Assignment Report

CSC 3150 I/O Systems

Wei Wu (吴畏)

118010335

November 29, 2020

The School of Data Science

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

1. Environment of Running My Program.

Linux version 4.10.14 (root@VM)

g++ version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.4)

2. The Steps to Execute My Program.

a. Open the Linux Terminal.

b. Enter "sudo su" to log in with the root account.

c. Enter "cd xxx/Assignment_5_118010335/Source" to change directory to the assignment folder.

d. Enter "make" to build and insert a kernel module.

e. For the first time of activating the device:

   Enter "dmesg" to check available MAJOR and MINOR numbers. Enter "sudo ./mkdev.sh MAJOR MINOR" to create a file node for the device.

f. Enter "./test" to run the test program.

g. Enter "make clean" to remove the kernel module, clean the test executable file, and display the kernel messages related to this device.

3. How Did I Design My Program?

3.1. Program Logic

Generally speaking, this program is simulating a prime device in Linux, which can find the n-th prime number from any given integer. To control this device, we need to implement a set of file operations in kernel module. The big-picture is divided into two parts: the user mode program and the kernel module device driver. They correspond to the user space and the kernel space respectively. To transfer data between the user space and the kernel space, we can use get_user() and put_user().

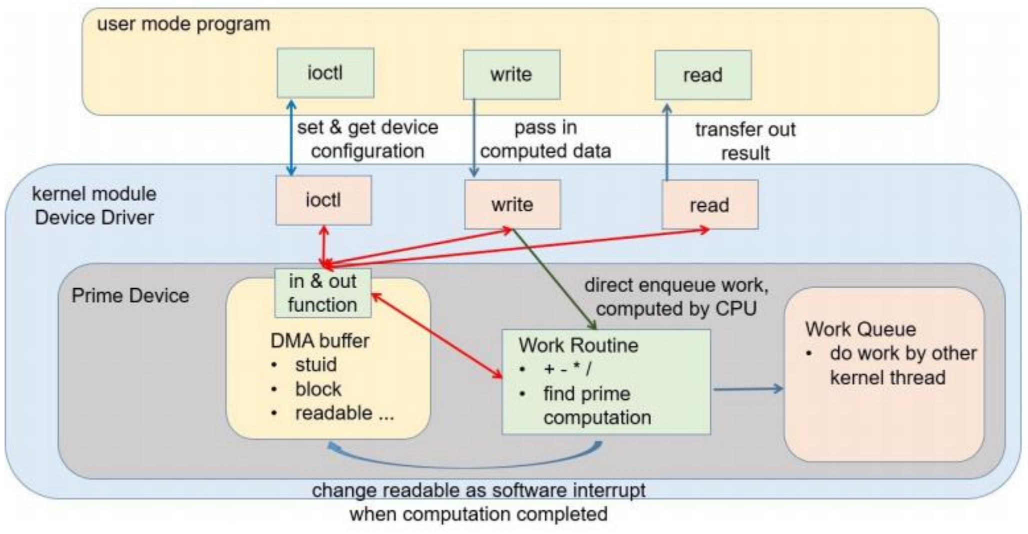
To start with, we need to make a filesystem node under the /dev directory for the device by mknod command. This filesystem node is used to record the information about the device.

The device takes Direct Memory Access (DMA) mechanism, which means it can read from and write into the main memory directly without going through the CPU. Therefore, we can allocate a block of memory as a DMA buffer to simulate the local memory and registers on the device. A series of in and out functions are implemented to access the DMA buffer.

As for the device driver, essential file operations, such as read and write, are implemented in a kernel module to control the device. The read function just gets the result from the device when ready. The write function initializes a work routine, which is associated with an arithmetic function for finding the n-th prime number from any given number, and puts the work routine into the work queue. When using the blocking I/O mode, the write function will immediately flush the work queue.

Besides, the ioctl function is needed to modify the device configuration. It provides subfunctions such as information printing, write mode toggling (blocking / non-blocking), and DMA readable checking.

To see more details, please refer to Section 6 of this Report.



3.2. The Original Program (Source)

3.2.1. Global Variables (In addition to those given in the template)

a. static int dev_major;

static int dev_minor;

The major and minor number of the device. Assigned in the function init_modules() after registering the character device. Used in the function exit_modules() to unregister the character device.

b. static struct cdev *dev_cdevp;

The pointer to the cdev structure. Assigned and used in the function init_modules() when initializing the cdev structure. Also used in the function exit_modules() to delete the character device.

c. static struct work_struct *work_routine;

The work routine pointer. Refer to Section 6.f of this Report for more details.

3.2.2. Data Structures

a. DMA buffer

```
// DMA
#define DMA_BUFSIZE 64
#define DMASTUIDADDR 0x0        // Student ID
#define DMARWOKADDR 0x4         // RW function complete
#define DMAIOCOKADDR 0x8        // ioctl function complete
#define DMAIRQOKADDR 0xc        // ISR function complete
#define DMACOUNTADDR 0x10       // interrupt count function complete
#define DMAANSADDR 0x14         // Computation answer
#define DMAREADABLEADDR 0x18    // READABLE variable for synchronize
#define DMABLOCKADDR 0x1c       // Blocking or non-blocking IO
#define DMAOPCODEADDR 0x20      // data.a opcode
#define DMAOPERANDBADDR 0x21    // data.b operand1
#define DMAOPERANDCADDR 0x25    // data.c operand2
```

Refer to Section 6.a of this Report for more details about the DMA.

DMA_BUFSIZE is the size of the buffer in bytes.

DMASTUIDADDR is the address where the student ID is stored.

DMARWOKADDR is the address where the Boolean variable indicating whether the RW function is complete is stored.

DMAIOCOKADDR is the address where the Boolean variable indicating whether the ioctl function is complete is stored.

DMAIRQOKADDR is the address where the Boolean variable indicating whether the interrupt service routine function is complete is stored.

