

# CSIS0230A Principles of Operating Systems (Class A)

## Notes for Tutorial 6

### Device driver: an introduction

Most kernel code are for device drivers. Our study of the kernel would be incomplete if we omit this large chunk of code. Most device drivers are in modules that can be added to and removed from the kernel conveniently. In the tutorial this week we will try to write a simple kernel module which adds an extremely simple device to the kernel.

#### 1. Kernel modules

Most kernel facilities (that is, except system calls) can be introduced into the kernel using kernel modules. This allows us to change the kernel code when the kernel is running, thus reducing the need for lengthy reboot cycles, and allow programmers to load and unload code for testing. A kernel module source file is a stand-alone C file. Suppose we have the following `mod.c`:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
/* other functions of the module */
static int mymod_init(void) {
    /* should register something here */
    printk("<0>Hello, kernel module world!\n");
    return 0; /* should return 0 if succeed, non-zero if failure */
}
static void mymod_exit(void) {
    printk("<0>Bye, kernel module world!\n");
}
module_init(mymod_init); /* Run mymod_init() on module initialization */
module_exit(mymod_exit); /* Run mymod_exit() on module removal */
/* other information of the module */
```

To compile it, use

```
gcc -Wall -O2 -D__KERNEL__ -DMODULE -c -I <kroot>/include mod.c
```

Note that we only compile the module, not the whole kernel. Once you get the compiled `mod.o`, you can become `root` and install it to a running kernel using `insmod mod.o`, and remove it from the kernel using `rmmmod mod.o`.

As noted in the comments within the code, the kernel will call an **init function** `mymod_init()` immediately after a new module is loaded, and an **exit function** `mymod_exit()` immediately before a module is unloaded. The former function should return an integer indicating whether your module think it is okay to load the module. If not, then your `init_module()` function should return an error code (like `-ENODEV`, “no such device”). Otherwise, 0 should be returned. In the former case (i.e., an error occur), the module is not considered to be loaded into the memory, and the memory it occupies will be deallocated immediately.

Most modules do very few things when being loaded. They just **register some functions** to the kernel, so that when the functionalities are needed, those functions are called. We will see one way to register a function later in this notes.

Our module can **define some information** that can be retrieved using `modinfo mod.o`. They are defined in the global scope of the module source file. Some macros in `<linux/module.h>` can be used to make such information available, like `MODULE_LICENSE`, `MODULE_DESCRIPTION` and `MODULE_AUTHOR`. E.g., to declare your module uses the GPL license, use this.

