Implementation Details

2.1 User Mode

In this part, I will introduce the interface given to the users. We can consider the methods of the interface as system calls.

ioctl (): This method needs a parameter as the request flag. This flag will decide which work to finish.

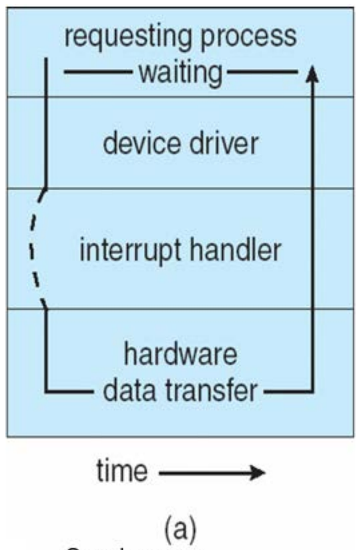
Here is an introduction for each flag:

a. HW5_IOC_SETSTUID: This system call will set the student ID using the third parameter ret.

b. HW5_IOCSETRWOK: This system call will read the status of the device. If the read or write function is equipped with this device. It will print out OK.

c. HW5_IOCSETIOCOK: This system call will read the status of the device. If ioctl method is set correctly, the device will print out the result in the kernel.

d. HW5_IOCSETIRQOK: This system call also reads the status of the device. If the interrupt service is done, the device will print out the result in the kernel.



(a)
Synchronous

e. HW5_IOCSETBLOCK: This system call sets the I/O mode. If the third parameter ret is 0: Non-blocking mode. If the third parameter is 1: Blocking mode. <u>I did not design the non-blocking/blocking read here, this system call will only affects the write mode.</u> The difference between blocking and non-blocking write will be introduced later.

f. HW5_IOCWAITREADABLE: This system call will synchronize the I/O of the application. As the graph at the right-side shows: this system call keeps the requesting thread sleeping until the write operation is done, to synchronize the job of those two threads. This system call is only used for non-blocking write because in the blocking write mode, this asynchronous will not occur. In non-blocking write mode, we cannot check the process of the write operation until it is finished. And this system call will wait for all the non-blocking threads to finish their jobs.

open (): This method opens the corresponding device. In Linux, every device is also considered as a file that can be edited by the users.

write (): This method sends some data structures to the device driver. This data structure contains three elements: an operator (+, -, x, ÷ or p), and two operands. Then the device driver will send those data into the device to conduct the arithmetic operations.

read (): This method reads the arithmetic result from the character device into the third parameter ret from the user.

release (): This method will be executed automatically after the life span of the device variable ends to free the kernel memory space.

2.2 Kernel Mode

In kernel mode, the task is simple. We need to map the system call to the corresponding device operations given by the device controller. Also, we should maintain a DMA buffer to save some data and the device configuration.

We use the file_operations structure from the Linux kernel to map the interface to the functions of the device.

ioctl -> drv_ioctl

open -> drv_open

read -> drv_read

write -> drv_write

release -> drv_realease

We have a 64 bytes DMA buffer. However, we will not use up all the space here. Also, the mapping process of the buffer is omitted to simplify this assignment. We will use the specific location to save our data or configuration of the device. Also, we deal with the problem of the memory alignment simply allocates 4 bytes to all kinds of data type (char, short, and int). The allocation of this buffer is as follows (byte address relative to the first element's address of the buffer):

0-3: Save student ID.

4-7: Save RW setting information.

8-11: Save IOCTL setting information.

12-15: Save whether the interrupt function is completed.

16-19: Count the completed interrupt function.

20-23: Save the result of the arithmetic operation

24-27: Save the information of whether the write work (arithmetic operation) is done.

28-31: Save the I/O mode: 1 – blocking, 0 – non-blocking

32-35: Save the arithmetic operator (+, -, x, ÷, p).

36-39: Save the first operand of the arithmetic operation.

40-43: Save the second operand of the arithmetic operation.

Besides those two parts, we need to create and register the character device, combining with the initialization of the DMA buffer when this module is inserted.

Finally, when the module is removed, we need to unregister the device, and free the memory in the kernel space.