DMACOUNTADDR is the address where the Boolean variable indicating whether the interrupt request counting function is complete is stored.

DMAANSADDR is the address where the computation result of the prime device is stored.

DMAREADABLEADDR is the address where the Boolean variable indicating whether the DMA is currently readable is stored.

DMABLOCKADDR is the address where the Boolean variable indicating the write mode (blocking / non-blocking) is stored.

DMAOPCODEADDR is the address where the opcode of the computation is stored.

DMAOPERANDBADDR is the address where the left-hand side operand of the computation is stored.

DMAOPERANDCADDR is the address where the right-hand side operand of the computation is stored.

b. static struct file_operations fops;

```
// cdev file_operations
static struct file_operations fops = {
        owner: THIS_MODULE,
        read: drv_read,
        write: drv_write,
        unlocked_ioctl: drv_ioctl,
        open: drv_open,
        release: drv_release,
};
```

This structure basically binds the required file operations to functions in this program. It serves as an interface between the Linux commands and the specific device driver. It works only if specified when initializing the device.

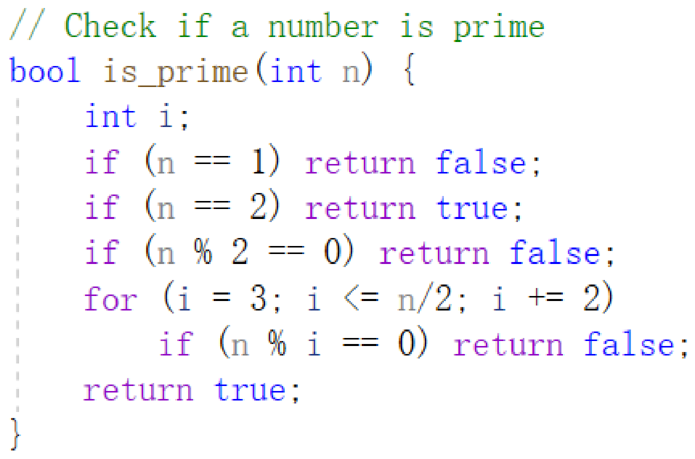
c. struct DataIn

```
// For input data structure
struct DataIn {
    char a;
    int b;
    short c;
} *dataIn;
```

The structure for input data, which includes an operator and two operands.

3.2.3. Utility Functions

bool is_prime(int n);

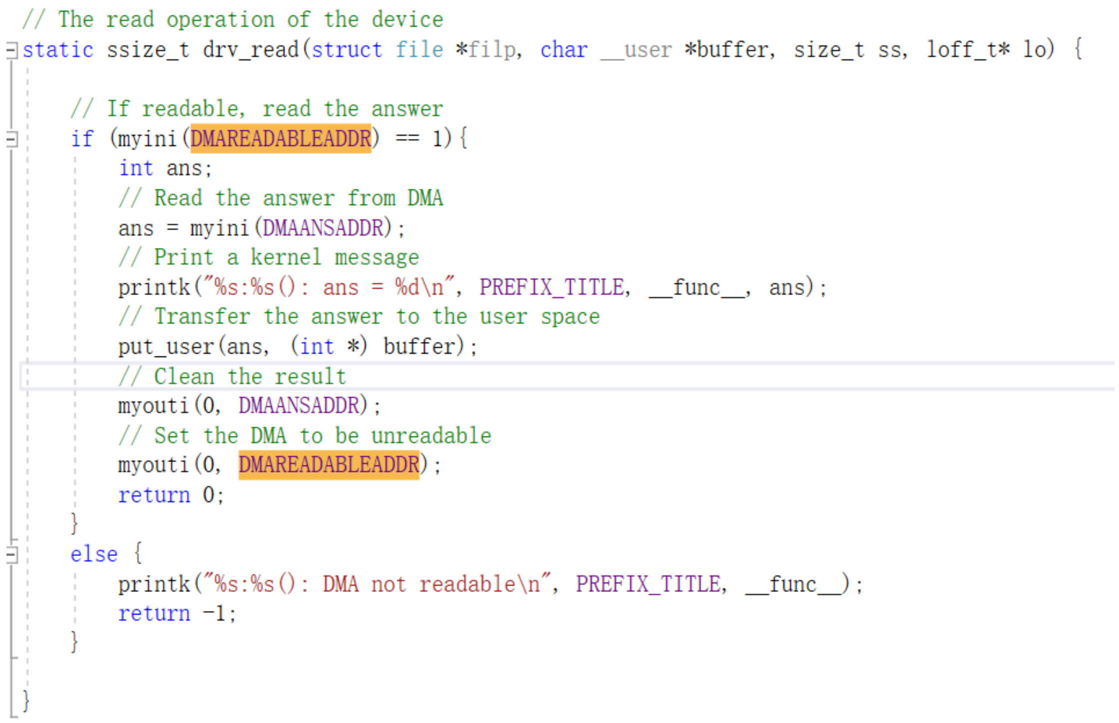


```
// Check if a number is prime
bool is_prime(int n) {
    int i;
    if (n == 1) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;
    for (i = 3; i <= n/2; i += 2)
        if (n % i == 0) return false;
    return true;
}
```

Check if a number is prime. Note that the efficiency of this function can be improved. Refer to Section 4.a of this Report for more details.

