> Name:
>
> Liqin Zhang 517370910123
>
> Siwei Ye 517370910122

Lab 9 Dice Module

A dice module

Tasks

- What needs to be returned by read and write file operations for a character device?

In gerneral, read and write for a character device have the following format.

```
ssize_t read(struct file *filp, char *buff, size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char *buff, size_t count, loff_t *offp);
```

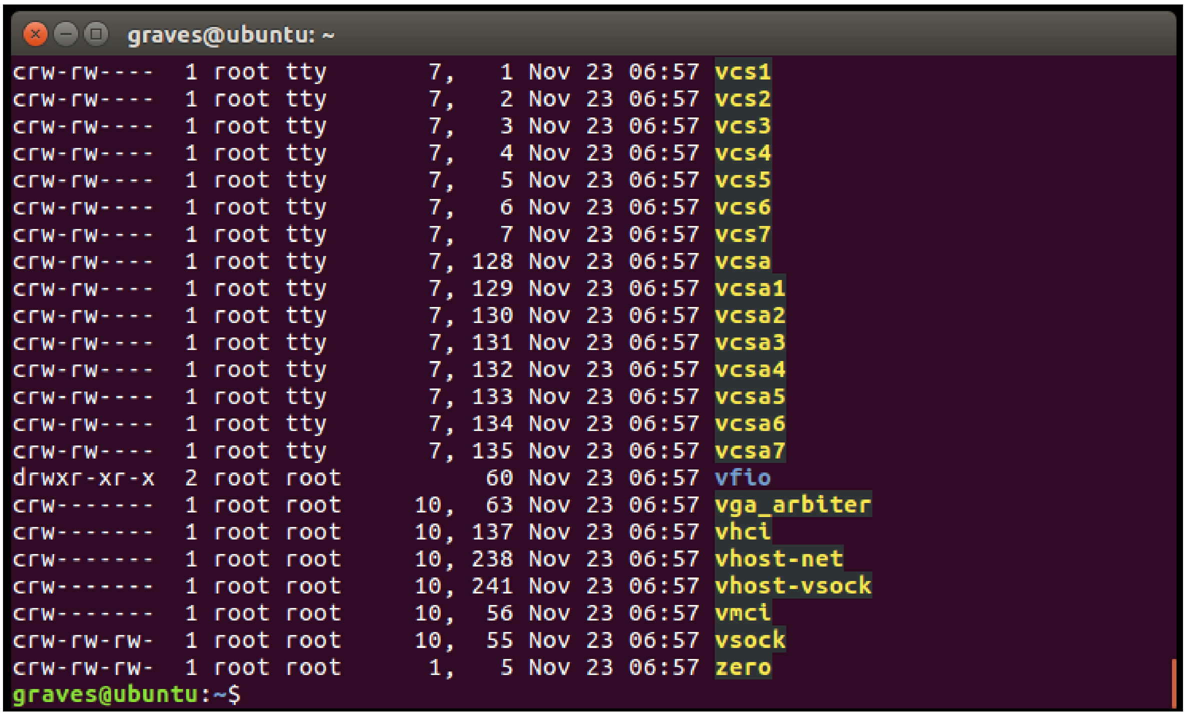
where `filp` is the file pointer and `count` is the size of the requested data transfer. The `buff` argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed. The return value is a `signed size type`.

Both the *read* and *write* methods return a negative value if an error occurs. A return value greater than or equal to 0 tells the calling program how many bytes have been successfully transferred. If some data is transferred correctly and then an error happens, the return value must be the count of bytes successfully transferred, and the error does not get reported until the next time the function is called.

Although kernel functions return a negative number to signal an error, and the value of the number indicates the kind of error that occurred, programs that run in user space always see -1 as the error return value. They need to access the `errno` variable to find out what happened. The difference in behavior is dictated by the POSIX calling standard for system calls and the advantage of not dealing with `errno` in the kernel.

- How are exactly those major and minor numbers working? You vaguely remember that you can
  display them using ls `-l /dev`

