# Operating System (CSC 3150)

## Tutorial 11

KAI SHEN

SCHOOL OF SCIENCE AND ENGINEERING

E-MAIL: 118010254@LINK.CUHK.EDU.CN

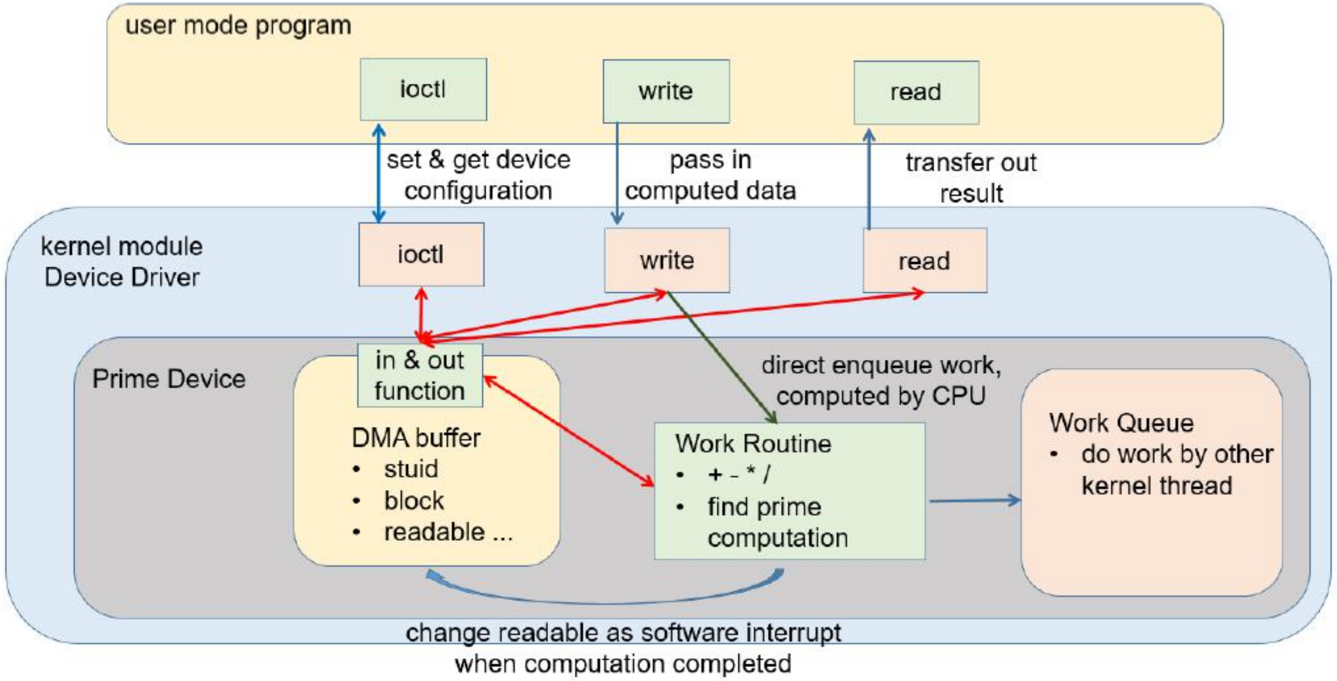
# Target

In this tutorial, we will discuss Assignment 5 related functions.

- Assignment 5 Structure
- DMA buffer
- ioctl
- arithmetic
- write (blocking / non-blocking)
- read (readable)
- work
- Assignment 5 makefile and scripts
- Bonus Hints
- Assignment 5 Problems

# Assignment 5 Structure

**Global View:**



# DMA buffer

- The data is stored in DMA buffer.
- Ports being defined:
  - #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

# DMA buffer

- When doing data transfer within kernel, you could use in/out function. “c/s/i” depends on what type of data you want to read or write.
- In and out function to write/read into/from DMA buffer. (Already defined in template)
  - void myoutc(unsigned char data,unsigned short int port)
    { *(volatile unsigned char*)(dma_buf+port) = data; }
  - void myouts(unsigned short data,unsigned short int port)
    { *(volatile unsigned short*)(dma_buf+port) = data;}
  - void myouti(unsigned int data,unsigned short int port)
    { *(volatile unsigned int*)(dma_buf+port) = data; }

Write data into DMA buffer with specific port.

  - unsigned char myinc(unsigned short int port)
    { return *(volatile unsigned char*)(dma_buf+port); }
  - unsigned short myins(unsigned short int port)
    { return *(volatile unsigned short*)(dma_buf+port); }
  - unsigned int myini(unsigned short int port)
    { return *(volatile unsigned int*)(dma_buf+port);}

Read data from DMA buffer with specific port.

# DMA buffer

- Demo usage of in and out function:
  - In user program, use ioctl to set I/O mode:
    ```
    int ret = 0;
    ioctl(fd, HW5_IOCSETBLOCK, &ret);
    ```
  - Transfer data from user to kernel:
    ```
    int value;
    get_user(value, (int *)arg);
    ```
  - Store I/O mode to DMA buffer in kernel:
    ```
    myouti(value, DMABLOCKADDR);
    ```
  - Get I/O mode from DMA buffer in kernel:
    ```
    int IOMode = myini(DMABLOCKADDR);
    ```

