Linux Kernel
Module Programming Experiment 1

518030910031 陳韋廷

1. Prints info when insmod and rmmod

a. Code

```c
#include <linux/init.h>   // macro
#include <linux/module.h> // needed for all module
#include <linux/kernel.h> // KERN_INFO

static int __init hello(void) {
  printk(KERN_INFO "*****HELLO*****\n");
  return 0;
}

static void __exit bye(void) {
  printk(KERN_INFO "*****BYEBYE*****\n");
}

module_init(hello); // hello will be executed upon insmod
module_exit(bye);   // bye will be executed upon rmmod
```

mod_1.c

```makefile
obj-m := mod_1.o
KERNEL := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
M_NAME := mod_1

all:
	make -C ${KERNEL} M=${PWD} modules

clean:
	make -C ${KERNEL} M=${PWD} clean

insert:
	insmod ${M_NAME}.ko

remove:
	rmmod ${M_NAME}
```

Makefile

b. Explanation

Use module_init() to register the initialization function and module_exit() to register the clean up function for this module.

c. Module in action

```
[  248.116404] *****HELLO*****
[  253.090575] *****BYEBYE*****
```

2. Module with parameters

a. Code

```c
#include <linux/init.h>        // macro
#include <linux/module.h>      // needed for all module
#include <linux/moduleparam.h> // module_param
#include <linux/kernel.h>      // KERN_INFO
// parameter holder
static int my_int = 0;
static char* my_string = "DEFAULT";
int my_int_array[5];
static int my_int_array_count = 0;
// parameter declaration
module_param(my_int, int, 0644);
module_param(my_string, charp, 0644);
module_param_array(my_int_array, int, &my_int_array_count, 0644);

static int __init hello(void) {
  printk(KERN_INFO "****[mod_2 is installed]****\n");
  printk(KERN_INFO "[mod_2] my_int = %d\n", my_int);
  printk(KERN_INFO "[mod_2] my_string = %s\n", my_string);
  printk(KERN_INFO "[mod_2] my_int_array_count = %d\n", my_int_array_count);
  int i = 0;
  for (i = 0; i < (sizeof my_int_array / sizeof (int)); ++i) {
    printk(KERN_INFO "[mod_2] my_int_array[%d] = %d\n", i, my_int_array[i]);
  }

  return 0;
}

static void __exit bye(void) {
  printk(KERN_INFO "****[mod_2 is removed]****\n");
}

module_init(hello);
module_exit(bye);
```

mod_2.c

```makefile
M_NAME := mod_2
obj-m := ${M_NAME}.o
KERNEL := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
	make -C ${KERNEL} M=${PWD} modules

clean:
	make -C ${KERNEL} M=${PWD} clean

insert:
	insmod ${M_NAME}.ko my_int=${my_int} my_string=${my_string} my_int_array=${my_int_array}

remove:
	rmmod ${M_NAME}
```

Makefile

b. Module in action

```
[  759.114271] ****[mod_2 is installed]****
[  759.114272] [mod_2] my_int = 12345
[  759.114273] [mod_2] my_string = TEST
[  759.114273] [mod_2] my_int_array_count = 5
[  759.114274] [mod_2] my_int_array[0] = 1
[  759.114274] [mod_2] my_int_array[1] = 2
[  759.114274] [mod_2] my_int_array[2] = 3
[  759.114275] [mod_2] my_int_array[3] = 4
[  759.114275] [mod_2] my_int_array[4] = 5
[  778.083344] ****[mod_2 is removed]****
```

make install my_int=12345 my_int_array=1,2,3,4,5 my_string=”TEST”