3.2.4. Major Functions

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

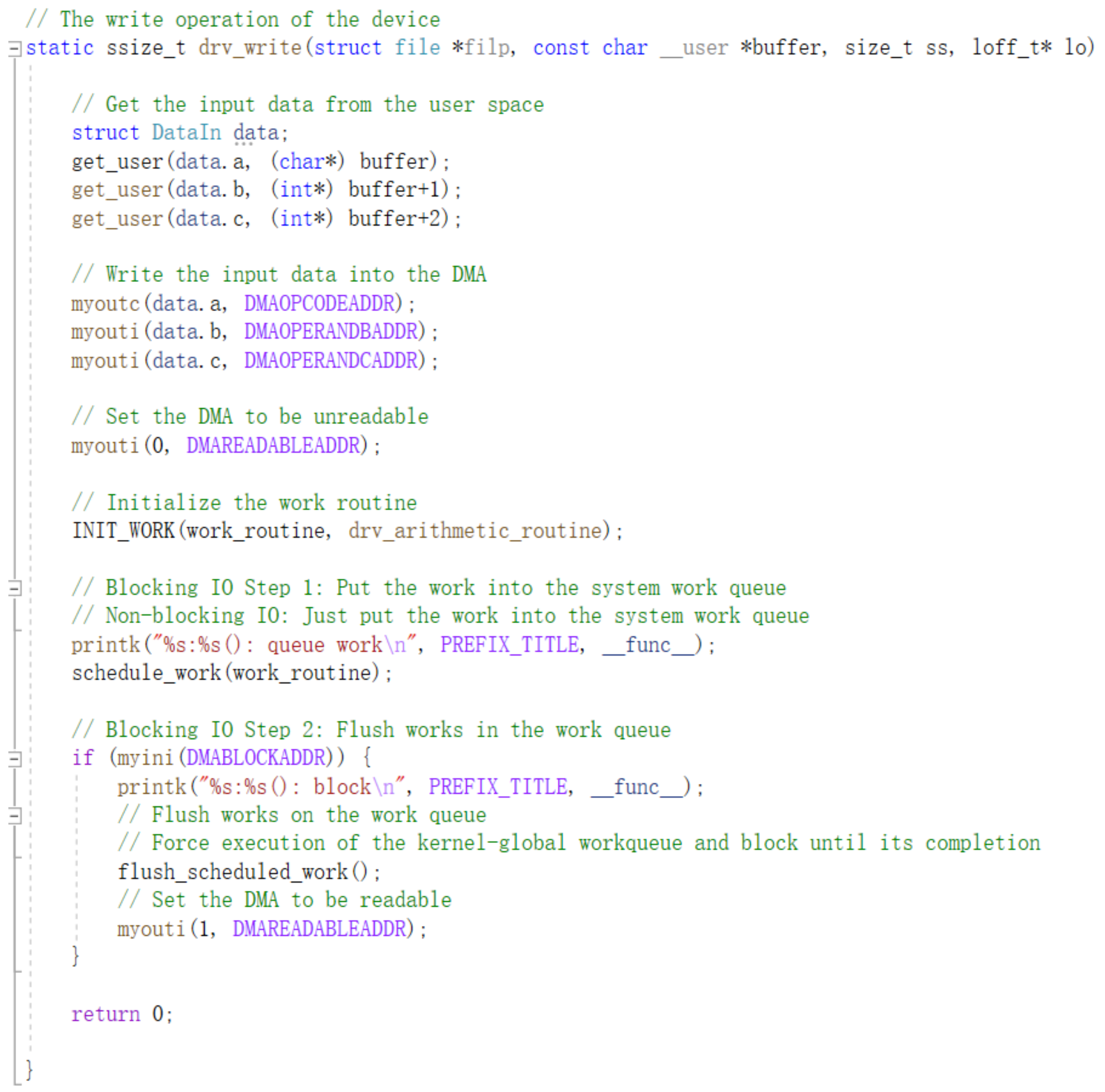


```
// The read operation of the device
static ssize_t drv_read(struct file *filp, char __user *buffer, size_t ss, loff_t* lo) {

    // If readable, read the answer
    if (myini(DMAREADABLEADDR) == 1){
        int ans;
        // Read the answer from DMA
        ans = myini(DMAANSADDR);
        // Print a kernel message
        printk("%s:%s(): ans = %d\n", PREFIX_TITLE, __func__, ans);
        // Transfer the answer to the user space
        put_user(ans, (int *) buffer);
        // Clean the result
        myouti(0, DMAANSADDR);
        // Set the DMA to be unreadable
        myouti(0, DMAREADABLEADDR);
        return 0;
    }
    else {
        printk("%s:%s(): DMA not readable\n", PREFIX_TITLE, __func__);
        return -1;
    }

}
```

This function reads the answer of the device from DMA to the user space.

At the beginning, it checks if the DMA is currently readable. If so, it reads the answer from the DMA. Then, it transfers the answer to the user space. Lastly, it cleans the result in DMA and sets the DMA to be unreadable.

