OS support on IMS report #3

Yusen Liu
liu797@wisc.edu

Jan. 28 ~Feb. 4

Abstract

Each piece of code that can be added to the kernel at runtime is called a module. Device drivers is one type of module. This week, I went through the chapters 2 and 3 of *Linux Device Drivers*, compiled a simple hello world module, and modified the example codes of a character driver from GitHub.

1 Introduction

A module runs in kernel space, whereas applications run in user space. Each module is made up of object code, which is not linked into a complete executable, that can be dynamically linked to the running kernel by the `insmod` program, and can be unlinked by the `rmmod` program. Executing the previous two programs requires sudo mode. Every kernel module registers itself with the aim to serve future requests. The module's initialization function terminates immediately. Hence, the task of its initialization function is to prepare for later invocation of the module's functions. The exit function of a module must carefully undo everything the init function built up, or the pieces remain around until the system is rebooted[1].

The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, including a *char module*, a *block module*, or a *network module*. In this week, I dived into the *char module*. A character device is the one that can be accessed as a stream of bytes (like a file). What a char driver does, is to implement this behavior , including *open*, *close*, *read*, *write* syscalls, etc.. Example of char devices include the text console (*/dev/console*) and the serial ports (*/dev/ttyS0* and friends) because they are represented by the stream abstraction.

Often, as we look at the kernel API, we could encounter function names starting with a double underscore. Those functions are a low-level component of the interface and should be used with caution. Kernel code cannot do floating point arithmetic.

2 Methodology

2.1 Simple hello world module

Below is a brief demonstration of building a module and loading it into the system. The file's name is `hello.c`.

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Note that:

- We need *init.h* to specify your initialization and cleanup functions.
- *module.h* contains a great many definitions of symbols and functions needed by loadable modules.
- Most modules also include *moduleparam.h* to enable the passing of parameters to the module at load time.
- The function `hello_init` is to be invoked when the module is loaded into the kernel, and the function `hello_exit` is for when the module is removed.
- The two special kernel macros on the last two lines - `module_init` and `module_exit` are used to indicate the role of the two functions we defined.
- The macro `MODULE_LICENSE` is used to tell the kernel that this module bears a free license.
- The `printk` function is defined in the Linux kernel and made available to modules. It can be called by the module because, after *insmod* has loaded it, the module is linked to the kernel and can access the kernel's public symbols.

- The string `KERN_ALERT` is the priority of the message, indicating that we specify a high priority in this module. Without it, a message might not show up anywhere useful.

The module initialization function registers any facility offered by the module. The actual definition of the init function looks like:

```
static int __init init_func(void)
{
    // initialization code here
}
module_init(init_func);
```

Initialization functions should be declared

After we have saved the file `hello.c`, we need to write the Makefile. For this simple module, a single line will suffice:

```
obj-m := hello.o
```

The above line is that the kernel build system handles the rest. The assignment above takes advantage of the extended syntax provided by GNU make, and states that there is one module to be build from the object file `hello.o`.

If we have a module named `module.ko` which is generated from two source files `file1.c` and `file2.c`, then the Makefile should be:

```
obj-m := module.o
module-objs := file1.o file2.o
```

For the Makefile to work, it must be invoded within the context of the larger kernel build system. Hence, we need to figure out where the kernel source tree locates. usually in the directory: `/lib/modules/$(shell uname -r)/build/`. The command is as follows:

```
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
```

where

```
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
```

The cleanup command is:

```
$(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

Hence, the Makefile should be like:

```
# if KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.

ifneq ($(KERNELRELEASE),)
obj-m := hello.o

# otherwise, we were called directly from the command line
# invoke the kernel build system
else
    KERNELDIR := /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif

clean:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) clean

test:
    sudo dmesg -C
    sudo insmod hello.ko
    sudo rmmod hello.ko
    dmesg
```

Note that, this following paragraph is copied from the book directly, without a lot of modifications due to the fact that I did not fully understand how this Makefile works. The Makefile is read twice on a typical build. When the Makefile is invoked from the command line, it would notice that the `KERNELRELEASE` variable has not been set. It locates the kernel source directory by taking advantage of the fact that the symbolic link build in the installed modules directory points back at the kernel build tree. Once the kernel source tree has been found, the Makefile invokes the default: target, which runs a second make command (parameterized in the makefile as `$(MAKE)`) to invoke the kernel build system. On the second reading, the Makefile sets `obj-m`, and the kernel makefiles take care of actually building the module.

After running `$ make`, we could run `$ make test` to check the output, which is shown below:

```
[12432.963916] Hello world
```

```
[12432.974035] Goodbye world
```

Note that running `$ make test` means we execute four command lines. The first line `sudo dmesg -C` is to clear the message buffer of the kernel. The second line is used to insert the module we have built. After executing the second line, we could execute `lsmod \ grep hello|` to check if the output includes the module we have inserted; the output on my virtual machine is:

```
hello                  16384  0
```

This means that the simple hello world module has been successfully inserted. The third line `sudo rmmod hello.ko` or `sudo rmmod hello` is used to remove the module. The last line then prints the message buffer of the kernel. Note that, another way of printing the kernel message is to execute `tail /var/log/syslog`.

2.2 Character driver

In this section, what I did was to follow the Chapter 3 of the book[1] and downloaded the example code from Javier Martinez's GitHub repository https://github.com/martinezjavier/ldd3. Since the chapter presents code fragments extracted from a real device driver - *scull*, downloading the loadable kernel module and reading the source code would be a faster approach to get used to the functionalities of the character driver. *scull* is a char driver that acts on a memory area as though it were a device.

Char devices are accessed through names in the filesystem. The names are located in */dev* directory. Files for char drivers are identified by a "c" in the first column of the output by running `$ ls -l /dev` while block devices are identified by a "b". Directories are identified by a "d". They are all considered as files in Linux.

Under the directory of `scull`, we could see several files including *access.c*, *main.c*, *Makefile*, *pipe.c*, *scull.h*, *scull.init*, *scull_load* and *scull_unload*. One common error might occur is that, the 5.0 kernel dropped the *type* argument of the function `access_ok()`. Hence, if there is an error regarding to that, we need to remove the *type* argument in the function in *main.c*.

Before executing the `$ make` command, I added followed a YouTube video (https://www.youtube.com/watch?v=juGNPLdjLH4) to add several features to the code. The first was, inside the `scull_open` in *main.c*, I added one line to it:

```
printk(KERN_ALERT "Hello world\n");
```

Inside the function `scull_read()`, in order to see how many bytes we were reading when calling the function, I added another line:

```
printk(KERN_ALERT "[Yusen] reading %ld bytes\n", count);
```

Inside the function `scull_write()`, in order to see how many bytes we have written, I added another line:

```
printk(KERN_ALERT "[Yusen] writing %ld bytes\n", count);
```

Inside the function `scull_release()`, I added one line of code before returning 0:

```
printk(KERN_ALERT "[Yusen] release!\n");
```

After those modifications, it is time to figure why I modified those functions. Firstly, inside the file *main.c*, we could see a struct called `file_operations`, as shown below:

```
struct file_operations scull_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .unlocked_ioctl = scull_ioctl,
    .open =     scull_open,
    .release =  scull_release,
};
```

A `file_operations` structure or a pointer to one is called `fops`. Each field in the structure must point to the function in the driver that implements a specific operation, or be left `NULL` for unsupported operations. For instance, the function

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

is used to retrieve data from the device. A null pointer in this position causes the *read* system call to fail with `-EINVAL` ("Invalid argument").

In conclusion, we are creating a character driver which looks like a file that implements functionality for typical system calls like *open()*, *read()* and *write()*.

After we execute the `$ make` command, we could run `scull_load` in sudo mode. Prior to running `$ sudo ./scull_load`, if we execute `$ ls /dev/scull*`, there will be no desired output. After running `$ sudo ./scull_load`, we run `$ ls /dev/scull*` again, we could see a bunch of *scull* devices loaded.

Next, we could execute the command `$ tail -f /var/log/syslog` to display the messages, and open another new terminal to run Python3 on it (in sudo mode). What we could do next is to open one of the *scull* devices and check the messages printed afterwards. The command is as follows:

```
>>> f = open("/dev/scull0", "w")
```

Now, we could see the new message from the original terminal:

```
vm kernel: [21096.510718] Hello world
```

When we executed `open()` in Python, Python eventually called the syscall `open`. Linux kernel noticed that we would like to open the device `/dev/scull0` and it looked into the `file_operations` structure, found the function `scull_open` that corresponds to the `open` syscall, and started to execute `scull_open`. That is why we could see the output on the kernel message.

Next, we run the following piece of code in Python3 to write something to the device `/dev/scull0` and then close the file:

```
>>> f.write("write something")
>>> f.close()
```

Something weird happened here. Because the function `scull_write()` should have been called, but there was no output until we closed the file. The output is as follows:

```
[Yusen] writing 15 bytes
[Yusen] release!
```

This is not an error, but some buffering, either in Python3 or in the Linux kernel. Calling `write()` does not mean Python3 would immediately call the syscall `write` and write to the destination. When we close the file, the bytes stored in the buffer would be flushed and written to the device. If we would like to see the result of writing to the device before closing the file, we could do the following:

```
>>> f.write("write something")
>>> f.flush()
```

This will lead to the single line output to the kernel message: `[Yusen] writing 15 bytes`.

Recall the structure of the `file_operations`, `scull_write` corresponds to the `write` syscall, and `scull_release` corresponds to the `release` syscall (which corresponds to `close()` in Python). This is like a step-by-step calling technique that connects the applications with hardware. It also demonstrates the encapsulation in the design of the computer; the applications will not worry about how to interact with the hardware. That is why we could write high-level codes and leave the rest to the operating system.

3 Discussion

Modifying the *scull* driver code helps me better understand the relationship between the operating system and the hardware. When we open a regular file, and would like to write something to it, it is basically done by the operating system that actually writes the data to the hardware. The kernel has to call a driver and let the driver handle the writes. The driver will then tell the hardware where to store the data to be written.

I would keep on to the example codes and try to explore as much as I could.

References

[1] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.