CS353: Linux Kernel Project 2(a) Report

Weichen Li
5120309662
eizo.lee@sjtu.edu.cn
Department of Computer
Science and Engineering
School of Electronic Information
and Electrical Engineering
Shanghai Jiao Tong University

CONTENTS

I. OBJECT

A. Module 1

Load/Unload the module that it can output some info

B. Module 2

- Module accepts a parameter (an integer).
- Load the module, output the parameter's value.

C. Module 3

- Module creates a proc file.
- Read the proc file and return some info.

II. COMPILE STEPS

A. Create relevant files

At the very beginning, to save some trivial operations on authority verification, we need get the root authority first. (Need to input the key)

```
su
```

To start our work, we need to create a new directory to store our work files.

```
mkdir /usr/src/Project_2A
cd /usr/src/Project_2A
```

Then we create 3 *.c* files.

```
vi module_2A_1.c
...
vi module_2A_2.c
...
vi module_2A_3.c
```

Every time the "vi" command is called, the terminal will jump into the command window of VIM. We can simply input ":wq" and press "Enter" button to get back to the terminal.

Next we need to create a Makefile file:

```
vi Makefile
```

Do not quit the VIM in a hurry, we need to add some content in the Makefile file, press the “i” button on the keyboard first, then input:

```
obj-m := module_2A_1.o module_2A_2.o module_2A_3.o

KDIR := /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

all:

make -C $(KDIR) M=$(PWD) modules

clean:

rm *.o *.ko *.mod.c Module.symvers modules.order -f
```

Then press the “ESC” button, input “:wq”, press “Enter” button, the Makefile file is built. (This is the typical way to modify and save text files using VIM, “wq” means save and quit)

Next, input:

```
vi module_2A_1.c
```

Add content below , save and quit.

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

static int __init hello_init(void){
        printk("<3>Greeting from a linux kernel module.\n");
        return 0;
}

static void __exit hello_exit(void){
        printk("<3>Bye.\n");
}

module_init(hello_init);

module_exit(hello_exit);

MODULE_LICENSE("GPL");
```

The same way, add content below to “module_2A_2.c”:

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

static int test[3];
int num;
module_param_array(test,int,&num,0644);

void hello_foo(void){
        printk("Hello\n");
}

EXPORT_SYMBOL(hello_foo);

static int __init hello_init(void){
        printk(KERN_INFO"Hello world\n");
        printk(KERN_INFO"Params:test:%d,%d,%d;\n",test[0],test[1],test[2]);
        return 0;
}

static void __exit hello_exit(void){
        printk(KERN_INFO"Goodbye world\n");
}
```

```
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Test");
MODULE_AUTHOR("Blade");
module_init(hello_init);
module_exit(hello_exit);
```

Finally, add content below to “module_2A_3.c”:

```
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>

static int hello_proc_show(struct seq_file *m, void *v) {
        seq_printf(m, "I am Blade Lee!\n");
return 0;
}

static int hello_proc_open(struct inode *inode, struct  file *file) {
        return single_open(file, hello_proc_show, NULL);
}

static const struct file_operations hello_proc_fops = {
        .owner = THIS_MODULE,
        .open = hello_proc_open,
        .read = seq_read,
        .llseek = seq_lseek,
        .release = single_release,
};

static int __init hello_proc_init(void) {
        proc_create("hello_proc", 0, NULL, &hello_proc_fops);
        return 0;
}

static void __exit hello_proc_exit(void) {
        remove_proc_entry("hello_proc", NULL);
}

MODULE_LICENSE("GPL");
module_init(hello_proc_init);
module_exit(hello_proc_exit);
```

B. Compile and insert the modules

So far we have completed all the preparation for the experiment 2(a), next, input:

```
make
```

Then input:

```
insmod module_2A_1.ko
dmesg
```

We can see that module 1 has been successfully inserted into the kernel. (Shown in Figure 1)
Next, input:

```
insmod module_2A_2.ko test=1,2,3
dmesg
```

We can see that module 2 has been successfully inserted into the kernel. (Shown in Figure 2)
Finally, input:

```
insmod module_2A_3.ko
cat /proc/hello_proc
```