2.3 Device Controller

In this part, I will introduce some details about the methods given by the device controller. All of those methods are called by the device driver in the kernel mode.

drv_open(): This function will open the character device, and return an integer back to the user space. Then, the user can use this returned value to control this device.

drv_read(): This function reads the arithmetic operations result from the user from the DMA buffer. The user should check the readable flag by calling ioctl to check whether the data is readable. Otherwise, the user can only get 0 from the device.

drv_write(): This function reads a structure $DataIn$ from the kernel space. This structure has three elements, one operator and two operands, which are very similar to the MIPS R-type instructions. This method saves those three elements to the DMA buffer for future use. This function has two writing modes according to the setting flag saved in DMA:

Blocking Write: In blocking write mode, the main thread calling this device function will add the work to the work queue, and wait for the complete of the writing and the calculation. As a result, we can directly read the data after the blocking write. The writing work is finished before our next operations.

Non-blocking Write: In non-blocking mode, the main thread calling this device function will **not** wait for the complete of the writing and the calculation. That is, our computer will execute the following instructions from the user without waiting for the writing work to be done. In this situation, it is not reasonable to read the data directly from the DMA buffer without checking whether the write work is done. Therefore, we need to check the readable flag from the DMA buffer before the read work. ioctl can check the status and wait for the finish of the writing operations to synchronize the I/O so that the read function can return the correct result to the user.

Both of two modes add another device method drv_arithmetic_routine to the work routine to finish the calculation using the three parameters got from the $DataIn$ structure.

drv_arithmetic_routine: This function reads the operator and two operands from the DMA buffer. Then, according to the operator, this method will conduct different arithmetic operations.

drv_release():

This method prints out the information to the kernel when the device is closed.

3. Difficulties

3.1 Transfer Data Between Kernel Space and User Space

This assignment is my first time to write a program contains both the user space and the kernel space. I cannot use a pointer in the kernel space to refer to an address in the user space. I have to use system calls (get_user, put_user, copy_from_user) to transfer the data.

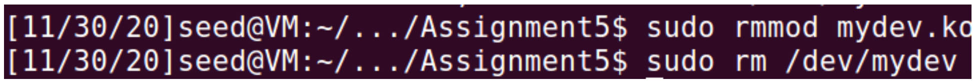
3.2 Transfer structure to the kernel space

Usually, I used get_user to get the data from the user space. However, when I coded the drv_write method that needed to transfer the $DataIn$ structure via a char pointer. get_user can only be applied to transfer simple variable like int or char. Therefore, I copy the data from the memory by bytes using the copy_from_user function, since the structure is saved contiguously in the memory.

4. Execution

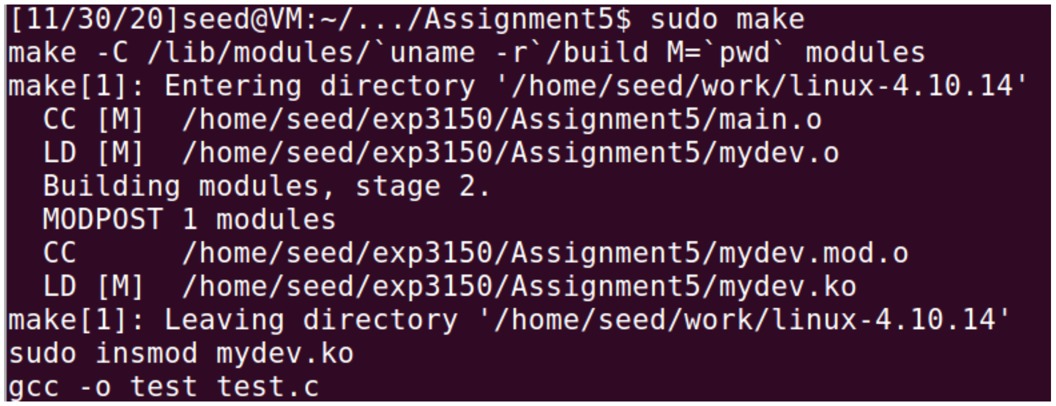
In order to test my program, you should do the following steps:

a. Remove the old mydev.ko and /dev/mydev file:



