CSC3150 Assignment 4

CHEN Ang (*118010009*)

In this assignment, we are asked to create a device in Linux that computes simple arithmetic expressions and desired prime numbers. Supporting driver functions for the device, including I/O control and R/W are implemented in a kernel module. Kernel memory is allocated to simulate registers on the device.

How to Run the Program

The following commands assume the user already has `sudo` permissions.

Inside the `source` folder, type in the terminal

```
make
```

to compile the device driver module, the test program, and insert the module `mydev.ko` into the kernel. The terminal should then display part of the kernel log. Find the last line which reads

```
OS_AS5:init_modules(): registering chrdev([MAJOR],[MINOR])
```

To build the file node, type in the terminal

```
./mkdev.sh [MAJOR] [MINOR]
```

By then you can use the test program to test the device. Type in the terminal

```
./test
```

to start testing. After the test, you can type in

```
make clean
```

to remove `mydev.ko` from the kernel and have a screenshot of the kernel log.

Finally to remove the file node, type in

```
./rmdev.sh
```

Program Design

Module Initialization (`init_modules`)

Upon initialization, The kernel module first asks the system to reserve a range of device numbers by invoking `alloc_chrdev_region(dev)`. It then obtains a set of major and minor device numbers using `MAJOR(dev)` and `MINOR(dev)` macros, and `printk` the obtained device numbers to the kernel logs.

For the device to work, we need to initialize a `cdev` `struct`:

```
dev_cdev = cdev_alloc();
dev_cdev->ops = &fops;
dev_cdev->owner = THIS_MODULE;
```

Here `fops` is a `file_operations` `struct` to bind user-end file operations with kernel-end functions implemented:

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

We can then make the device live, using

```
cdev_add(dev_cdev, dev, 1)
```

For later work, we allocate a Direct Memory Access (DMA) buffer `dma_buf` to simulate the registers on a physical device. We also allocate enough space for the work routine, both using `kmalloc` at the end of the module initialization.

Read Operation (`drv_read`)

The `read` operation is bound with the `drv_read` kernel function. The implementation of `drv_write` is rather straightforward. One thing to note is that since data needs to be transferred between user and kernel space, the `put_user` function is invoked to place the answer of the expression at `dma_buf[DMAANSADDR]` to the user-end `buffer`. After this the answer in `dma_buf` is cleared with zero bits and the readability status `dma_buf[DMAREADBLEADDR]` is set to unreadable (`0`).

Write Operation (`drv_write`)

The `write` operation is bound with the `drv_write` kernel function. Our device only handles arithmetic expressions, and we assume the input `buffer` from the user is organized in the following way:

```
typedef struct expr {
    char a;
    int b;
    short c;
} expr_t;
```

The expression is then copied to the corresponding locations in `dma_buf`:

```
get_user( *(char *)(dma_buf+DMAOPCODEADDR), (char*)buffer);
get_user( *(int  *)(dma_buf+DMAOPERANDBADDR), &(data->b) );
get_user( *(short*)(dma_buf+DMAOPERANDCADDR), &(data->c) );
```

Until then, we are ready to enqueue the `drv_arithmetic_routine` to the `work_q` with

```
INIT_WORK(work_q, drv_arithmetic_routine);
schedule_work(work_q);
```

Now is a critical moment with regard to the behavior of `drv_write`, namely, either it uses blocking I/O or non-blocking one. If the user has chosen blocking I/O, `drv_write` needs to wait for the work in the `work_q` to finish before proceeding. However if the user has chosen non-blocking I/O, `drv_write` immediately returns. The `flush_scheduled_work()` function can thus be of our use:

```
// BLOCKING I/O: Wait until all work flushed
if (myini(DMABLOCKADDR) == 1) {
    printk("%s:%s(): blocking\n", PREFIX_TITLE, __func__);
    flush_scheduled_work();
}
return 0;
```

I/O Control (`drv_ioctl`)

The user uses `ioctl` to control the I/O behavior of the device. In particular, the user can pass different command to either set entries in `dma_buf` (e.g., to set non/blocking IO with `HW5_IOCSETBLOCK`), or wait for the device to respond with an interrupt signal when the answer is readable (`HW_IOCWAITREADABLE`). This behavior is implemented with an indefinite loop:

```
while( myini(DMAREADABLEADDR) != 1 )
    msleep(1000);
```

After the sleep is finished, we send the signal of `READABLE` to user by

```
put_user(1, (unsigned int*)arg)
```

Arithmetic Routine (`drv_arithmetic_routine`)

In `drv_arithmetic_routine`, the module evaluates the expression stored in `dma_buf` and places the answer at `dma_buf[DMAANSADDR]`. Since the answer is then ready to read, we also need to set the readability status to `READABLE` (`1`) in case of non-blocking I/O.