By using `ls -l /dev`, we have the following result



```
crw-rw----  1 root tty       7,   1 Nov 23 06:57 vcs1
crw-rw----  1 root tty       7,   2 Nov 23 06:57 vcs2
crw-rw----  1 root tty       7,   3 Nov 23 06:57 vcs3
crw-rw----  1 root tty       7,   4 Nov 23 06:57 vcs4
crw-rw----  1 root tty       7,   5 Nov 23 06:57 vcs5
crw-rw----  1 root tty       7,   6 Nov 23 06:57 vcs6
crw-rw----  1 root tty       7,   7 Nov 23 06:57 vcs7
crw-rw----  1 root tty       7, 128 Nov 23 06:57 vcsa
crw-rw----  1 root tty       7, 129 Nov 23 06:57 vcsa1
crw-rw----  1 root tty       7, 130 Nov 23 06:57 vcsa2
crw-rw----  1 root tty       7, 131 Nov 23 06:57 vcsa3
crw-rw----  1 root tty       7, 132 Nov 23 06:57 vcsa4
crw-rw----  1 root tty       7, 133 Nov 23 06:57 vcsa5
crw-rw----  1 root tty       7, 134 Nov 23 06:57 vcsa6
crw-rw----  1 root tty       7, 135 Nov 23 06:57 vcsa7
drwxr-xr-x  2 root root          60 Nov 23 06:57 vfio
crw-------  1 root root     10,  63 Nov 23 06:57 vga_arbiter
crw-------  1 root root     10, 137 Nov 23 06:57 vhci
crw-------  1 root root     10, 238 Nov 23 06:57 vhost-net
crw-------  1 root root     10, 241 Nov 23 06:57 vhost-vsock
crw-------  1 root root     10,  56 Nov 23 06:57 vmci
crw-rw-rw-  1 root root     10,  55 Nov 23 06:57 vsock
crw-rw-rw-  1 root root      1,   5 Nov 23 06:57 zero
graves@ubuntu:~$
```

From above, we notice that there are two numbers with the form as `[xx, yy]`, where `xx` denotes the major number and `yy` denotes the minor number for the device.

The major number identifies the driver associated with the device. For example, `/dev/vcs1` and `/dev/vcs2` are both managed by driver 7, similarly, both `vga_arbiter` and `vhci` devices are managed by driver 10. The kernel uses the major number at *open* time to dispatch execution to the appropriate driver.

The minor number is used only by the driver specified by the major number; other parts of the kernel don't use it, and merely pass it along to the driver. It is common for a driver to control several devices (as shown in the listing); the minor number provides a way for the driver to differentiate among them.

- Knowing the major number and minor numbers of a device, how to add a character device to /dev?

First we have to had a char device driver with following file options

```
struct file_operations fun{
.open=open_fun,
.release=release_fun,
.write=write_fun,
.read=read_fun,
};
```

- Then, include the header file **linux/device.h** and **linux/kdev_t.h**
  - Add static struct class c_dev;
  - Add static struct dev_t dev;
- Add the below API 's inside __init fuction of the driver
  - `cl = class_create(THIS_MODULE ,"x")` where x - Name to be displayed inside /sys/class/ when driver is loaded.
  - Use device_create () kernel api with `device_create(cl, NULL, dev, NULL, "d")` where device file to be created under `/dev`.
- Where are the following terms located in linux source code

    - `module_init`: defined as a prototype: `include/linux/module.h`
    - `module_exit`: defined as a prototype: `include/linux/module.h`
    - `printk`: defined as a prototype: `include/linux/printk.h`
    - `container_of`: defined as a macro: `include/linux/kernel.h`
    - `dev_t`: defined as a typedef: `include/linux/types.h`
    - `MAJOR`: defined as a macro: `include/linux/kdev_t.h`
    - `MINOR`: defined as a macro: `include/linux/kdev_t.h`
    - `MKDEV`: defined as a macro`include/linux/kdev_t.h`
    - `alloc_chardev_region`:
        - defined as a prototype: `include/linux/fs.h`
        - defined as a function: `fs/char_dev.c`
    - `module_param`: refered as `kernel/module.c`
    - `cdev_init`: defined as a prototype: `include/linux/cdev.h`
    - `cdev_add`: defined as a prototype: `include/linux/cdev.h`
    - `cdev_del`: defined as a prototype: `include/linux/cdev.h`
    - `THIS_MODULE`:
        - defined in as a macro: `include/linux/export.h`
        - referred in: `include/linux/module.h`
- How to generate random numbers when working inside the Linux kernel? You think that a while back you read something about getting the current time.

  To generate random numbers in kernel space, we can use `void get_random_bytes(void *buf, int nbytes)` defined in `include/linux/random.h`. However, the returned value is a pseudorandom number, to make it truely random, we can divide it by time, which can be obtained through `void getnstimeofday(struct timespec *ts)` from `linix/time.h`

- How to define and specify module options?

  `module_param(name, type, perm)` enables defining module parameters during `insmod`, where:

    - `name` name of the variable
    - `type`: type of variable
    - `perm`: visibility in sysfs (0 if the variable is not to appear in sysfs, i.e. unchanged & unseen after `insmod`)

  e.g. Assign a major number when `insmod`:

  ```
  int gen_sides = 20;
  module_param(gen_sides, int, 0);
  ```

  Then we can specify the module option on `insmod`:

  ```
  insmod dicedevice.ko gen_sides=30
  ```

Code Implementation

See `/src`, and check `README.md` to learn how to use the dice device.