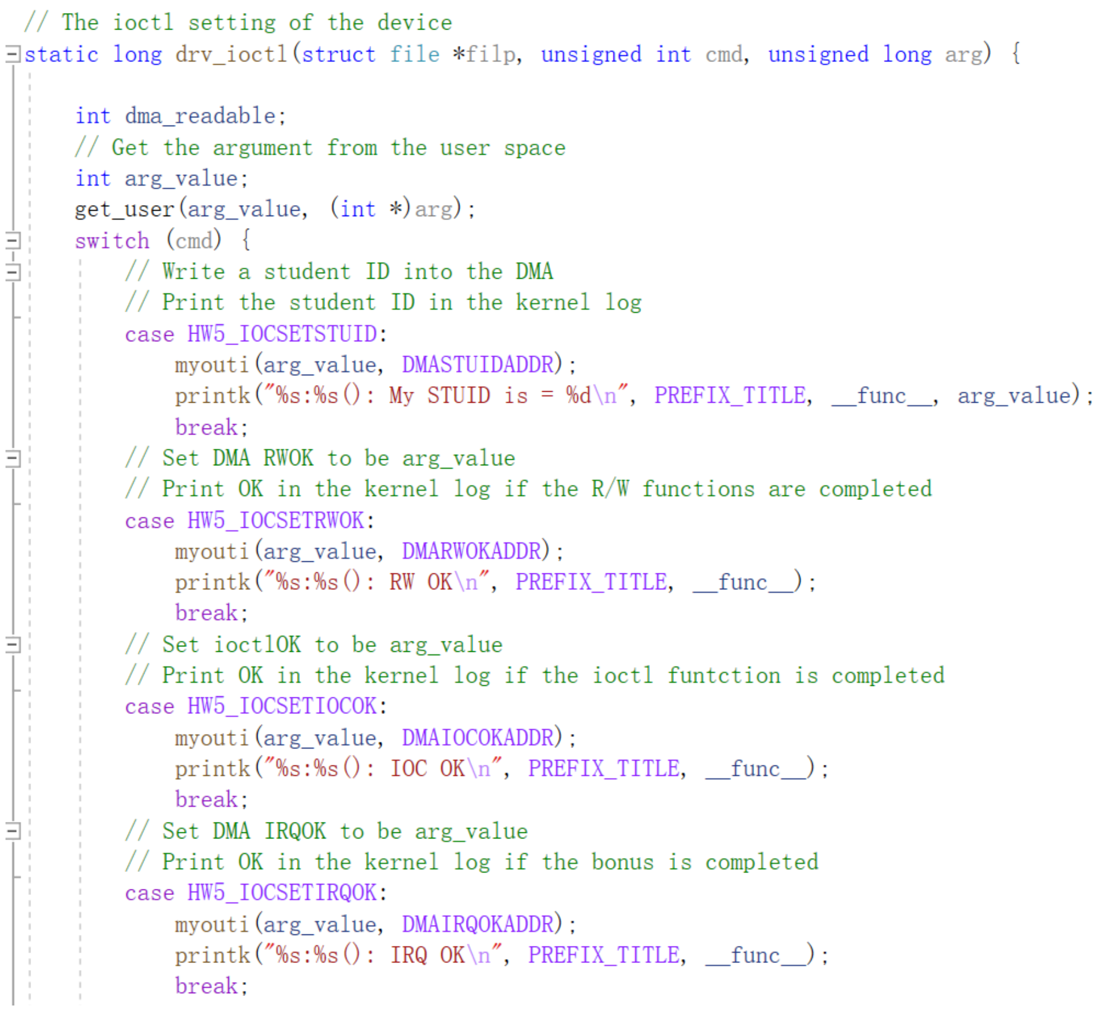
b. static ssize_t drv_write(struct file *filp, const char __user *buffer, size_t ss, loff_t* lo);



```
// The write operation of the device
static ssize_t drv_write(struct file *filp, const char __user *buffer, size_t ss, loff_t* lo)
{
    // Get the input data from the user space
    struct DataIn data;
    get_user(data.a, (char*) buffer);
    get_user(data.b, (int*) buffer+1);
    get_user(data.c, (int*) buffer+2);

    // Write the input data into the DMA
    myoutc(data.a, DMAOPCODEADDR);
    myouti(data.b, DMAOPERANDBADDR);
    myouti(data.c, DMAOPERANDCADDR);

    // Set the DMA to be unreadable
    myouti(0, DMAREADABLEADDR);

    // Initialize the work routine
    INIT_WORK(work_routine, drv_arithmetic_routine);

    // Blocking IO Step 1: Put the work into the system work queue
    // Non-blocking IO: Just put the work into the system work queue
    printk("%s:%s(): queue work\n", PREFIX_TITLE, __func__);
    schedule_work(work_routine);

    // Blocking IO Step 2: Flush works in the work queue
    if (myini(DMABLOCKADDR)) {
        printk("%s:%s(): block\n", PREFIX_TITLE, __func__);
        // Flush works on the work queue
        // Force execution of the kernel-global workqueue and block until its completion
        flush_scheduled_work();
        // Set the DMA to be readable
        myouti(1, DMAREADABLEADDR);
    }

    return 0;
}
```

This function writes the input data (operator and operands) from the user space into the DMA. At the beginning, it gets the input data from the user space and then write them into the DMA. Next, it initializes the work routine of the device using INIT_WORK(), sets the DMA to be unreadable to indicate that the answer (output) is not ready, and puts the work into the work queue using schedule_work(). If the write mode is blocking I/O, it flushes the works in the work queue, forcing execution of the kernel-global work queue and blocking until its completion.

