CS5250 – Advanced Operating Systems

AY2018/2019 Semester 2

Assignment 3

Deadline: Friday, 29 Mar 2019 • 11.59pm

1. Objectives

1. Learn the basic knowledge about building a device driver

2. Rules

1. It is fine to ask for "reasonable" amount of help from others, but ensure that you do all the tasks on your own and write the report on your own. The University's policy on plagiarism applies here and any breaches will be dealt with severely.
2. **Generate your report as a pdf file, name it as *"Name (Student Number) Assignment 3.pdf* and upload your report in the IVLE folder Assignment-3 of CS5250. Violating any of the rules will incur a mark penalty, i.e., have a wrong file name, directly submit your code file into the folder rather than copy the codes in the report, or submit a .docx file instead of a .pdf file. We need to be strict here due to the number of submissions involved.**
3. The deadline of Assignment 3 is 29 Mar 2019. **Late assignments lose 4 marks per day.**
4. For any question, contact the teaching assistant Mr Chen Cheng, e0205030@u.nus.edu or Mr Vanchinathan Venkataramani, v.vanchinathan@u.nus.edu.

5. Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.

6. Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.

3. Assignment

Part A:

Task 1: Build and run modules

The Hello World Module :

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");
static int hello_init(void)
{
     printk(KERN_ALERT "Hello, world\n");
     return 0;
}
static void hello_exit(void)
{
     printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

You should search online for some Linux-module relevant sources to understand the code, build and run the module in your VM and answer the questions for Task 1.

Task 2: Build a device module

The first step of driver writing is defining the capabilities (the mechanism) the driver will offer to user programs. Our "device" is part of the computer's memory. The device we will build is

global and *persistent* one-byte char device. "Global" means that if the device is opened multiple times, the data contained within the device is shared by all the file descriptors that opened it. "Persistent" means that if the device is closed and reopened, data isn't lost. This device can be fun to work with, because it can be accessed and tested using conventional commands, such as `cp`, `cat`, and shell I/O redirection. "One byte" means that this device can only deal with one byte.

To figure out device drivers in your kernel, you can use the command "`ls -l /dev`" and check the meaning of every column. The first letter of every line can be "`c`" for char device or "`b`" for block device. The major and minor numbers are also shown. Traditionally, the major number identifies the driver associated with the device. The minor number is used by the kernel to determine exactly which device is being referred to.

Basically, the device driver needs to be registered before using and unregistered during exit. For simplicity, we will use a classic mode of registering/unregistering device. The structure of ***file_operation*** needs to be checked. The read and write function needs to be implemented and the structure of the code is given. Read function should read the context of the one-byte device. Write function should write one byte to the device. However, if there are more than 1 bytes written to the device at the same time, the first byte will be written and there will be an error message. If you are really familiar with fancier way of implementing a char driver, you can use other approaches.

Students are required to build or use their github accounts and sync their codes throughout the whole process of modifying codes and provide a screenshot of the commits. Remember to sync your codes every time you do some modification so there will be multiple commits. Contact the TA if you have any issue on this.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/errno.h>
```

```
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define MAJOR_NUMBER 61

/* forward declaration */
int onebyte_open(struct inode *inode, struct file *filep);
int onebyte_release(struct inode *inode, struct file *filep);
ssize_t onebyte_read(struct file *filep, char *buf, size_t
count, loff_t *f_pos);
ssize_t onebyte_write(struct file *filep, const char *buf,
size_t count, loff_t *f_pos);
static void onebyte_exit(void);

/* definition of file_operation structure */
struct file_operations onebyte_fops = {
     read:     onebyte_read,
     write:    onebyte_write,
     open:     onebyte_open,
     release: onebyte_release
};

char *onebyte_data = NULL;

int onebyte_open(struct inode *inode, struct file *filep)
{
     return 0; // always successful
}

int onebyte_release(struct inode *inode, struct file *filep)
{
     return 0; // always successful
}

ssize_t onebyte_read(struct file *filep, char *buf, size_t
count, loff_t *f_pos)
{
     /*please complete the function on your own*/
}
ssize_t onebyte_write(struct file *filep, const char *buf,
size_t count, loff_t *f_pos)
{
```