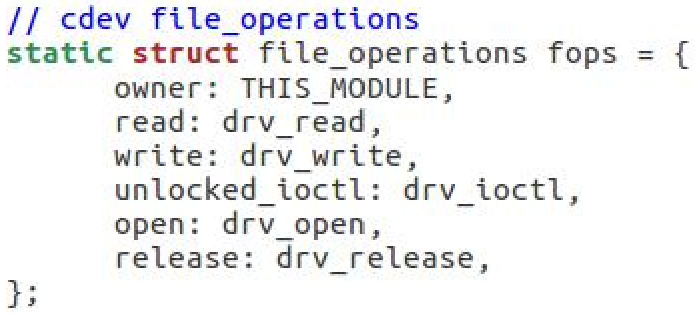
# ioctl

- Set and get device configuration

- Masked labels: (defined in “ioc_hw5.h”)
  - (HW5_IOC_SETSTUID) Set student ID: printk your student ID
  - (HW5_IOCSETRWOK) Set if RW OK: printk OK if you complete R/W function
  - (HW5_IOCSETIOCOK) Set if ioctl OK: printk OK if you complete ioctl function
  - (HW5_IOCSETIRQOK) Set if IRQ OK: printk OK if you complete bonus
  - (HW5_IOCSETBLOCK) Set blocking or non-blocking: set write function mode
  - (HW5_IOCWAITREADABLE) Wait if readable now (synchronize function): used before read to confirm it can read answer now when use non-blocking write mode.

# ioctl

- In user program, when ioctl is called, it will map to drv_ioctl in kernel. (If you've defined operation ioctl as drv_ioctl when adding the cdev.)



```
// 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,
};
```

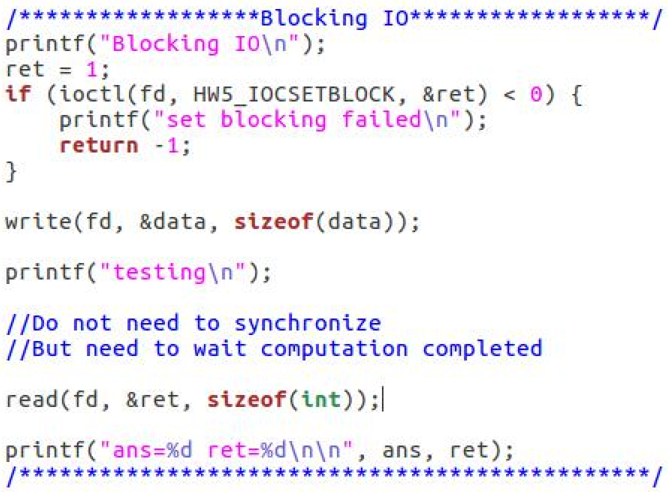
- In kernel, when you received different command from user program, you need to use the in and out function to change the configurations in DMA buffer.

# arithmetic

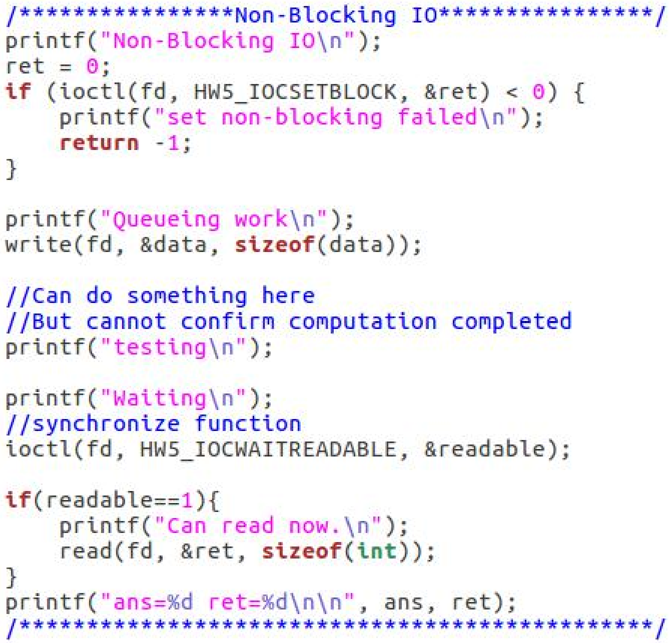
- In user program, use arithmetic function to trigger read and write.
- When write is triggered, it will transfer data into kernel. At the same time, it will put the arithmetic work into work queue.

# arithmetic

- For blocking write, the work queue will be forced to wait the termination of computation. So we could read the result when write completed.
- For non-blocking write, the work routine will be continued. So in user program, we use ioctl to check device's readable configuration before read the result. This is for synchronization and ensure you've read the correct answer.



```
/******************Blocking IO******************/
printf("Blocking IO\n");
ret = 1;
if (ioctl(fd, HW5_IOCSETBLOCK, &ret) < 0) {
    printf("set blocking failed\n");
    return -1;
}

write(fd, &data, sizeof(data));

printf("testing\n");

//Do not need to synchronize
//But need to wait computation completed

read(fd, &ret, sizeof(int));

printf("ans=%d ret=%d\n\n", ans, ret);
/************************************************/
```



```
/****************Non-Blocking IO****************/
printf("Non-Blocking IO\n");
ret = 0;
if (ioctl(fd, HW5_IOCSETBLOCK, &ret) < 0) {
    printf("set non-blocking failed\n");
    return -1;
}

printf("Queueing work\n");
write(fd, &data, sizeof(data));

//Can do something here
//But cannot confirm computation completed
printf("testing\n");

printf("Waiting\n");
//synchronize function
ioctl(fd, HW5_IOCWAITREADABLE, &readable);

if(readable==1){
    printf("Can read now.\n");
    read(fd, &ret, sizeof(int));
}
printf("ans=%d ret=%d\n\n", ans, ret);
/************************************************/
```

# write (blocking / non-blocking)

- In user program, write the data into your device.
- In kernel, transfer data into DMA buffer.
- Check the device I/O mode.
- Use “INIT_Work” to define what function (drv_arithmetic) should be executed in work routine.
- Basing on I/O mode (blocking or non-blocking), place the work into work queue.