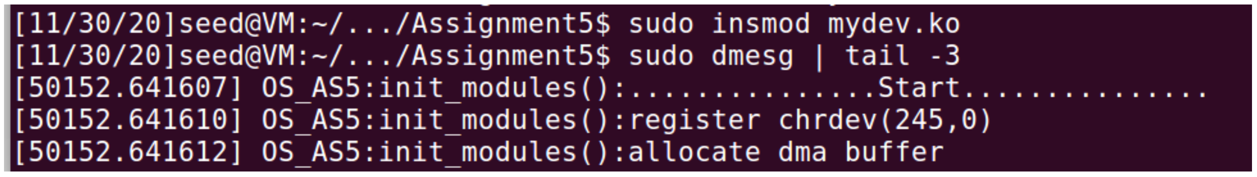
```
[11/30/20]seed@VM:~/.../Assignment5$ sudo rmmod mydev.ko
[11/30/20]seed@VM:~/.../Assignment5$ sudo rm /dev/mydev
```

b. Use makefile to compile the module:



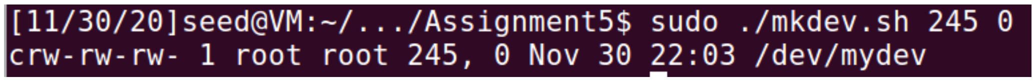
```
[11/30/20]seed@VM:~/.../Assignment5$ sudo make
make -C /lib/modules/`uname -r`/build M=`pwd` modules
make[1]: Entering directory '/home/seed/work/linux-4.10.14'
  CC [M]  /home/seed/exp3150/Assignment5/main.o
  LD [M]  /home/seed/exp3150/Assignment5/mydev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seed/exp3150/Assignment5/mydev.mod.o
  LD [M]  /home/seed/exp3150/Assignment5/mydev.ko
make[1]: Leaving directory '/home/seed/work/linux-4.10.14'
sudo insmod mydev.ko
gcc -o test test.c
```

c. Insert the module into the kernel and use dmesg to get the major and minor value of the device:



```
[11/30/20]seed@VM:~/.../Assignment5$ sudo insmod mydev.ko
[11/30/20]seed@VM:~/.../Assignment5$ sudo dmesg | tail -3
[50152.641607] OS_AS5:init_modules():...............Start...............
[50152.641610] OS_AS5:init_modules():register chrdev(245,0)
[50152.641612] OS_AS5:init_modules():allocate dma buffer
```

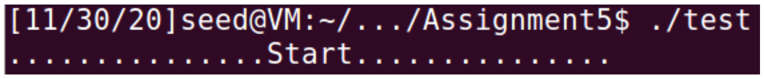
d. Use mkdev.sh to generate the device file using the major and minor value:



```
[11/30/20]seed@VM:~/.../Assignment5$ sudo ./mkdev.sh 245 0
crw-rw-rw- 1 root root 245, 0 Nov 30 22:03 /dev/mydev
```

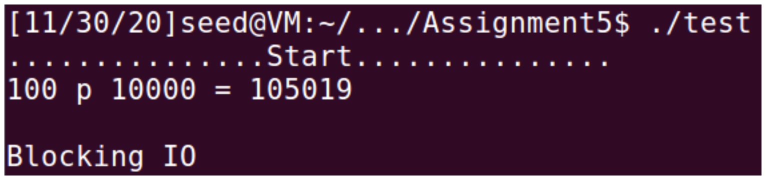
e. Run the test file from the template (the stuid is changed to 118010077):

We will first find the result like this:



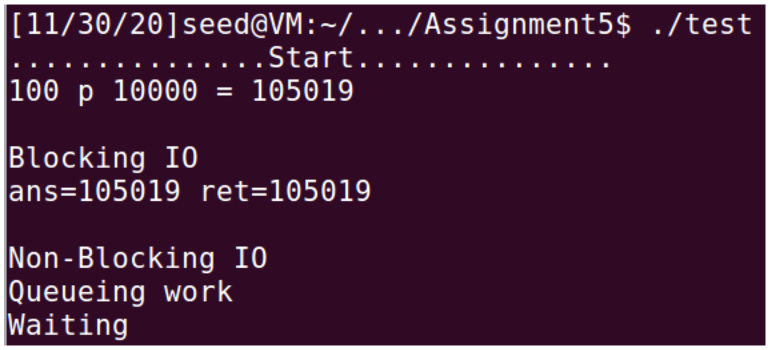
```
[11/30/20]seed@VM:~/.../Assignment5$ ./test
...............Start...............
```