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

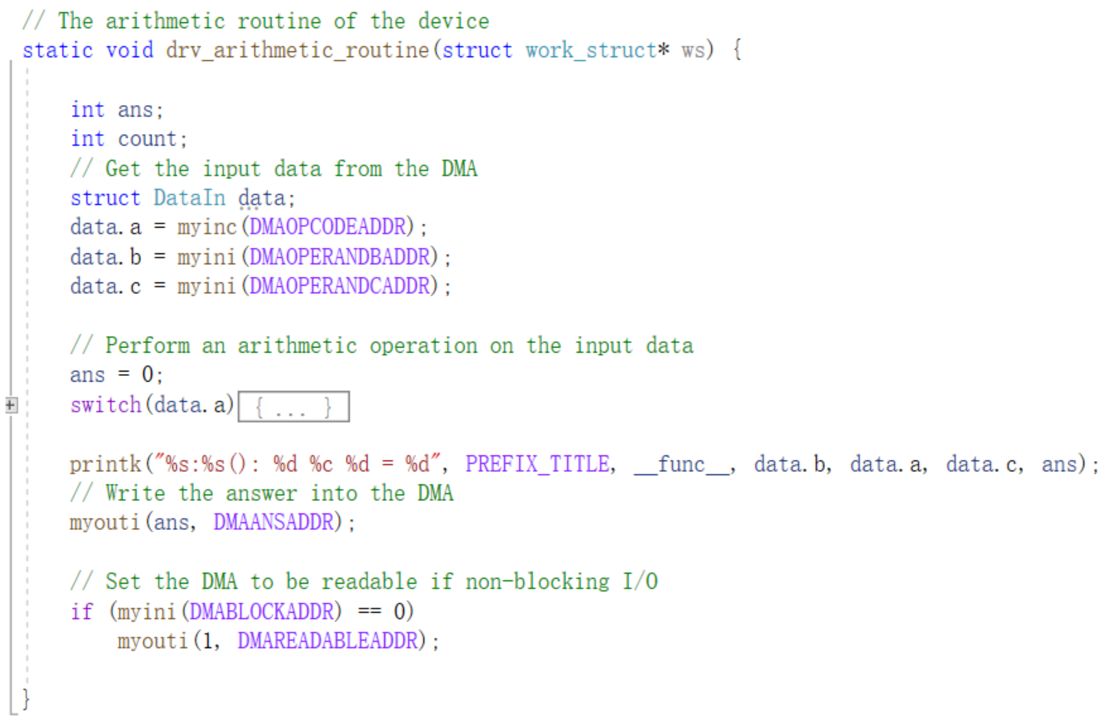


```
// The ioctl setting of the device
static long drv_ioctl(struct file *filp, unsigned int cmd, unsigned long arg) {

    int dma_readable;
    // Get the argument from the user space
    int arg_value;
    get_user(arg_value, (int *)arg);
    switch (cmd) {
        // Write a student ID into the DMA
        // Print the student ID in the kernel log
        case HW5_IOCSETSTUID:
            myouti(arg_value, DMASTUIDADDR);
            printk("%s:%s(): My STUID is = %d\n", PREFIX_TITLE, __func__, arg_value);
            break;
        // Set DMA RWOK to be arg_value
        // Print OK in the kernel log if the R/W functions are completed
        case HW5_IOCSETRWOK:
            myouti(arg_value, DMARWOKADDR);
            printk("%s:%s(): RW OK\n", PREFIX_TITLE, __func__);
            break;
        // Set ioctlOK to be arg_value
        // Print OK in the kernel log if the ioctl funtction is completed
        case HW5_IOCSETIOCOK:
            myouti(arg_value, DMAIOCOKADDR);
            printk("%s:%s(): IOC OK\n", PREFIX_TITLE, __func__);
            break;
        // Set DMA IRQOK to be arg_value
        // Print OK in the kernel log if the bonus is completed
        case HW5_IOCSETIRQOK:
            myouti(arg_value, DMAIRQOKADDR);
            printk("%s:%s(): IRQ OK\n", PREFIX_TITLE, __func__);
            break;
        // Set the write mode to be arg_value
        case HW5_IOCSETBLOCK:
            myouti(arg_value, DMABLOCKADDR);
            if (arg_value == 0)
                printk("%s:%s(): Non-blocking IO\n", PREFIX_TITLE, __func__);
            else if (arg_value == 1)
                printk("%s:%s(): Blocking IO\n", PREFIX_TITLE, __func__);
            else{
                printk("%s:%s(): invalid write function mode: %d\n", PREFIX_TITLE, __func__, arg_value);
                return -1;
            }
            break;
        // Wait until the DMA is readable
        // Used before a read operation when using non-blocking write mode
        case HW5_IOCWAITREADABLE:
            // Check if the DMA is readable
            dma_readable = myini(DMAREADABLEADDR);
            while (dma_readable != 1) {
                // Sleep for some time
                msleep(SLEEP_TIME);
                // Check if the DMA is readable again
                dma_readable = myini(DMAREADABLEADDR);
            }
            // Write DMA readable into the user space
            put_user(dma_readable, (int *)arg);
            printk("%s:%s(): wait readable 1\n", PREFIX_TITLE, __func__);
            break;
        // Invalid command
        default:
            printk("%s:%s(): invalid command: %d\n", PREFIX_TITLE, __func__, cmd);
            return -1;
    }
    return 0;

}
```

This function handles the I/O control settings of the device. It firstly gets the argument from the user space. Then, according to the command code, it does one of the following 6 operations: HW5_IOCSETSTUID, HW5_IOCSETRWOK, HW5_IOCSETIOCOK, HW5_IOCSETIRQOK, HW5_IOCSETBLOCK, and HW5_IOCWAITREADABLE. HW5_IOCSETSTUID is writing a student ID into the DMA; HW5_IOCSETRWOK is setting DMA RWOK to be the value of the argument; HW5_IOCSETIOCOK is setting iocltOK to be the value of the argument; HW5_IOCSETIRQOK is setting DMA IRQOK to be the value of the argument; HW5_IOCSETBLOCK is setting the write mode to be the value of the argument; HW5_IOCWAITREADABLE is waiting until the

DMA is readable and writing DMA readable into the user space, which is used before a read operation when using non-blocking write mode.

d. drv_arithmetic_routine(struct work_struct* ws);



```
// The arithmetic routine of the device
static void drv_arithmetic_routine(struct work_struct* ws) {

    int ans;
    int count;
    // Get the input data from the DMA
    struct DataIn data;
    data.a = myinc(DMAOPCODEADDR);
    data.b = myini(DMAOPERANDBADDR);
    data.c = myini(DMAOPERANDCADDR);

    // Perform an arithmetic operation on the input data
    ans = 0;
    switch(data.a){ ... }

    printk("%s:%s(): %d %c %d = %d", PREFIX_TITLE, __func__, data.b, data.a, data.c, ans);
    // Write the answer into the DMA
    myouti(ans, DMAANSADDR);

    // Set the DMA to be readable if non-blocking I/O
    if (myini(DMABLOCKADDR) == 0)
        myouti(1, DMAREADABLEADDR);

}
```

This function is the arithmetic routine of the device. It firstly reads the input data from the DMA. Then, it performs an arithmetic operation on the input data. The arithmetic operation is one of addition, subtraction, multiplication, division, and prime (the cth prime number greater than b), which is determined by the input operand. When the calculation is done, it writes the answer into the DMA and sets the DMA to be readable when using non-blocking I/O to indicate that the answer (output) is ready.