# read (readable)

- In user program, use ioctl to check readable flag to synchronize the result when there is non-blocking write.
- In user program, read the computed result from device.
- In kernel, read the result from DMA buffer.
- Clean the result and set readable as false.
- Readable setting is checked via drv_ioctl. If it is non-readable, then just sleep and waiting.

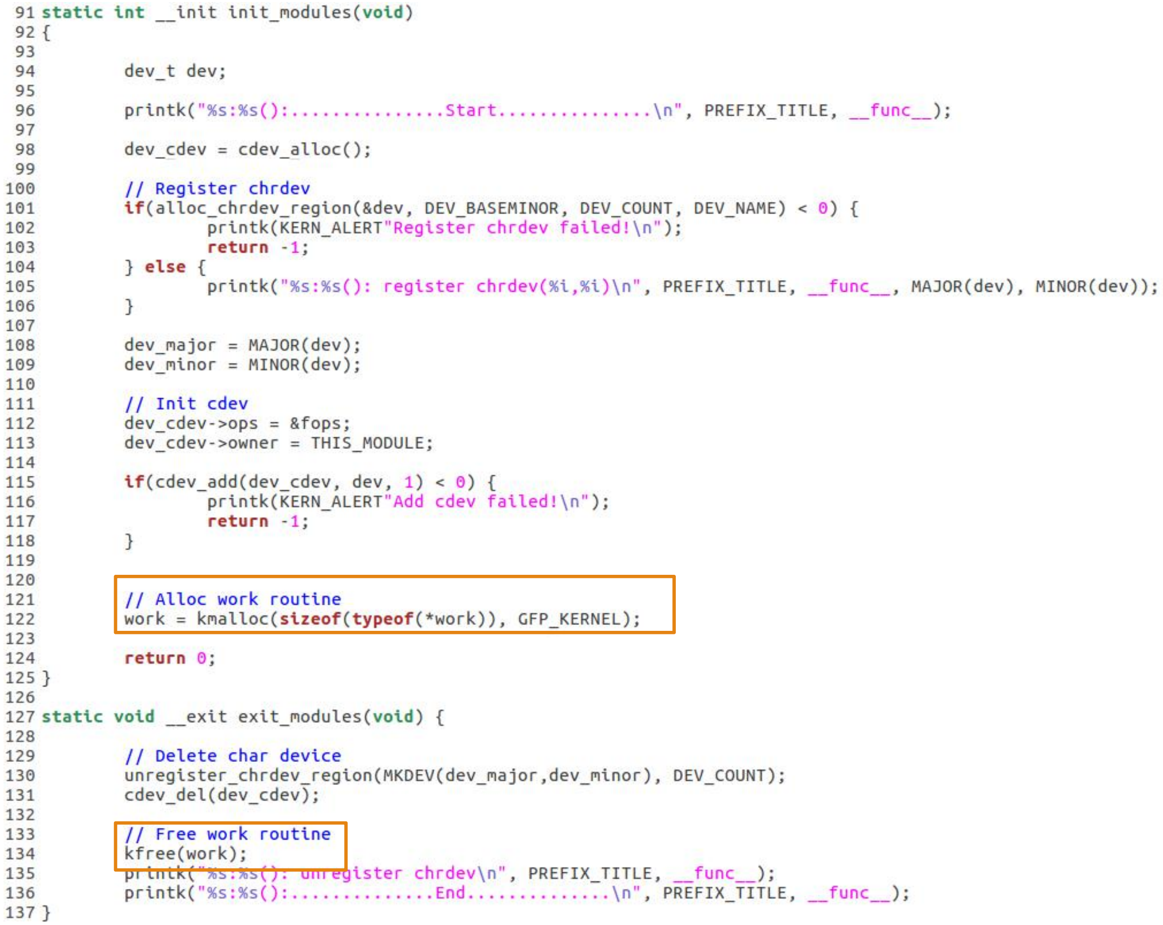
# drv_arithmetic

- In kernel, drv_arithmetic is to execute the computation when the work is in queue.
- It should complete ‘+’, ‘-’, ‘*’, ‘/’ and ‘p’ computations basing on the data being stored in DMA buffer.
- It should check blocking and non-blocking setting from DMA buffer.
- If the device is executing non-blocking write, set the readable as true when completed computation.