Module Exit (`exit_modules`)

When exiting, the module

- frees the IRQ via `free_irq` and `printk()` the interrupt count
- unregisters the device via `unregister_chrdev_region`
- deletes `cdev` structs by invoking `cdev_del`
- `kfree` the `dma_buf` and `work_routine`

Bonus

We first use

```
watch -n 1 cat /proc/interrupts
```

to find out the IRQ number for Keyboard is `1`.

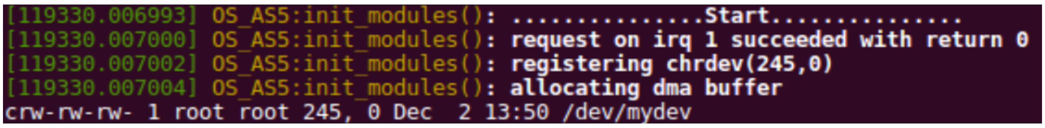
To count the number of interrupts from Keyboard, we make use of `request_irq` to add an interrupt handler named `irq_handler` which increments a global `count` every time we have an interrupt request from Keyboard. We remove the handler with `free_irq` and `printk` the interrupt number at module exit.

Problems I met

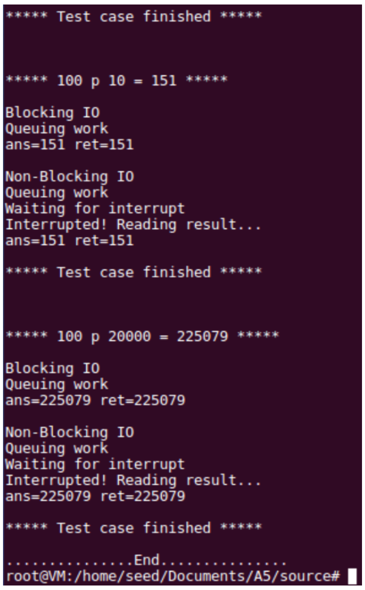
This project went smooth overall and most difficulties I met were minor, e.g., some typos in the code breaking the program which were fairly easy to debug.

Screenshots

- Kernel log after insertion of the driver module and creation of device file node



- Output of the test program to the user



- Kernel log during the execution of the test program

```
[119330.006993] OS_AS5:init_modules(): ...............Start...............
[119330.007000] OS_AS5:init_modules(): request on irq 1 succeeded with return 0
[119330.007002] OS_AS5:init_modules(): registering chrdev(245,0)
[119330.007004] OS_AS5:init_modules(): allocating dma buffer
[119330.698010] OS_AS5:drv_open(): device open
[119330.698017] OS_AS5:drv_ioctl(): STUID = 118010009
[119330.698018] OS_AS5:drv_ioctl(): RW OK
[119330.698019] OS_AS5:drv_ioctl(): IOC OK
[119330.698020] OS_AS5:drv_ioctl(): IRQ OK
[119330.698074] OS_AS5:drv_ioctl(): Blocking IO
[119330.698085] OS_AS5:drv_write(): work enqueued
[119330.698086] OS_AS5:drv_write(): blocking
[119330.698090] OS_AS5:drv_arithmetic_routine(): 100 + 10 = 110
[119330.698095] OS_AS5:drv_read(): ans = 110
[119330.698114] OS_AS5:drv_ioctl(): Non-Blocking IO
[119330.698123] OS_AS5:drv_write(): work enqueued
[119330.698125] OS_AS5:drv_arithmetic_routine(): 100 + 10 = 110
[119330.698134] OS_AS5:drv_ioctl(): readable now, sending interrupt to user
[119330.698143] OS_AS5:drv_read(): ans = 110
```

- Kernel log after removal of the kernel module.

```
[119339.399915] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225079
[119339.406387] OS_AS5:drv_read(): ans = 225079
[119339.406400] OS_AS5:drv_ioctl(): Non-Blocking IO
[119339.406402] OS_AS5:drv_write(): work enqueued
[119342.959442] OS_AS5:drv_arithmetic_routine(): 100 p 20000 = 225079
[119342.961171] OS_AS5:drv_ioctl(): readable now, sending interrupt to user
[119342.961197] OS_AS5:drv_read(): ans = 225079
[119342.961399] OS_AS5:drv_release(): device closed
[119552.721358] OS_AS5:exit_modules(): interrupt count = 77
[119552.721359] OS_AS5:exit_modules(): dma buffer freed
[119552.721362] OS_AS5:exit_modules(): chrdev unregistered
[119552.721362] OS_AS5:exit_modules(): ................End................
root@VM:/home/seed/Documents/A5/source#
```

What I learned

I learned the basic working principles of a device driver in Linux and how I can implement it on my own. I get to know the difference between blocking and non-blocking I/O modes. I learned what a DMA buffer is. I also learned how to add an interrupt handler for an IRQ number.