```
MODULE_LICENSE("GPL");
```

Finally, every loaded module has a **usage count**. When this count drops to 0, the OS may remove the module from memory. This is a safety net against accidental removal of a module when it is in use (which would crash the kernel). To increment and decrement the usage count, one may use the `MOD_INC_USE_COUNT` and `MOD_DEC_USE_COUNT` macros. But in case of character device drivers, we will see that this can be done automatically.

## 2. Device major number, minor number, and registration

Now it is time to see how the kernel knows about devices. There are two types of devices (character vs. block), and we will focus on **character devices**. Two “small” numbers identify a character device: the **major number** and the **minor number**. These two numbers must be specified when you create a device file (using a command-line program called `mknod`). When a device is opened, the kernel creates a *file* object (to be described later), and calls an “open” method for the object. The open method to call is selected based solely on the device major number. The open method, which is defined by the device driver, may then modify the object based on the minor number (or any other information), thus selecting another set of methods to be used for subsequent calls (e.g., when the device is read, written or closed). So **the major numbers are for use by the kernel, while the minor numbers are for use by the device drivers**.

It is important to understand the role of a device driver: **device drivers serves the kernel**, not the other way round. E.g., when a process want to read data from your device, it uses a `read()` system call to ask the kernel to read it. The kernel then **calls the device driver**, in particular a “read method” of the device driver, to get the data. The device driver must then **generate the data by itself**, e.g., by reading hardware devices, etc. It cannot ask the kernel for the data.

When a device driver module is **loaded**, it **registers a set of methods** to be called when a particular major number is accessed. This is done using `register_chrdev(unsigned int major, const char *name, struct file_operations *fops)`. Here *major* is the major device number of the device driver. If you **specify 0**, you will instead **get a major device number allocated dynamically** (via the return value of `register_chrdev()`). The *name* argument specifies a string that would appear when you read the `/proc/devices` file, and must be presented again when you unregister the device. The *fops* argument specifies a structure which contains the device methods. We will examine this structure in a just moment. A negative return value means the registration fails. The value is suitable as a return value of the system call, like `-EBUSY` or `-EINVAL`.

Before you **unload** a device driver module, you must **unregister** its major number. This can be done by calling the function `unregister_chrdev(unsigned int major, const char *name)`. Again, a negative result indicates an error. The only possible error is `-EINVAL`.

## 3. The file object

When a device is opened, a **file object**, of type `struct file`, is allocated and filled by the kernel to

keep information like the current access pointer, the associated directory entry, the open mode, the file operations functions to use for this file, etc. The *open* method is then called, so that the device driver can perform further initialization. The kernel will keep this structure until all copies<sup>1</sup> are closed. At that time the *release* method is called, and the kernel deallocate the *file* object. It is of the **struct file** type, which is **defined in** `<linux/fs.h>`. Some more important members include:

*loff\_t f\_pos*;

The current file position. This should never be changed by the device driver directly. Instead, when a *read* and *write* method is called, a pointer to the *f\_pos* is passed, and the modification should be done via that pointer.

**struct file\_operations** \*f\_op;

The operations associated with the file. A device driver will modify this pointer during the *open* function if it uses different set of operations on different minor numbers.

**void** \*private\_data;

Each file has an extra pointer, and the kernel do not use this pointer. If the device driver writer wants to keep some data specific for the *file* object, it will allocate memory, and let *private\_data* point to the allocated memory.

#### 4. The file operations

It remains to examine how to define methods (like the *open* and *release* methods). They are just normal functions, with predefined prototypes. Let's see a few of them:

**static int** *foo\_open*(**struct inode** \*inode, **struct file** \*filp)

Called when the file is opened. The function should return 0 if it decides that the file opening is to succeed, and return an error code (like `-EPERM`) otherwise, explaining why the call failed. Many device drivers use this function to allocate private data and modify the file operations structure of the device based on the minor number of the device. The device number can be found in the *inode* passed to this function, using `inode->i_rdev`. The minor number can then be extracted using the `MINOR` macro like `MINOR(inode->i_rdev)`. If the function is missing, no initialization is done.

**static int** *foo\_release*(**struct inode** \*inode, **struct file** \*filp)

Called when the file is finally for the last time. The function should return 0 if the file closing succeeds, and an error code otherwise. If missing, no deinitialization is done when the file object is removed.

**static ssize\_t** *foo\_read*(**struct file** \*filp, **char** \*buffer, **size\_t** count, **loff\_t** \*f\_pos);

This is called when the user program tries to read from the file. At most *count* bytes should be written to the user-provided *buffer*, probably using `copy_to_user` or one of its variants. The code should increment the number *\*f\_pos* by the number of bytes read. The return value of this function should be the number of bytes read, or 0 if it reaches the end of file, or a negative error code to indicate an error. It is possible that the number of bytes read is less than the number of bytes requested to be read, and it does not indicate any error condition

---

<sup>1</sup>The `fork()`, `dup` and `dup2` system calls allow the same *file* structure to be used multiple times. Each of them can be removed by `close()` of the file, termination of the process holding the file, or the file descriptor being used for other purpose by `dup2()`.

nor EOF condition. The user should call *read* again if he desires to read the remaining bytes. If this method is missing, reading the file will return an error (*EINVAL*).

```
static ssize_t foo_write(struct file *filp, const char *buffer, size_t count, loff_t *f_pos);
```

This is called when the user program tries to write into the file. At most *count* bytes should be copied from the user-provided *buffer*, probably using *copy\_from\_user* or one of its variants. The code should increment the number *\*f\_pos* by the number of bytes written. The return value of this function should be the number of bytes written, or a negative error code to indicate error. The number of bytes written may be less than *count*, and may even be 0; they should not be interpreted as errors. In these cases the user program should call *write* again to write the remaining bytes. Again, this method can be missing, making it an error to write to the device.

After writing a set of device methods (probably replacing the “*foo*” word with something more meaningful), we need to **tell the kernel about them**. A set of such device methods is defined by a **file operation structure**, which has one field (function pointer) for each possible file operation. A simple device driver has only one file operation structure, registered to the kernel through the *register\_chrdrv()* function. A more complicated one has several of them, one for each type of device that it handles. During device registration, the “main” file operations structure is registered to the kernel. Thus, when a device is opened, the main *open* method is called, which can assign a different file operation structure the *fops* field of the *file* object. The file operation structure is defined like this:

```
static struct file_operations foo_fops = {  
    .open = foo_open,  
    .release = foo_release,  
    .read = foo_read,  
    .write = foo_write,  
    .owner = THIS_MODULE,  
};
```

Earlier in the note we say that **usage count** of a character device driver module can be maintained automatically. The automatic usage counting comes from the *owner* field of the structure. If it is assigned, the usage counter of the module will be incremented and decremented automatically when a file with this *file\_operations* structure is created and destroyed by the kernel.

## 5. References

This notes aim to be a quickstart guide to writing device driver, and no attempt is made to make sure that the resulting device driver is clean, SMP-aware or portable. For details about writing device drivers, you should read the following book:

Linux Device Drivers, 2nd Edition.

(Available Online at <http://www.xml.com/ldd/chapter/book/> )