# work

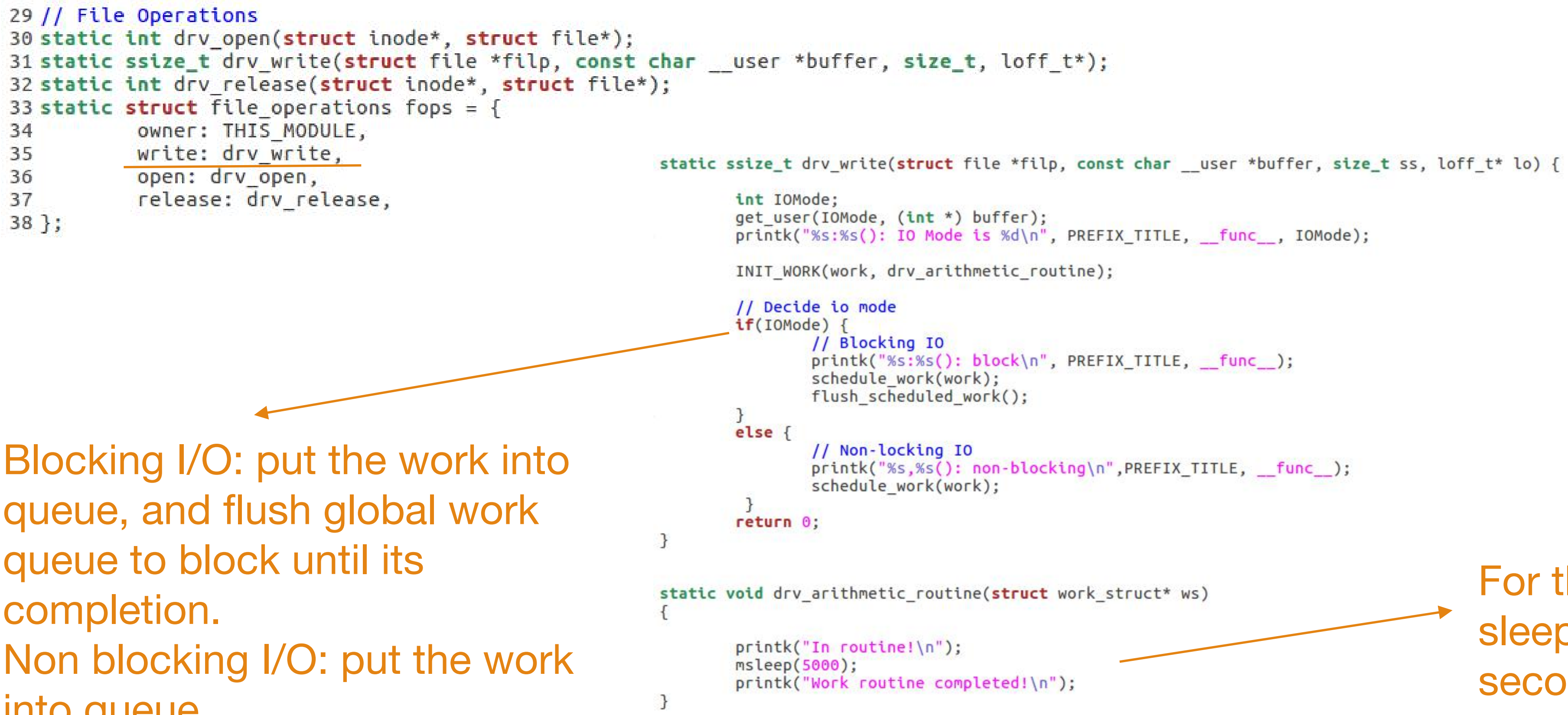
- Define a work: struct work_struct *work
- INIT_WORK(work, func): initialize work. Func is defined what function need to be executed for this work.
- schedule_work(work): put work task in global workqueue.
- flush_scheduled_work(): It is to flush the kernel-global work queue. Forces execution of the kernel-global workqueue and blocks until its completion.
- Use work to control block and non-blocking write.

# work example (kernel)



```c
static int __init init_modules(void)
{

        dev_t dev;

        printk("%s:%s():...............Start...............\n", PREFIX_TITLE, __func__);

        dev_cdev = cdev_alloc();

        // Register chrdev
        if(alloc_chrdev_region(&dev, DEV_BASEMINOR, DEV_COUNT, DEV_NAME) < 0) {
                printk(KERN_ALERT"Register chrdev failed!\n");
                return -1;
        } else {
                printk("%s:%s(): register chrdev(%i,%i)\n", PREFIX_TITLE, __func__, MAJOR(dev), MINOR(dev));
        }

        dev_major = MAJOR(dev);
        dev_minor = MINOR(dev);

        // Init cdev
        dev_cdev->ops = &fops;
        dev_cdev->owner = THIS_MODULE;

        if(cdev_add(dev_cdev, dev, 1) < 0) {
                printk(KERN_ALERT"Add cdev failed!\n");
                return -1;
        }


        // Alloc work routine
        work = kmalloc(sizeof(typeof(*work)), GFP_KERNEL);

        return 0;
}

static void __exit exit_modules(void) {

        // Delete char device
        unregister_chrdev_region(MKDEV(dev_major,dev_minor), DEV_COUNT);
        cdev_del(dev_cdev);

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

# work example (kernel)



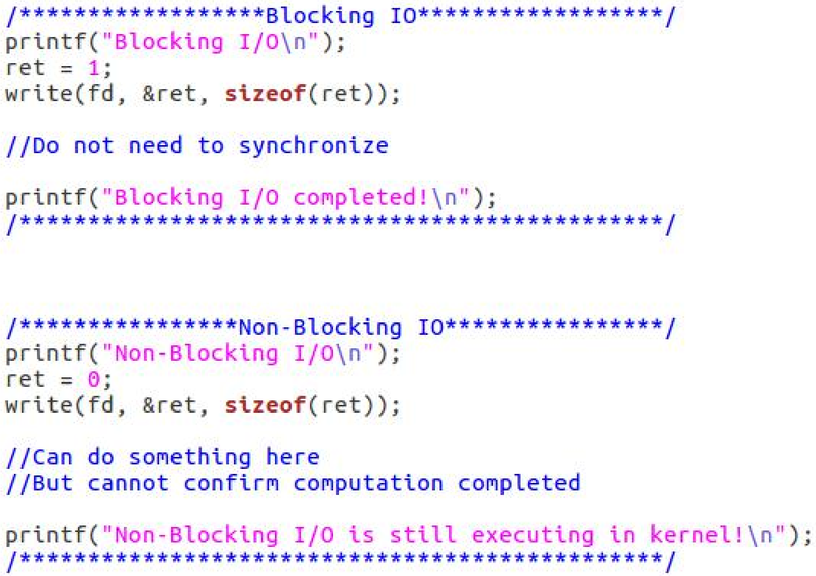
```c
// File Operations
static int drv_open(struct inode*, struct file*);
static ssize_t drv_write(struct file *filp, const char __user *buffer, size_t, loff_t*);
static int drv_release(struct inode*, struct file*);
static struct file_operations fops = {
        owner: THIS_MODULE,
        write: drv_write,
        open: drv_open,
        release: drv_release,
};
```