```
     /*please complete the function on your own*/
}

static int onebyte_init(void)
{
     int result;
     // register the device
     result = register_chrdev(MAJOR_NUMBER, "onebyte",
&onebyte_fops);
     if (result < 0) {
          return result;
     }
     // allocate one byte of memory for storage
     // kmalloc is just like malloc, the second parameter is
     // the type of memory to be allocated.
     // To release the memory allocated by kmalloc, use kfree.
     onebyte_data = kmalloc(sizeof(char), GFP_KERNEL);
     if (!onebyte_data) {
          onebyte_exit();
          // cannot allocate memory
          // return no memory error, negative signify a
     failure
          return -ENOMEM;
     }
     // initialize the value to be X
     *onebyte_data = 'X';
     printk(KERN_ALERT "This is a onebyte device module\n");
     return 0;
}

static void onebyte_exit(void)
{
     // if the pointer is pointing to something
     if (onebyte_data) {
          // free the memory and assign the pointer to NULL
          kfree(onebyte_data);
          onebyte_data = NULL;
     }
     // unregister the device
     unregister_chrdev(MAJOR_NUMBER, "onebyte");
     printk(KERN_ALERT "Onebyte device module is unloaded\n");
}

MODULE_LICENSE("GPL");
```

```
module_init(onebyte_init);
module_exit(onebyte_exit);
```

Remember to use **mknod** command to give your device a name before compiling and building the new module. After the module works normally, finish the questions in task 2.

The test cases are the following:

4. Tasks (20 marks)

Part A:

1. 12 marks, about 1 hour

   Answer the following questions:

   a. 2 marks. When will **module_init** and **module_exit** be loading/called?

   b. 3 marks. What is the command of building the module, installing the module and removing the module?

   Hint : there is no need for recompiling the whole kernel and rebooting.

   c. 3 marks. Give the screenshot of the previous three commands and their results if any in the shell. If the output of `printk` doesn't show in the shell, take a screenshot with '`dmesg | tail`' or any other command to show the `printk` of hello world module.

   d. 4 marks. Add a `<who>` parameter to your module so that your module will show `hello <who>` during init stage. Give the added lines which implements the function and give the screenshot of the new printk.

2. 8 marks, about 2 hours to 4 hours

   a. 1 mark. Give the mknod command you use.

   b. 1 marks. Give the screenshot of your device with "`ls -l /dev`" command and highlight your device.

   c. 6 marks. **Students are required to build or use their github accounts and sync their codes throughout the whole process of modifying codes and**

provide a screenshot of the commits. Give the codes of read and write functions that you implemented and the screenshots of the four testing cases.

Hint : Do not run other commands on your device before

```
cat /dev/<name>
```

Part B: (5 marks each, total 10 marks)

1. Let's assume that we have the following processes entering the system (in the given order):

- P1 burst CPU time: 23, arrival time 0
- P2 burst CPU time: 12, arrival time 5
- P3 burst CPU time: 41, arrival time 10
- P4 burst CPU time: 17, arrival time 15
- P5 burst CPU time: 29, arrival time 40

(a) What is the execution schedule and the completion time for each if we schedule the processes using pre-emptive shortest remaining time first with a context switching overhead of 1 time unit?

(b) What is the completion time for each task if we schedule the processes using round robin with a quantum of 10 with **no** overhead in context switching?

2. Draw the final red-black tree after the following insertions are completed. Use a single circle to represent a "red" node and a double concentric circle to represent a "black" node. If you use other notations, make sure you give a legend.

- Insert 112
- Insert 12
- Insert 304
- Insert 4
- Insert 55
- Insert 176
- Insert 64
- Insert 224
- Insert 24

Have fun!