

CSIS0230A Principles of Operating Systems (Class A)

Notes for Tutorial 10

Device driver: an introduction

Most kernel code are device drivers. Our study of the OS kernel would be incomplete if we omit this large part. 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

In the note of tutorial 4, we introduced kernel modules. Now is a good time to review the information there. Essentially, from a source file `mod.c` of a kernel module, you compile it with `gcc -Wall -O2 -c mod.c` (you might want to add `-I <kernel-path>/linux` as well to make sure that the headers included are those of the currently installed kernel version). 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`.

Most modules does not do a lot of things when being loaded. They just register some functions to the kernel, so that when the functionalities are needed, those functions are called. The skeleton of the source code of a typical kernel module looks like this:

```
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif
/* define functions of the module */
int init_module(void) {
    register_something(...);
    return 0; /* should return 0 if succeed, non-zero if failure */
}
void cleanup_module(void) {
    unregister_something(...);
}
/* extra information of the module */
```

You can also obtain the information defined in the module using `modinfo mod.o`. They are simply static data defined within the module file. Some macros are defined in `<linux/modules.h>` for making such information available, like `MODULE_LICENSE`, `MODULE_DESCRIPTION` and `MODULE_AUTHOR`. E.g.,

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

declares that your module uses the GPL license.

Finally, every loaded module has a reference count. When this count drops to 0, the OS may remove it 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 reference 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 `mknod`). When a device is opened, the kernel will create a *file* object (to be described later), and call an initialize function for the object. The initialization function to call is selected based solely on the device **major** number. The function, which is within the device driver, may then **modify** the object based on the **minor** number (or any other information), thus selecting another set of functions 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.

When a device driver module is loaded, it would register its set of function to be called when a particular major number is accessed. This can be done by using the `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 the function). 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 functions, which we will examine in a moment. The return value would be negative if registration fails.

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

3. The in-kernel *file* structure

When a device is opened, a *file* structure 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 references¹ to it are removed, by process being terminated, explicitly `close()` the file, or used `dup2()` which implicitly closed the file. At that time the *release* method is called, and the kernel deallocate the *file* structure. The *file* structure 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* is called, a pointer is given, and the modification should be done there.

`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 has no use with this pointer. If the device driver writer wants to keep some data specific for the *file* structure, it will allocate a structure (using

¹The `fork()`, `dup` and `dup2` system calls allow the same *file* structure to be used multiple times.

kmalloc) and put the pointer to the structure here when the device is first opened. This should be deallocated when the file is destroyed.

4. The file operations

It remains to examine how to define methods like the *open()* and *release()*. They are 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 valid, 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. The function can thus be missing, in which case no initialization is done by the device driver.

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 a file is closed.

static ssize_t *foo_read*(**struct file** **file*, **char** **buffer*, **size_t** *count*, **loff_t** **filp*);

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 **filp* 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 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** **file*, **char** **buffer*, **size_t** *count*, **loff_t** **filp*);

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

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

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 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/>)