```c
static ssize_t drv_write(struct file *filp, const char __user *buffer, size_t ss, loff_t* lo) {

        int IOMode;
        get_user(IOMode, (int *) buffer);
        printk("%s:%s(): IO Mode is %d\n", PREFIX_TITLE, __func__, IOMode);

        INIT_WORK(work, drv_arithmetic_routine);

        // Decide io mode
        if(IOMode) {
                // Blocking IO
                printk("%s:%s(): block\n", PREFIX_TITLE, __func__);
                schedule_work(work);
                flush_scheduled_work();
        }
        else {
                // Non-locking IO
                printk("%s,%s(): non-blocking\n",PREFIX_TITLE, __func__);
                schedule_work(work);
         }
        return 0;
}


static void drv_arithmetic_routine(struct work_struct* ws)
{

        printk("In routine!\n");
        msleep(5000);
        printk("Work routine completed!\n");
}
```

Blocking I/O: put the work into queue, and flush global work queue to block until its completion.
Non blocking I/O: put the work into queue.

For this work, let it sleep for 5 seconds.

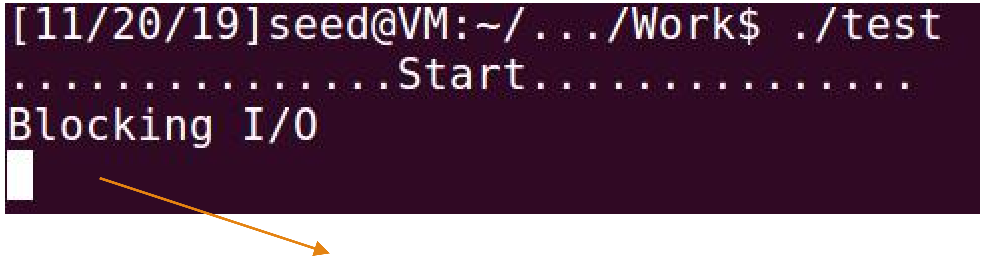
# work example (user program)



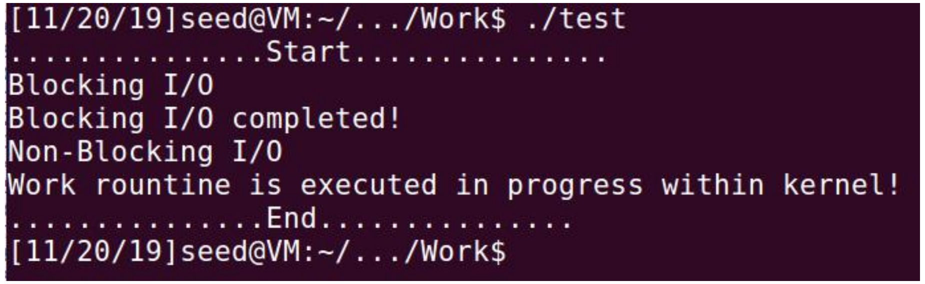
```
/******************Blocking IO******************/
printf("Blocking I/O\n");
ret = 1;
write(fd, &ret, sizeof(ret));

//Do not need to synchronize

printf("Blocking I/O completed!\n");
/***********************************************/


/****************Non-Blocking IO****************/
printf("Non-Blocking I/O\n");
ret = 0;
write(fd, &ret, sizeof(ret));

//Can do something here
//But cannot confirm computation completed

printf("Non-Blocking I/O is still executing in kernel!\n");
/***********************************************/
```

# work example (output)

User program output



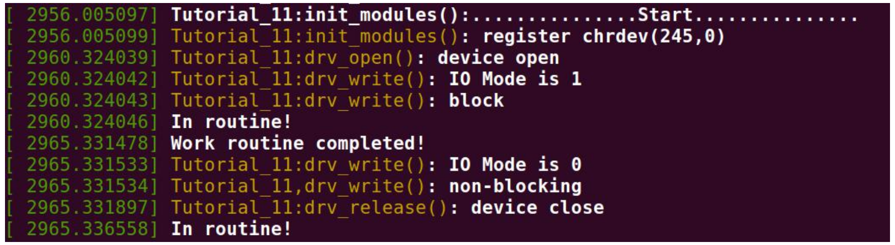
```
[11/20/19]seed@VM:~/.../Work$ ./test
...............Start...............
Blocking I/O
```