We can see that module 3 has been successfully inserted into the kernel. (Shown in Figure 3)
At this moment, we have successfully completed all the tasks for experiment 2(a), congratulations!
Tips: you can use “rmmod xxx.ko” command to remove module xxx from the kernel, or use “make clean” to delete all the files produced when compiling, then only the initial files left.

```
[   87.768242] Bluetooth: Core ver 2.20
[   87.768756] NET: Registered protocol family 31
[   87.768758] Bluetooth: HCI device and connection manager initialized
[   87.768761] Bluetooth: HCI socket layer initialized
[   87.768763] Bluetooth: L2CAP socket layer initialized
[   87.768859] Bluetooth: SCO socket layer initialized
[   87.861179] Bluetooth: BNEP (Ethernet Emulation) ver 1.3
[   87.861181] Bluetooth: BNEP filters: protocol multicast
[   87.861185] Bluetooth: BNEP socket layer initialized
[   88.324273] SELinux: initialized (dev tmpfs, type tmpfs), uses transition SID
s
[   93.465261] ISO 9660 Extensions: Microsoft Joliet Level 3
[   93.531161] ISO 9660 Extensions: RRIP_1991A
[   93.531276] SELinux: initialized (dev sr0, type iso9660), uses genfs_contexts
[  100.849514] nf_conntrack: automatic helper assignment is deprecated and it wi
ll be removed soon. Use the iptables CT target to attach helpers instead.
[  225.171764] SELinux: initialized (dev tmpfs, type tmpfs), uses transition SID
s
[ 1037.307339] SELinux: initialized (dev tmpfs, type tmpfs), uses transition SID
s
[ 1429.633626] module_2A_1: module verification failed: signature and/or  requir
ed key missing - tainting kernel
[ 1429.637479] <3>Greeting from a linux kernel module.
[root@localhost Project_2A]#
```

Fig. 1. Module 1

```
[   87.768758] Bluetooth: HCI device and connection manager initialized
[   87.768761] Bluetooth: HCI socket layer initialized
[   87.768763] Bluetooth: L2CAP socket layer initialized
[   87.768859] Bluetooth: SCO socket layer initialized
[   87.861179] Bluetooth: BNEP (Ethernet Emulation) ver 1.3
[   87.861181] Bluetooth: BNEP filters: protocol multicast
[   87.861185] Bluetooth: BNEP socket layer initialized
[   88.324273] SELinux: initialized (dev tmpfs, type tmpfs), uses transition SID
s
[   93.465261] ISO 9660 Extensions: Microsoft Joliet Level 3
[   93.531161] ISO 9660 Extensions: RRIP_1991A
[   93.531276] SELinux: initialized (dev sr0, type iso9660), uses genfs_contexts
[  100.849514] nf_conntrack: automatic helper assignment is deprecated and it wi
ll be removed soon. Use the iptables CT target to attach helpers instead.
[  225.171764] SELinux: initialized (dev tmpfs, type tmpfs), uses transition SID
s
[ 1037.307339] SELinux: initialized (dev tmpfs, type tmpfs), uses transition SID
s
[ 1429.633626] module_2A_1: module verification failed: signature and/or  requir
ed key missing - tainting kernel
[ 1429.637479] <3>Greeting from a linux kernel module.
[ 1737.938195] Hello world
[ 1737.938198] Params:test:1,2,3;
[root@localhost Project_2A]#
```

Fig. 2. Module 2

III. Reflection about the Experiment

During the process of this part of Experiment 2, I learned a lot about Linux kernel modules.

Additionally, I found something interesting on module 2, which required array parameters when inserting the module.

In the original version of module 2, I just established a 1-dimension array to obtain the parameters given by command line. I wondered what will happen when I set a 2-dimension array?

So I modified the $module_2A_2.c$ like below.

