

CSIS0230A Principles of Operating Systems (Class A)

Tutorial 10

A simple char driver

In this tutorial, you will try to actually make a kernel module, and try to use it to implement a device driver. We will follow a step-by-step approach. A few places of the tutorial requires the use of the `root` account. For those machines that are designated for the tutorial, the root password is set to be the same as the password of the `os` account. You should use it only when actually needed (i.e., for those steps marked as [Root]).

Part 1: Modules

1. Write a simple module (in source file “`mymodule.c`”) as described in the notes. At this time, don’t register anything yet. Just try to print a message from the kernel module using `printk()`. Compile the kernel module. When compiling, you will need to use the flag `-I /usr/src/linux-2.4/include` to force the kernel header files to be used.
2. [Root] Now try to load and unload the kernel module. Note that no message appears on the screen. Switch to a text-mode console (using Control-Alt-F1), login as `root`, and try loading and unloading it again. Use `/sbin/lsmmod` to see what kernel module is currently loaded. If possible, continue to use the text mode for the rest of the tutorial. Our advice is to logout the graphical console now (you can switch to it using Control-Alt-F7), switch to virtual console 2 with Control-Alt-F2, login as `os` there, and edit your files there. You might want to log into one more virtual console for executing commands using the `os` account.
3. As an experiment, try changing to use the normal `printf` to do the printing, and include the normal `<stdio.h>` for it, in order to see how it doesn’t work. Remember that you are writing a part of the kernel program, not a special user program, when you write a module.
4. Note that when loading and unloading the kernel module, a complaint that the module doesn’t declare its license is printed. Once you are convinced that the kernel module loads and unloads correctly, use `MODULE_LICENSE` to add this information. Use `/sbin/modinfo` to show the information.

Part 2: Device

1. Modify the code to register a device named `mydevice` on loading and unregistering the device on unloading. Use an empty `file_operations` array for registration. Allow the system to choose any available major device number. Look at `/proc/devices` to make sure that the device is actually created.
2. [Root] Create two device files `mydev` and `mydev1` for the major number you see in `/proc/device`. Use 0 and 1 as the minor numbers. See the man page `mknod(1)` to see how to do this. Change their permission to 666 using `chmod`.
3. Use `cat < mydev` to read from the device, and use `cat > mydev` to write to the device. Note when error occurs (since we didn’t have implemented a read and write function yet).
4. Use `cat > mydev` again, and before an error occur, look at the module list again. See that the usage count of the device is still 0, although it is dangerous to remove the device driver at this time. Try to actually remove the module and type something to the program to see what will happen.

5. Now add the *owner* field to the *file_operations* structure. Repeat the above step to see how the usage count is managed automatically.

Part 3: device operations

1. Note that both devices currently work in exactly the same way. Write a function *mydev_open* according to the specification of the *open* method. If the minor number is not 0, deny the open request by returning *-ENODEV* (no such device). Add the method to the file operations structure. Test your modifications by trying to read and write the two devices.
2. Add a similar function *mydev_read* according to the specification of the *read* method. Currently, let the method always put the 13 characters *"Hello, world\n"* into the buffer provided using *copy_to_user*. Compile the module, and test it by reading the device again using *cat*.
3. Now try to modify the device driver so that it will output each character of *"Hello, world\n"* exactly once, rather than repeatedly. This can be done by carefully managing the *f_pos* argument of the *read* function. You should make sure that the number of characters being put into the buffer is at most *count*. Test your new module.
4. Now do the final test: run the user program stored and compiled in the computer, which looks like this:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <string.h>
int main() {
    char c;
    int ret;
    int fd = open("mydev", O_RDONLY);
    while ((ret = read(fd, &c, 1)) == 1)
        printf("%c", c);
    if (ret == -1) {
        printf("Error: %s\n", strerror(errno));
        return 1;
    }
    return 0;
}
```

Unlike the *cat* test, this reads the device character by character, and test whether you device driver correctly handles the *count* passed to it.

Note that we have done no real I/O in this tutorial. But we learn how different device drivers can be invoked when different device files are opened, each performing specific things in kernel mode. Once you can do this, it is relatively easy to actually read and write the I/O ports and memory in order to perform I/O.