```
[11/20/19]seed@VM:~/.../Work$ ./test
...............Start...............
Blocking I/O
Blocking I/O completed!
Non-Blocking I/O
Work rountine is executed in progress within kernel!
...............End...............
[11/20/19]seed@VM:~/.../Work$
```

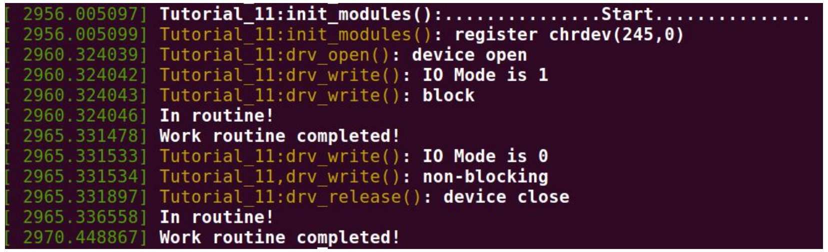
For blocking write, you will see it will not continue print message until the work is executed completely.

Kernel log output (display immediately when user program terminates)



```
[ 2956.005097] Tutorial_11:init_modules():...............Start...............
[ 2956.005099] Tutorial_11:init_modules(): register chrdev(245,0)
[ 2960.324039] Tutorial_11:drv_open(): device open
[ 2960.324042] Tutorial_11:drv_write(): IO Mode is 1
[ 2960.324043] Tutorial_11:drv_write(): block
[ 2960.324046] In routine!
[ 2965.331478] Work routine completed!
[ 2965.331533] Tutorial_11:drv_write(): IO Mode is 0
[ 2965.331534] Tutorial_11,drv_write(): non-blocking
[ 2965.331897] Tutorial_11:drv_release(): device close
[ 2965.336558] In routine!
```

Kernel log output (display 5 seconds later)



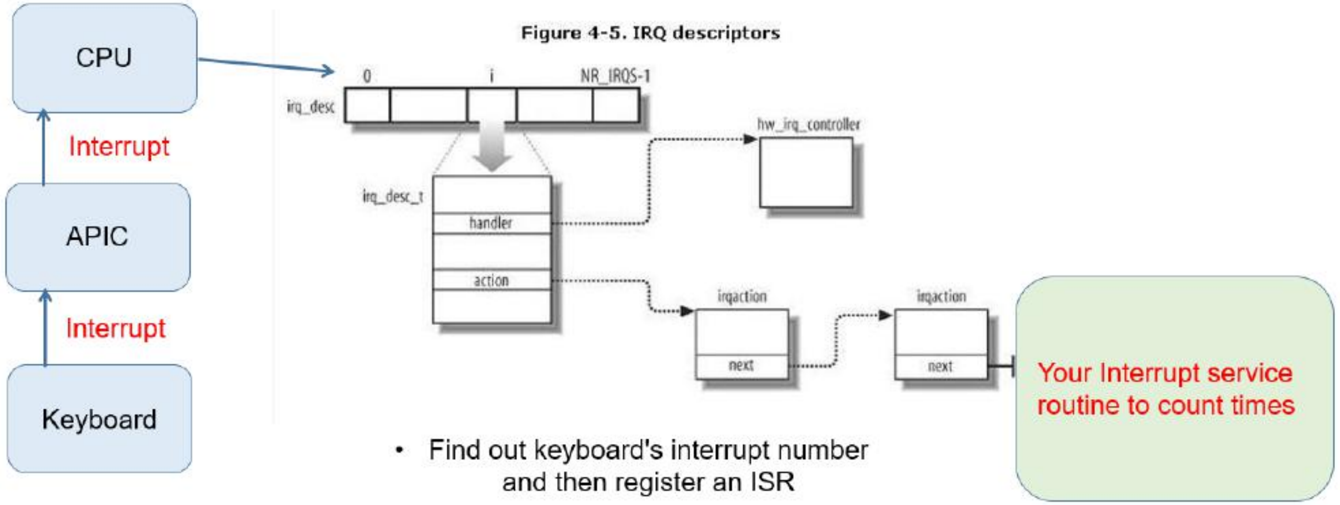
```
[ 2956.005097] Tutorial_11:init_modules():...............Start...............
[ 2956.005099] Tutorial_11:init_modules(): register chrdev(245,0)
[ 2960.324039] Tutorial_11:drv_open(): device open
[ 2960.324042] Tutorial_11:drv_write(): IO Mode is 1
[ 2960.324043] Tutorial_11:drv_write(): block
[ 2960.324046] In routine!
[ 2965.331478] Work routine completed!
[ 2965.331533] Tutorial_11:drv_write(): IO Mode is 0
[ 2965.331534] Tutorial_11,drv_write(): non-blocking
[ 2965.331897] Tutorial_11:drv_release(): device close
[ 2965.336558] In routine!
[ 2970.448867] Work routine completed!
```

# Assignment 5 makefile and scripts

- Command “make” includes executions:
  - Build kernel module “mydev.ko”
  - Insert kernel module “sudo insmod mydev.ko”
- For first time active your device:
  - Check available MAJOR and MINOR number. (When initialize your kernel module, use alloc_chrdev_region() and MAJOR() / MINOR() to check. After insert your module, type “dmesg” to check the result.)
  - create a file node for “mydev”. Run “sudo ./mkdev.sh  MAJOR MINOR”.
  - You don’t need to create this file node every time. Just create it if it does not exist. (You can run “sudo ./rmdev.sh” to remove this file node.)
- Run test “./test”:
- Command “make clean” includes executions:
  - Remove kernel module “sudo rmmod mydev.ko”
  - Clean test executable file “rm test”
  - Display kernel message including “OS_AS5” (“dmesg | grep OS_AS5”)

# Bonus Hints

## Global View (Bonus)



- Find out keyboard's interrupt number and then register an ISR

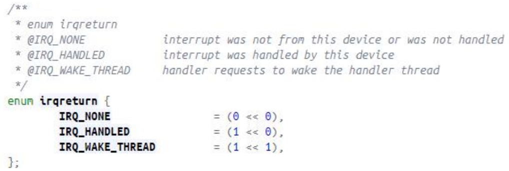
# Bonus Hints