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

static int test[3][3];
int num;
module_param_array(test,int,&num,0644);
```

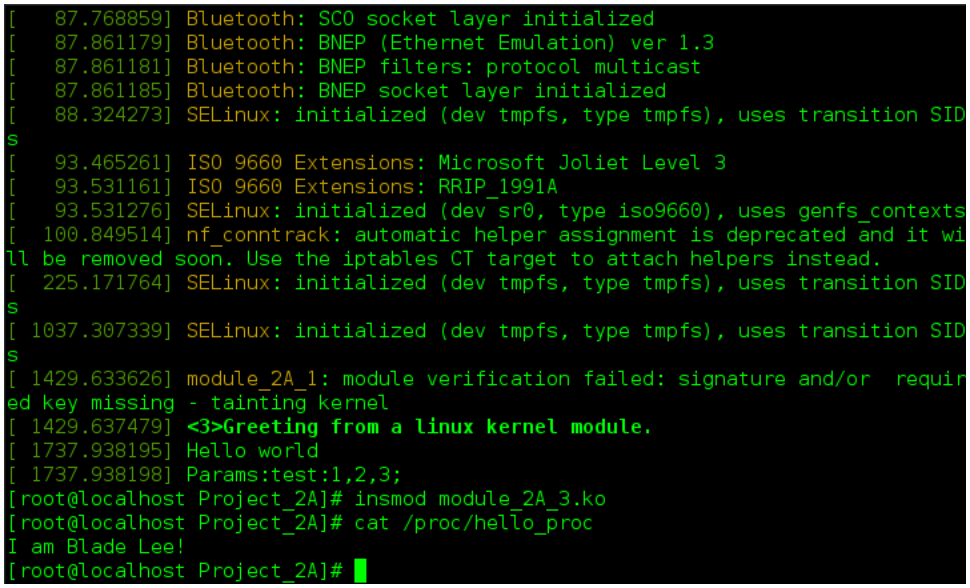

Fig. 3. Module 3

```
void hello_foo(void){
printk("Hello\n");
}

EXPORT_SYMBOL(hello_foo);

static int __init hello_init(void){
printk(KERN_INFO"Hello world\n");
printk(KERN_INFO"Params:test:%d,%d,%d,%d;\n",test[0][0],test[0][1],test[0][2],test[1][0]);
return 0;
}

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

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Test");
MODULE_AUTHOR("Blade");
module_init(hello_init);
module_exit(hello_exit);
```

The modification is that $test$ is a 3×3 matrix now, and I set the outputs as the three elements in the first row (test[0][0], test[0][1], test[0][2]) and the first element in the second row (test[1][0]).

In the command line, I input commands below after compiling:

```
insmod module_2A_2.ko test=1,2,3
```

And I got result shown in Figure 4:

As the result showed, the four elements mentioned above are 1, 0, 0 and 2. So I think the matrix of $test$ should look like this:

$$test = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix}$$

which means that the array definition in the Linux module may be in column-major order.

Though it is just a small discovery for me in the course Linux Kernel, I was inspired to discover more about the unknowns in further study.

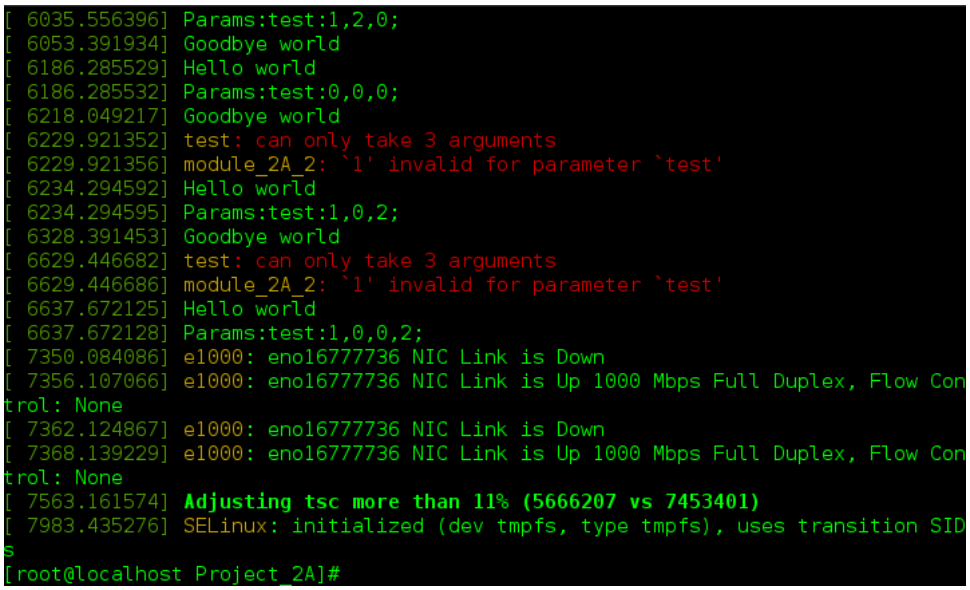

Fig. 4. Module 2 result after modification