New York University
Tandon School of Engineering
Department of Computer Science and Engineering

Introduction to Operating Systems
Fall 2024

Assignment 8 notes

How to invoke your driver/kernel module?

From your test application, you open the file "/dev/lab8". Of course you do this after your kernel module has been loaded already (and you should make sure the virtual file lab8 exists in /dev directory, using the `ls` command)

How do you load the kernel module?

You load your module just like in the prior driver assignment; using `insmod`
When done, you may unload your module using `rmmod`

Does the driver need to open a file? Is it associated with some file on my hard drive?

NO, there is no file involved with your module. Your driver appears as a virtual file to user-mode processes (applications + system processes)

In order to see a "lab8" entry (i.e. a virtual file) in the dev filesystem (i.e in /dev), your kernel module must've already registered itself with the devfs. When does it do that?

Your module (just like in the prior kernel module assignment) implements an init function and this is where it has to register itself as a misc device. If you register successfully (with the name lab8), then /dev/lab8 will appear, and your application can open it.

When your test application does an open system call or a read system call on that virtual file (/dev/lab8), then the calls get routed to your module. Thus, you need to implement an open function that obtains the user id and saves it, and also implement a read function that returns the user ID that was already saved, using the technique described in the assignment and the link provided in the assignment.

Can I return any number of bytes when implementing the read() function?

No, you should only return no more than the number of bytes the user told you in the third parameter of the read method (and that's the value you return in the read function). I included the file operations struct below, hiliting the read method and showing the third parameter in red. Note the user-mode program that invoked your application allocated a buffer of a certain size (possibly given in the third parameter) and you do not want to cause a buffer overflow by writing more number of bytes into that buffer (whose pointer is given in the second parameter).

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
          ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
            loff_t *);
          ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
            loff_t *);
    };
```

Where do I return the elapsed time and process ID?

You should first create a **c-string buffer** that can encapsulate all that information, let's call it the **message_buffer**. Then you copy information from that buffer to the one provided to you by the user-mode process (in the second parameter of read()).

How many bytes will the user-mode process try to read?

You don't know! It may try to ready 1, 10 or a 100. The user-mode application will know the end of file (EOF) condition is reached when you return a number smaller than the number of bytes it desired in the third parameter.

Note that your test program is not the only program that will use the driver. Myself or the TA's may also try to read from it using 1, 4, 10 or 100 bytes passed in the third parameter.

If my message_buffer contains 10 characters, and the user reads less than 10 characters, how do I keep track of what was read?

Using the last parameter "loff_t*", e.g. if the user only read 2, then you increment the value of that dereferenced parameter by 2.

You should return ascii characters:

That way, myself and the TA's may also test your module using the shell command `cat`.