-
  ```
  request_irq (     unsigned int        irq,
                    irq_handler_t       handler,
                    unsigned long       irqflags,
                    const char *        devname,
                    void *      dev_id);
  ```
  - irq: Interrupt line to allocate (Hints: Can check from “watch -n 1 cat /proc/interrupts”)
  - handler: Function to be called when the IRQ occurs. (Hints: count interrupt times when IRQ occurs)
  - irqflags: Interrupt type flags (Hints: IRQF_SHARED Interrupt is shared)
  - devname: An ascii name for the claiming device. (Hints: Once your kernel module is inserted/removed, it will auto be displayed under “watch -n 1 cat /proc/interrupts”. Define any name you want to show.)
  - dev_id: A cookie passed back to the handler function.

# Bonus Hints

- free_irq ( unsigned int irq,
  void * dev_id);
  - irq: Interrupt line to free
  - dev_id: Device identity to free.

- irq_handler_t handler
  - https://elixir.bootlin.com/linux/v4.10.14/source/include/linux/interrupt
  - typedef irqreturn_t (*irq_handler_t)(int, void *);
  - https://elixir.bootlin.com/linux/v4.10.14/source/include/linux/irqreturn.h

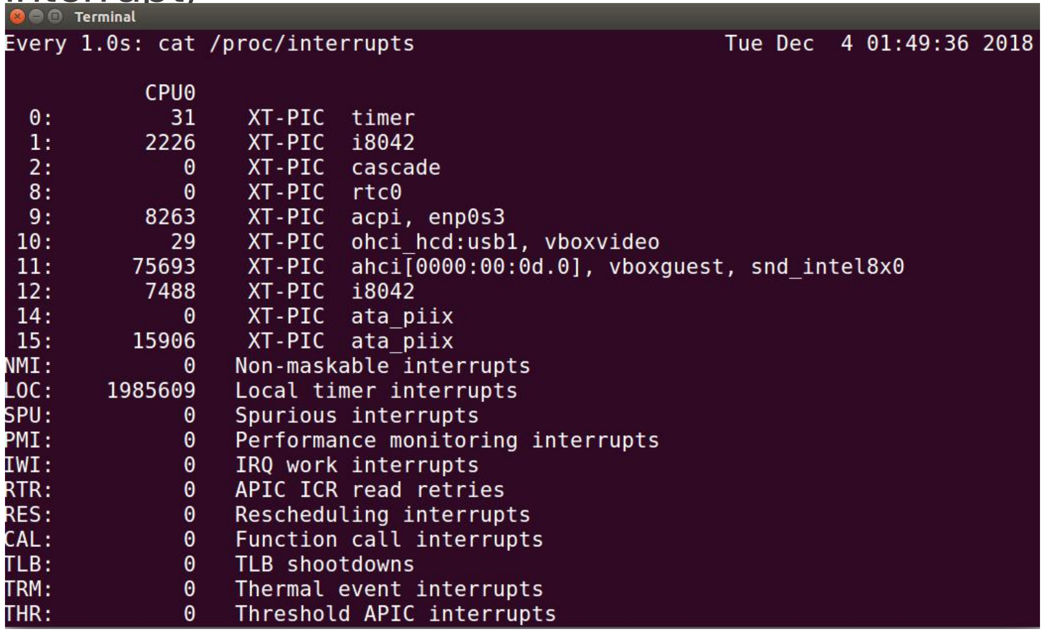


```
/**
 * enum irqreturn
 * @IRQ_NONE            interrupt was not from this device or was not handled
 * @IRQ_HANDLED         interrupt was handled by this device
 * @IRQ_WAKE_THREAD     handler requests to wake the handler thread
 */
enum irqreturn {
        IRQ_NONE                = (0 << 0),
        IRQ_HANDLED             = (1 << 0),
        IRQ_WAKE_THREAD         = (1 << 1),
};
```

- More detailed usage for irq handler
  - Understanding Linux Kernel, Page 591 - 595

# Bonus Hints

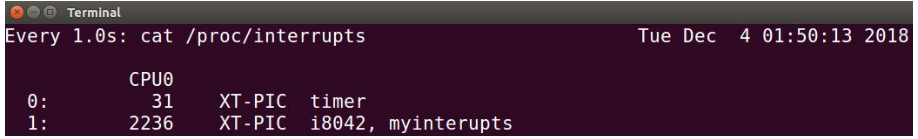
- Run command “watch -n 1 cat /proc/interrupts”
  - IRQ_NUM should be defined as 1 for below case. (Share interrupt with i8042, the keyboard interrupt)



```
Every 1.0s: cat /proc/interrupts                        Tue Dec  4 01:49:36 2018

