

CSIS0230A Principles of Operating Systems (Class A)

Tutorial 6

A simple char driver

In this tutorial, you make a kernel module, and 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]). To switch to `root`, type `su -`. To go back, type `exit`.

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 print a message from the kernel module with `printk()`. Compile the kernel module. Use `/usr/src/linux-2.4.20-8` as the `<kroot>`.
2. [Root] Load and unload the kernel module, and note `printk()` show you the message. Use `/sbin/lsmmod` to read the loaded module list and check that the module is indeed loaded into the kernel.
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. Back out the changes when you finish.
4. Note that when loading the kernel module, `insmod` complains that the module doesn’t declare its license. Use `MODULE_LICENSE` to declare it “GPL”. Use `/sbin/modinfo` to show the information. Load and unload the kernel module again to see the effect. It is also a good time to comment out the `printk()` so that it won’t clutter up the screen too much.

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` structure 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/devices`. Use 0 and 1 as the minor numbers, and make the file mode to be 666. See the man page `mknod(1)` to see how to do this.
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 `cat` program that you have just run 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 modifying `*f_pos` after the `read` function by adding the number of character read, so that the next time when the `read` function is called, fewer characters are read. Test your new module.
4. Now do the final test: run the user program `testdev.cc` stored and compiled in the computer, which looks like this:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

void die(char *msg) {
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(1);
}

int main() {
    char c;
    int ret;
    int fd = open("mydev", O_RDONLY);
    if (fd == -1)
        die("open");
    while ((ret = read(fd, &c, 1)) == 1)
        printf("%c", c);
    if (ret == -1)
        die("read");
    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. If it segfaults, it means your kernel driver writes too much data to the user-provided buffer.

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.