VE482 - Lab 9

Introduction to Operating Systems

Weili Shi 519370910011

Kexuan Huang 518370910126

December 3, 2021

3. Tasks

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

According to *The Linux Kernel Documentation*,

> The value returned by read or write can be:
>
> - the number of bytes transferred; if the returned value is less than the size parameter (the number of bytes requested), then it means that a partial transfer was made. Most of the time, the user-space app calls the system call (read or write) function until the required data number is transferred.
> - 0 to mark the end of the file in the case of read ; if write returns the value 0 then it means that no byte has been written and that no error has occurred; In this case, the user-space application retries the write call.
> - a negative value indicating an error code.

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

According to *The Linux Kernel Documentation*,

> The identifier consists of two parts: major and minor.
>
> - The first part identifies the device type (IDE disk, SCSI disk, serial port, etc.)
> - The second one identifies the device (first disk, second serial port, etc.).
>
> Most times, the major identifies the driver, while the minor identifies each physical device served by the driver. In general, a driver will have a major associate and will be responsible for all minors associated with that major.
>
> To create a device type file, use the `mknod` command; the command receives the type (block or character), major and minor of the device (mknod name type major minor).
>
> Thus, if you want to create a character device named mycdev with the major 42 and minor 0, use the command:

```
1  # mknod /dev/mycdev c 42 0
```

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

With `cdev_add()` and number of major and minor, we can add a character device to `/dev`:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

According to *The Linux Kernel Documentation*, the devices can be registered and initialized by:

```
#include <linux/fs.h>
#include <linux/cdev.h>

#define MY_MAJOR       42
#define MY_MAX_MINORS  5

struct my_device_data {
    struct cdev cdev;
    /* my data starts here */
    //...
};

struct my_device_data devs[MY_MAX_MINORS];

const struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .read = my_read,
    .write = my_write,
    .release = my_release,
    .unlocked_ioctl = my_ioctl
};

int init_module(void)
{
    int i, err;
    err = register_chrdev_region(MKDEV(MY_MAJOR, 0), MY_MAX_MINORS
                                 ,"my_device_driver");
    if (err != 0) {
        /* report error */
        return err;
    }
    for(i = 0; i < MY_MAX_MINORS; i++) {
        /* initialize devs[i] fields */
        cdev_init(&devs[i].cdev, &my_fops);
        cdev_add(&devs[i].cdev, MKDEV(MY_MAJOR, i), 1);
    }
    return 0;
}
```

Where are the following terms located in linux source code?

According to results in *Bootlin Elixir Cross Referencer*:

- `module_init` `include/linux/module.h`
- `module_exit` `include/linux/module.h`
- `printk` `include/linux/printk.h`
- `container_of` `include/linux/kernel.h`
- `dev_t` `include/linux/types.h`
- `MAJOR` `include/linux/kdev_t.h`
- `MINOR` `include/linux/kdev_t.h`
- `MKDEV` `include/linux/kdev_t.h`
- `alloc_chrdev_region` `include/linux/fs.h`
- `module_param` `include/linux/moduleparam.h`
- `cdev_init` `include/linux/cdev.h`
- `cdev_add` `include/linux/cdev.h`
- `cdev_del` `include/linux/cdev.h`
- `THIS_MODULE` `include/linux/export.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.

```
void get_random_bytes(void *buf, int nbytes);
```

> Returns the requested number of random bytes and stores them in a buffer.

How to define and specify module options?

According to *Linux manual page*,

> The `module_init()` macro defines which function is to be called at module insertion time (if the file is compiled as a module), or at boot time
>
> (if the file is not compiled as a module the `module_init()` macro becomes equivalent to `__initcall()`, which through linker magic ensures that the function is called on boot.)

- To load a module by filename:

  ```
  # insmod <filename> [args]
  ```

- To pass a parameter to a kernel module,

  ```
  # modprobe <module_name> [parameter=value]
  ```