e. static int __init init_modules(void);

This function initializes the kernel module.

```
/* Register chrdev */
// Allocate a range of char device numbers
ret = alloc_chrdev_region(&dev, DEV_BASEMINOR, DEV_COUNT, DEV_NAME);
// If alloc_chrdev_region() returns an error
if (ret) {
    printk("%s:%s(): cannot alloc chrdev\n", PREFIX_TITLE, __func__);
    return ret;
}
// Get the major number
dev_major = MAJOR(dev);
// Get the first minor number
dev_minor = MINOR(dev);
printk("%s:%s(): register chrdev(%d, %d)\n", PREFIX_TITLE, __func__, dev_major, dev_minor);
```

The first step is to register the character device. It calls alloc_chrdev_region() to allocate a range of character device numbers. The major number is allocated dynamically by the kernel, the minor number is specified by DEV_BASEMINOR and DEV_COUNT, and the device name is specified by DEV_NAME. Then, it calls MAJOR() and MINOR() to get the major number and the first minor number.

```
/* Init cdev and make it alive */
// Allocate a cdev structure
dev_cdevp = cdev_alloc();
// Initialize the cdev, remembering fops
cdev_init(dev_cdevp, &fops);
dev_cdevp->owner = THIS_MODULE;
// Add the device to the system, making it alive immediately
ret = cdev_add(dev_cdevp, MKDEV(dev_major, dev_minor), DEV_COUNT);
// If cdev_add() returns an error
if (ret) {
    printk("%s:%s(): add chrdev failed\n", PREFIX_TITLE, __func__);
    return ret;
}
```

The second step is to initialize the character device and make it alive. It first calls cdev_alloc() to allocate a cdev structure and cdev_init() to initialize the cdev with the defined file operations. Then, it calls cdev_add()

to add the device to the system, making it alive immediately.

```
/* Allocate DMA buffer */
printk("%s:%s(): allocate DMA buffer\n", PREFIX_TITLE, __func__);
// Use kzalloc() to allocate and zero-set memory
dma_buf = kzalloc(DMA_BUFSIZE, GFP_KERNEL);
```

The third step is to allocate the DMA buffer. It calls kzalloc() to allocate and zero-set memory for the DMA buffer.

```
/* Allocate work routine */
work_routine = kzalloc(sizeof(typeof(*work_routine)), GFP_KERNEL);
```

The last step is to allocate the work routine pointer. It again calls kzalloc() to allocate and zero-set memory for the work routine pointer.

f. static void __exit exit_modules(void);

This function exits the kernel module.

```
/* Free DMA buffer when exit modules */
kfree(dma_buf);
printk("%s:%s(): free DMA buffer\n", PREFIX_TITLE, __func__);
```

The first step is to free the DMA buffer by kfree().

```
/* Delete character device */
unregister_chrdev_region(MKDEV(dev_major, dev_minor), DEV_COUNT);
cdev_del(dev_cdevp);
```

The second step is to delete the character device. It first deallocates the device numbers by unregister_chrdev_region() and then deletes the character device by cdev_del().

```
/* Free work routine */
kfree(work_routine);
printk("%s:%s(): unregister chrdev\n", PREFIX_TITLE, __func__);
```

The last step is to free the work routine pointer by kfree().

3.3. The Bonus Program (Included in Source)

3.3.1. Global Variables

a. const int IRQ_NUM = 1;

The interrupt request number corresponding to the keyboard.

b. static int irq_count = 0;

The count of keyboard interrupt requests.

3.3.2. Utility Functions

static irqreturn_t count_irq(int irq, void* dev_id);

```
// Count the interrupt request if it is from the keyboard
// typedef irqreturn_t (*irq_handler_t)(int, void *);
static irqreturn_t count_irq(int irq, void* dev_id) {
    irq_count++;
    // return type: (*irq_handler_t)(int, void *)
    // IRQ_NONE = (0 << 0)
    // IRQ_HANDLED = (1 << 0)
    // IRQ_WAKE_THREAD = (1 << 1)
    return IRQ_NONE;
}
```

Count the interrupt request if it is from the keyboard. By the definition of irqreturn_t, it is equivalent to the type (*irq_handler_t)(int, void*) or irq_handler_t. Therefore, this function can be passed as the second argument of the function request_irq(), which is of type irq_handler_t. In the end, this function returns IRQ_NONE.

3.3.3. Major Functions

a. static int __init init_modules(void);

```
/* Register interrupt service routine */
ret = request_irq(IRQ_NUM, count_irq, IRQF_SHARED, IRQ_DEV_NAME, (void*)&IRQ_NUM);
if (ret) {
    printk("%s:%s(): cannot request irq\n", PREFIX_TITLE, __func__);
    return ret;
}
printk("%s:%s(): request_irq %d return %d\n", PREFIX_TITLE, __func__, IRQ_NUM, ret);
```

When initializing the kernel module, the interrupt service routine to count IRQ is registered. It calls request_irq() to register the interrupt service routine, where IRQ_NUM is the interrupt request number, count_irq is the function corresponding to the interrupt service routine, IRQF_SHARED is the flag, IRQ_DEV_NAME is the device name, and (void*)&IRQ_NUM is a cookie passed back to the handler function.

b. static void __exit exit_modules(void);

```
/* Free IRQ */
free_irq(IRQ_NUM, (void*)&IRQ_NUM);
printk("%s:%s(): interrupt count = %d\n", PREFIX_TITLE, __func__, irq_count);
```

When exiting the kernel module, the interrupt service routine is freed by free_irq().

4. What Problems I Met in This Assignment and What Are My Solution?

a. How to improve the efficiency of the prime-check function is_prime()?