After some moments we will see the answer of the arithmetic operation is got and we will see the write operation blocks other threads:



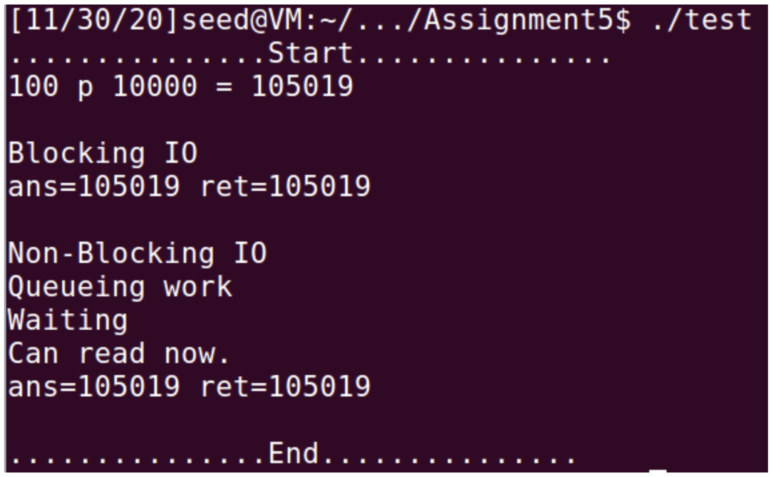
```
[11/30/20]seed@VM:~/.../Assignment5$ ./test
...............Start...............
100 p 10000 = 105019

Blocking IO
```

Later, we will find the result given by the device, and the non-blocking is finished. Then the main thread waits for the writing thread to finish.



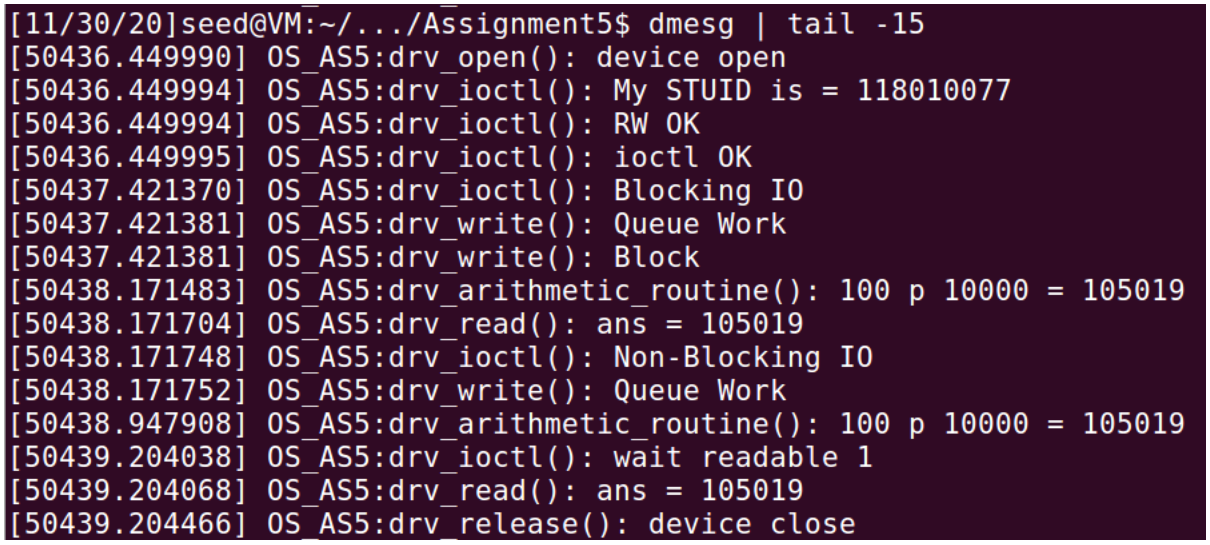
```
[11/30/20]seed@VM:~/.../Assignment5$ ./test
...............Start...............
100 p 10000 = 105019

Blocking IO
ans=105019 ret=105019

Non-Blocking IO
Queueing work
Waiting
```

After few seconds, the result of the non-blocking write is printed out, and the program finished here:



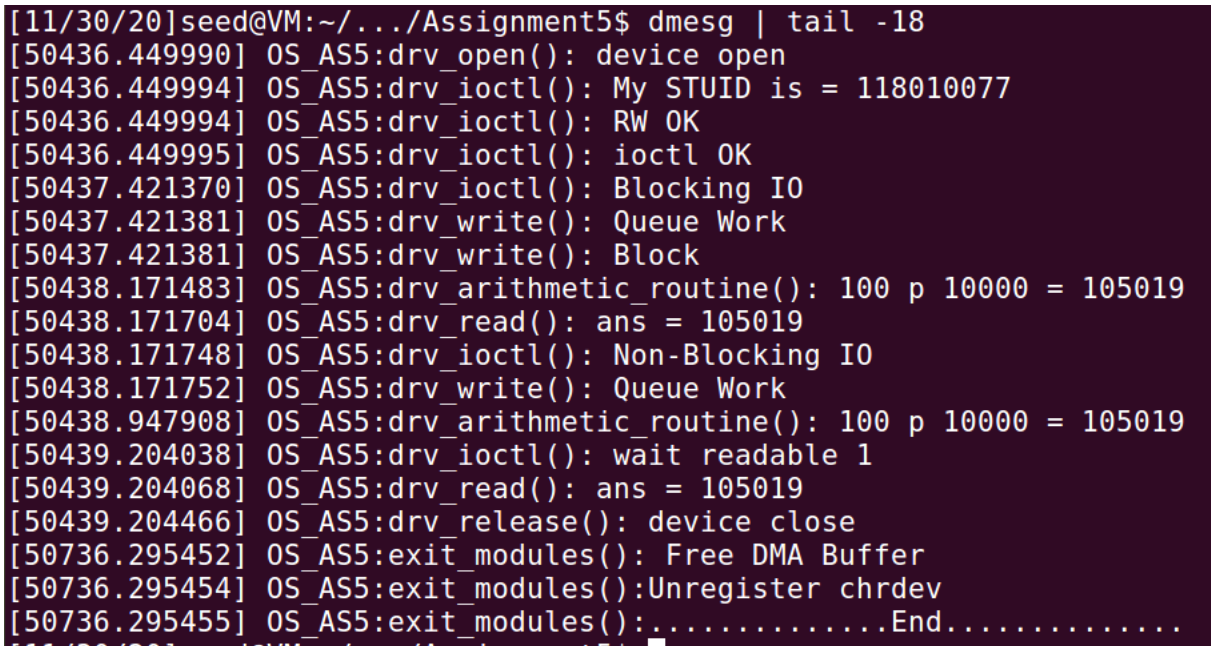
```
[11/30/20]seed@VM:~/.../Assignment5$ ./test
...............Start...............
100 p 10000 = 105019

Blocking IO
ans=105019 ret=105019

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

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

f. Check information from the kernel:



```
[11/30/20]seed@VM:~/.../Assignment5$ dmesg | tail -15
[50436.449990] OS_AS5:drv_open(): device open
[50436.449994] OS_AS5:drv_ioctl(): My STUID is = 118010077
[50436.449994] OS_AS5:drv_ioctl(): RW OK
[50436.449995] OS_AS5:drv_ioctl(): ioctl OK
[50437.421370] OS_AS5:drv_ioctl(): Blocking IO
[50437.421381] OS_AS5:drv_write(): Queue Work
[50437.421381] OS_AS5:drv_write(): Block
[50438.171483] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[50438.171704] OS_AS5:drv_read(): ans = 105019
[50438.171748] OS_AS5:drv_ioctl(): Non-Blocking IO
[50438.171752] OS_AS5:drv_write(): Queue Work
[50438.947908] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[50439.204038] OS_AS5:drv_ioctl(): wait readable 1
[50439.204068] OS_AS5:drv_read(): ans = 105019
[50439.204466] OS_AS5:drv_release(): device close
```

g. Remove the module and check the kernel information again:



```
[11/30/20]seed@VM:~/.../Assignment5$ dmesg | tail -18
[50436.449990] OS_AS5:drv_open(): device open
[50436.449994] OS_AS5:drv_ioctl(): My STUID is = 118010077
[50436.449994] OS_AS5:drv_ioctl(): RW OK
[50436.449995] OS_AS5:drv_ioctl(): ioctl OK
[50437.421370] OS_AS5:drv_ioctl(): Blocking IO
[50437.421381] OS_AS5:drv_write(): Queue Work
[50437.421381] OS_AS5:drv_write(): Block
[50438.171483] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[50438.171704] OS_AS5:drv_read(): ans = 105019
[50438.171748] OS_AS5:drv_ioctl(): Non-Blocking IO
[50438.171752] OS_AS5:drv_write(): Queue Work
[50438.947908] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[50439.204038] OS_AS5:drv_ioctl(): wait readable 1
[50439.204068] OS_AS5:drv_read(): ans = 105019
[50439.204466] OS_AS5:drv_release(): device close
[50736.295452] OS_AS5:exit_modules(): Free DMA Buffer
[50736.295454] OS_AS5:exit_modules():Unregister chrdev
[50736.295455] OS_AS5:exit_modules():..............End..............
```

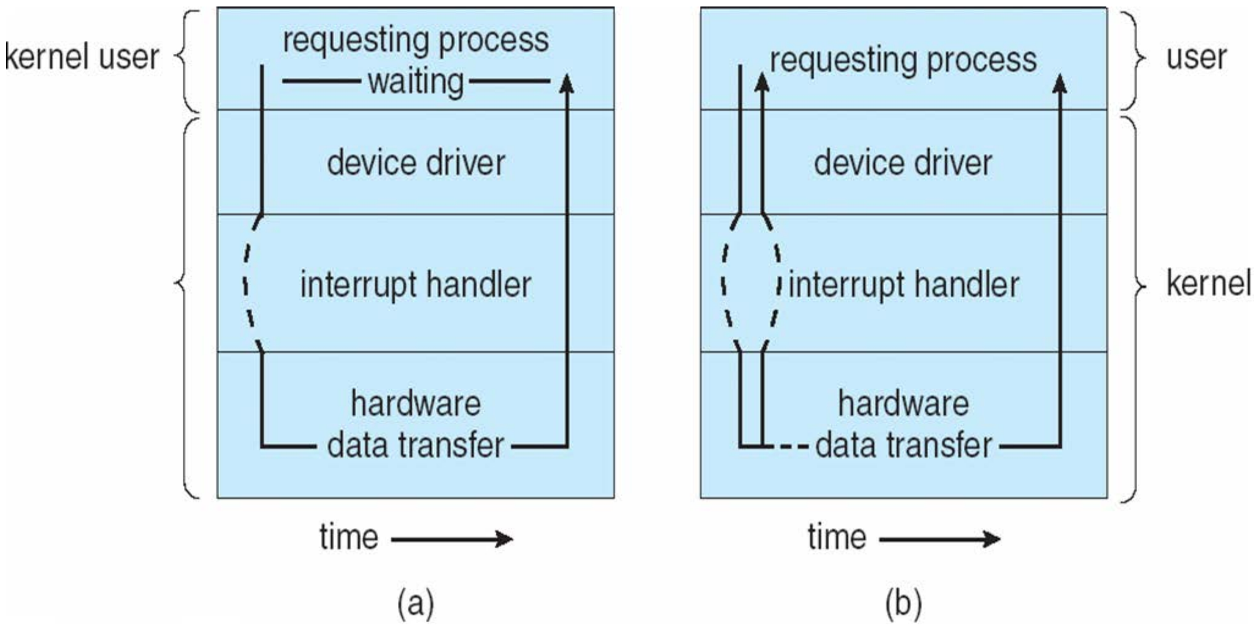
h. Finally, remove the device file using rmdev.sh

5. What I Learned

When I first met this project's instruction, I was confused by those tedious instructions, and did not have any idea to start my project. Then I learned some knowledge about the I/O system and divide this project into three parts: User Space Instructions (system call / interface) + Device Driver (kernel space) + Device Controller.

What I have done is just like the following graph, though the interrupt handler is simplified as a readable variable in my program. It is a good practice to link the device and the user with the device driver and the device controller. This experience gives me a deeper understanding of how I/O system works.

In the following works, I will try to learn more about the mechanism of the interrupt to improve and finish this full simplified I/O system.


kernel user
requesting process
waiting
device driver
interrupt handler
hardware
data transfer
time
(a)
requesting process
user
device driver
interrupt handler
kernel
hardware
data transfer
time
(b)