           CPU0
  0:         31    XT-PIC  timer
  1:       2226    XT-PIC  i8042
  2:          0    XT-PIC  cascade
  8:          0    XT-PIC  rtc0
  9:       8263    XT-PIC  acpi, enp0s3
 10:         29    XT-PIC  ohci_hcd:usb1, vboxvideo
 11:      75693    XT-PIC  ahci[0000:00:0d.0], vboxguest, snd_intel8x0
 12:       7488    XT-PIC  i8042
 14:          0    XT-PIC  ata_piix
 15:      15906    XT-PIC  ata_piix
NMI:          0   Non-maskable interrupts
LOC:    1985609   Local timer interrupts
SPU:          0   Spurious interrupts
PMI:          0   Performance monitoring interrupts
IWI:          0   IRQ work interrupts
RTR:          0   APIC ICR read retries
RES:          0   Rescheduling interrupts
CAL:          0   Function call interrupts
TLB:          0   TLB shootdowns
TRM:          0   Thermal event interrupts
THR:          0   Threshold APIC interrupts
```

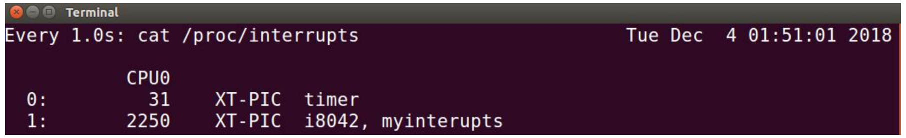
# Bonus Hints

- Inserted mydev.ko



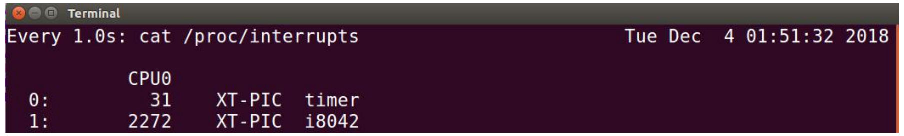
```
Every 1.0s: cat /proc/interrupts                       Tue Dec  4 01:50:13 2018

           CPU0
  0:         31    XT-PIC  timer
  1:       2236    XT-PIC  i8042, myinterupts
```

- After run test



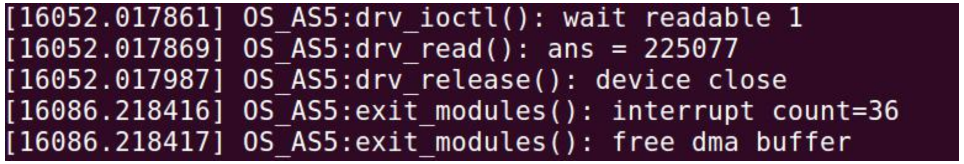
```
Every 1.0s: cat /proc/interrupts                       Tue Dec  4 01:51:01 2018

           CPU0
  0:         31    XT-PIC  timer
  1:       2250    XT-PIC  i8042, myinterupts
```

- Remove mydev.ko



```
Every 1.0s: cat /proc/interrupts                       Tue Dec  4 01:51:32 2018

           CPU0
  0:         31    XT-PIC  timer
  1:       2272    XT-PIC  i8042
```

- Interrupt counting



```
[16052.017861] OS_AS5:drv_ioctl(): wait readable 1
[16052.017869] OS_AS5:drv_read(): ans = 225077
[16052.017987] OS_AS5:drv_release(): device close
[16086.218416] OS_AS5:exit_modules(): interrupt count=36
[16086.218417] OS_AS5:exit_modules(): free dma buffer
```

# Assignment 5 Problems

- “Command not found” for “sudo ./mkdev.sh Major Minor”.
    - Do not use the script anymore, type the command directly.
    - sudo mknod /dev/mydev c major minor
    - sudo chmod 666 /dev/mydev
    - ls -l /dev/mydev

# Assignment 5 Problems

- get_user / put_user cannot transfer data with type DataIn
  - This macro copies a single simple variable from user space to kernel space.
  - It supports simple types like char and int, but not larger data types like structures or arrays.
  - Some students may get killed error when running test program if transferring DataIn via get_user / put_user.
  - For data with type DataIn, you can transfer via buffer, second parameter of the drv_write / drv_read.
  - Or you could use copy_from_user / copy_to_user.

# Assignment 5 Problems

- How to change device readable setting?
  - When the device is unreadable, use a while loop with msleep in kernel.
  - Use the while loop to keep checking readable setting.
  - If unreadable, use msleep to let this work to be sleeped.
  - In this case, the CPU is available to continue computation.
  - Once the computation completes, change readable setting.
  - Then in while loop, if it detects the readable setting is updated, it will stop looping.

# Assignment 5 Problems

- When will drv_release be called?
  - If close() is called in user program, drv_release() will be called in kernel.
  - If close() is not called in user program, when that process is terminated (./test completes execution), close() will be automatically called.

# Assignment 5 Problems

- Bonus: interrupt counting will not always be 36.
  - It depend on what you’ve typed in via keyboard.
  - You only need to calculate the interrupt numbers during mydev.ko being inserted.
  - You could verify the number via “watch -n 1 cat /proc/interrupts”.
  - Sometimes, even if you only press once in keyboard, the interrupt will occur twice or multiple times.

# Reference

- Work queues
  - https://www.ibm.com/developerworks/library/l-tasklets/index.html
  - https://linuxtv.org/downloads/v4l-dvb-internals/device-drivers/ch01s06.html

- Get user / Put user
  - https://www.fsl.cs.sunysb.edu/kernel-api/re244.html
  - https://www.fsl.cs.sunysb.edu/kernel-api/re245.html

- Request and free irq
  - https://elixir.bootlin.com/linux/v4.10.14/source/include/linux/interrupt.h
  - https://www.fsl.cs.sunysb.edu/kernel-api/re667.html
  - https://www.kernel.org/doc/htmldocs/kernel-api/API-free-irq.html

Thank you