The naïve method to check whether an integer n greater than 3 is prime is to divide it by every integer from 3 to it. There are approximately n divisions. In fact, it is sufficient to divide it by every odd integer from 3 to a half of it. The number of division operations can be reduced to about n/4. Furthermore, we can divide the integer by only every odd integer from 3 to the square root of it. This leads to only around sqrt(n)/2 divisions. If the C Standard Library can be used, we can use sqrt() from <math.h> to calculate the square root.

b. How to implement blocking I/O and non-blocking I/O in the drv_write() function?

In both blocking I/O and non-blocking I/O, the write function should set the DMA to be unreadable to indicate that the answer is not ready, initialize the work routine by INIT_WORK(), and then put the work into the system work queue by schedule_work(). For non-blocking I/O, the write operation is done. For blocking I/O, however, the works in the work queue must be flushed immediately by flush_scheduled_work().

c. How to wait the DMA to be readable? That is, how to implement HW5_IOCWAITREADABLE?

First, check if the DMA is currently readable. If not, make the process to sleep for a while using msleep() and check if the DMA is readable again.

Repeat the above procedure until DMA is readable. Then write the DMA readable status (should be 1) into the user space at the address specified by the argument.

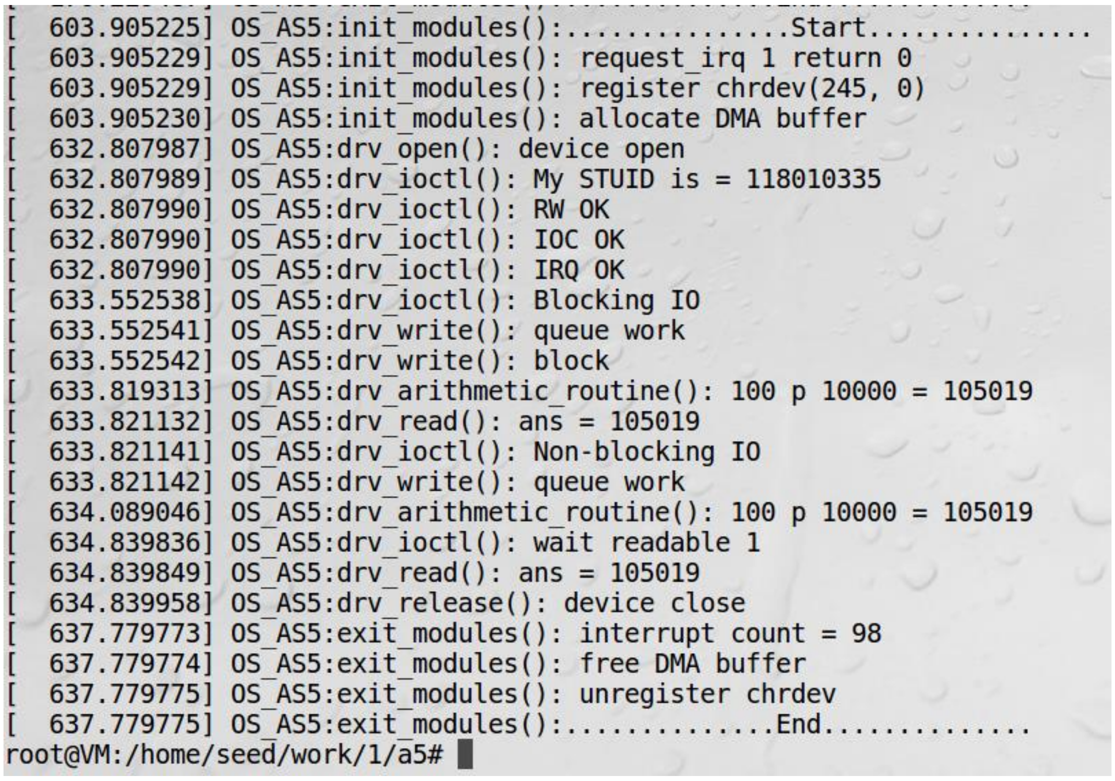
d. In Bonus, how to implement the function that counts the number of interrupt requests?

According to the definition of the function request_irq(), the handler function to be called when the interrupt request occurs must be of type irq_handler_t. Furthermore, according to the type definition of irqreturn_t, irqreturn_t is equivalent to (*irq_handler_t) (int, void*). This indicates that the handler function should take two parameters, which are of type int and void*, and return a value of type irqreturn_t. By the definition of the enumeration type irqreturn, the handler function can just return IRQ_NONE, which equals to 0 << 0.

e. When unregistering the character device, how to get the dev_t to unregister without making the dev_t a global variable?

We can use MKDEV(major, minor) to get the dev_t corresponding to a given combination of a major number and a base minor number.

5. Screenshot of My Program Output.



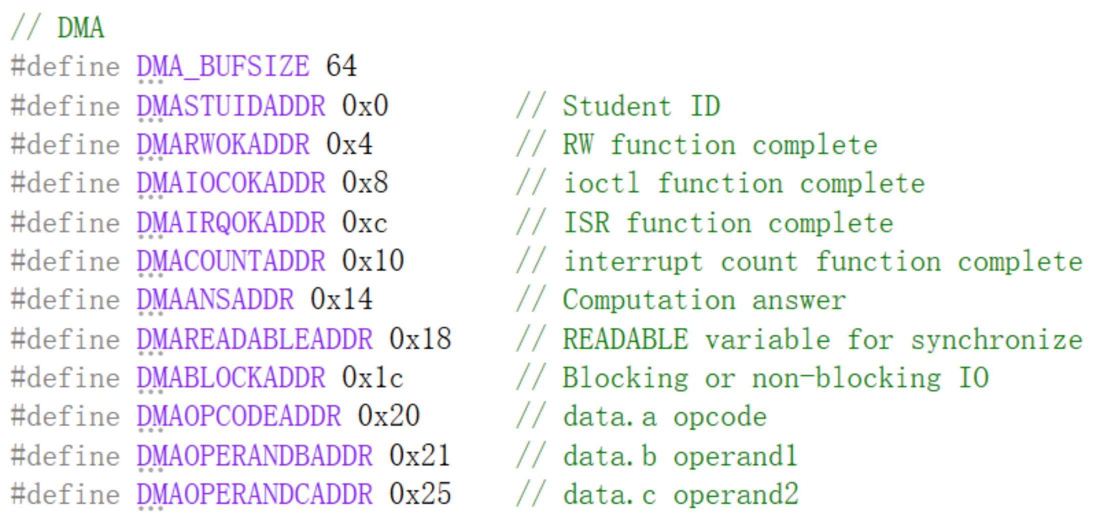
```
[  603.905225] OS_AS5:init_modules():..............Start..............
[  603.905229] OS_AS5:init_modules(): request_irq 1 return 0
[  603.905229] OS_AS5:init_modules(): register chrdev(245, 0)
[  603.905230] OS_AS5:init_modules(): allocate DMA buffer
[  632.807987] OS_AS5:drv_open(): device open
[  632.807989] OS_AS5:drv_ioctl(): My STUID is = 118010335
[  632.807990] OS_AS5:drv_ioctl(): RW OK
[  632.807990] OS_AS5:drv_ioctl(): IOC OK
[  632.807990] OS_AS5:drv_ioctl(): IRQ OK
[  633.552538] OS_AS5:drv_ioctl(): Blocking IO
[  633.552541] OS_AS5:drv_write(): queue work
[  633.552542] OS_AS5:drv_write(): block
[  633.819313] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[  633.821132] OS_AS5:drv_read(): ans = 105019
[  633.821141] OS_AS5:drv_ioctl(): Non-blocking IO
[  633.821142] OS_AS5:drv_write(): queue work
[  634.089046] OS_AS5:drv_arithmetic_routine(): 100 p 10000 = 105019
[  634.839836] OS_AS5:drv_ioctl(): wait readable 1
[  634.839849] OS_AS5:drv_read(): ans = 105019
[  634.839958] OS_AS5:drv_release(): device close
[  637.779773] OS_AS5:exit_modules(): interrupt count = 98
[  637.779774] OS_AS5:exit_modules(): free DMA buffer
[  637.779775] OS_AS5:exit_modules(): unregister chrdev
[  637.779775] OS_AS5:exit_modules():.............End.............
root@VM:/home/seed/work/1/a5# 
```

6. What Did I Learn from This Assignment?

a. What is DMA?

Direct Memory Access (DMA) is a feature of computer systems that allows certain hardware subsystems to access main system memory (RAM) independent of the Central Processing Unit (CPU).

In this assignment, the character device can access the main memory directly through DMA. To simulate the registers and memory on the device, I allocate a DMA buffer in the kernel memory space. This buffer is as I/O port mapping in main memory. The values stored in the DMA buffer are as follows:



```
// DMA
#define DMA_BUFSIZE 64
#define DMASTUIDADDR 0x0        // Student ID
#define DMARWOKADDR 0x4         // RW function complete
#define DMAIOCOKADDR 0x8        // ioctl function complete
#define DMAIRQOKADDR 0xc        // ISR function complete
#define DMACOUNTADDR 0x10       // interrupt count function complete
#define DMAANSADDR 0x14         // Computation answer
#define DMAREADABLEADDR 0x18    // READABLE variable for synchronize
#define DMABLOCKADDR 0x1c       // Blocking or non-blocking IO
#define DMAOPCODEADDR 0x20      // data.a opcode
#define DMAOPERANDBADDR 0x21    // data.b operand1
#define DMAOPERANDCADDR 0x25    // data.c operand2
```

b. What is blocking I/O and what is non-blocking I/O?

The write function has two modes, blocking and non-blocking. When using blocking I/O, the write function must wait until the computation is completed and then return. That is, the write function queues the work into the work queue and then flushes the work queue, forcing execution of all the works in the queue. When using non-blocking I/O, the write function just returns after queueing the work into the system work queue.

c. What are the three types of devices?

Linux Device Drivers include character devices (such as printers, terminals, and mice), block devices (including all disk drives), and network interface devices.

Block devices include all devices that allow random access to completely independent, fixed-sized blocks of data, including hard disks and floppy disks, CD-ROMs and Blu-ray discs, and flash memory. Block devices are typically used to store file systems, but direct access to a block device is also allowedso that programs can create and repair the file system that the device contains. Applications can also access these block devices directly if they wish. Forexample, a database application may prefer to perform its own fine-tuned layout of data onto a disk rather than using the general-purpose file system.

Character devices include most other devices, such as mice and keyboards. The fundamental difference between block and character devices is random access—block devices are accessed randomly, while character devices are accessed serially byte by byte. For example, seeking to a certain position in a file might be supported for a DVD but makes no sense for a pointing device such as a mouse.

Network devices are dealt with differently from block and character devices. Users cannot directly transfer data to network devices. Instead, they must communicate indirectly by opening a connection to the kernel's networking subsystem.

d. How to create a filesystem node?

int mknod(const char *pathname, mode_t mode, dev_t dev);

The system call mknod() creates a filesystem node (file, device special file, or named pipe) named pathname, with attributes specified by mode and dev.

e. What is the major number and what is the minor number?

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. We can get available major and minor number by:

Create a kernel module

Use alloc_chrdev_region() in module_init() function

Call MAJOR() and MINOR().

f. What is the workqueue in Linux?

There are many cases where an asynchronous process execution context is needed and the workqueue (wq) API is the most commonly used mechanism for such cases.

When such an asynchronous execution context is needed, a work item describing which function to execute is put on a queue. An independent thread serves as the asynchronous execution context. The queue is called workqueue and the thread is called worker.

While there are work items on the workqueue the worker executes the functions associated with the work items one after the other. When there is no work item left on the workqueue the worker becomes idle. When a new work item gets queued, the worker begins executing again.