3. Read-only proc file

a. Code

```c
#include <linux/init.h>    // marco
#include <linux/module.h>  // needed for all modules
#include <linux/kernel.h>  // KERN_INFO
#include <linux/proc_fs.h> // proc filesystem
#include <linux/uaccess.h> // kernel space to user space

#define BUFSIZE 100

static struct proc_dir_entry* myproc;

static ssize_t mywrite(struct file* f, const char __user* ubuf, size_t count, loff_t* ppos) {
  printk(KERN_INFO "[mod_3] mywrite\n");
  return -EROFS;
}

static ssize_t myread(struct file* f, char __user* ubuf, size_t count, loff_t* ppos) {
  char buf[BUFSIZE];
  int len = 0;
  printk(KERN_INFO "[mod_3] myread\n");
  if (*ppos > 0 || count < BUFSIZE) return 0;
  len += sprintf(buf, "THIS IS MYPROC!!\n");
  if (copy_to_user(ubuf, buf, len)) return -EFAULT;
  *ppos = len;
  return len;
}

static struct file_operations myops = {
  .owner = THIS_MODULE,
  .read = myread,
  .write = mywrite
};

static int __init myinit(void) {
  printk(KERN_INFO "*****[mod_3 is installed]*****\n");
  myproc = proc_create("my_readonly_proc", 0600, NULL, &myops);
  printk(KERN_INFO "[mod_3] created /proc/my_readonly_proc\n");
  return 0;
}

static void __exit myexit(void) {
  proc_remove(myproc);
  printk(KERN_INFO "*****[mod_3 is removed]*****\n");
}

module_init(myinit);
module_exit(myexit);
```

mod_3.c

```
M_NAME := mod_3
obj-m := ${M_NAME}.o
KERNEL := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
  make -C ${KERNEL} M=${PWD} modules

clean:
  make -C ${KERNEL} M=${PWD} clean

insert:
  insmod ${M_NAME}.ko

remove:
  rmmod ${M_NAME}
```

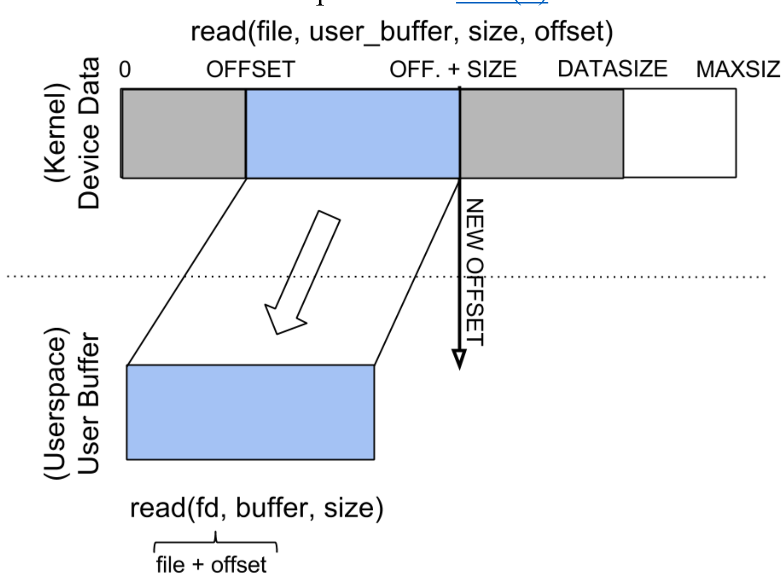
b. Explanation

i. mywrite

Returning -30 (-EROFS) indicates the file is read-only, as shown in the following section.

ii. myread

The following picture describes the relationship between read(2) and the read function in the module:



source: https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html

The following operation should be performed when a read is triggered:

- transfer maximum number of possible bytes between source and target
    a. sprintf() is used to determine how many bytes is written in to buf[]
    b. copy_to_user() is used to copy data from kernel space to user space buffer
- update the offset to where the next read data will begin (*ppos = len;)
- return the number of bytes transferred (return len;)

iii. myops

Register the read and write function of this module (proc file).

iv. initialization and clean up

Create the proc file using proc_create(), NULL indicates there's no parent folder for this file.
Remove the proc file using proc_remove().

c. Module in action

```
root@nick-ubuntu-vm:~# cat /proc/my_readonly_proc
THIS IS MYPROC!!
root@nick-ubuntu-vm:~# echo 1 > /proc/my_readonly_proc
-bash: echo: write error: Read-only file system
```

read / write test

```
[  317.926585] *****[mod_3 is installed]*****
[  317.926592] [mod_3] created /proc/my_readonly_proc
[  340.592601] [mod_3] myread
[  340.593311] [mod_3] myread
[  347.578426] [mod_3] mywrite
[  359.905812] *****[mod_3 is removed]*****
```

dmesg output

4. Readable / writable proc file inside a folder

a. Code

```
#include <linux/module.h>  // needed for all modules
#include <linux/kernel.h>  // kernel thing
#include <linux/proc_fs.h> // working with proc filesystem
#include <linux/slab.h>    // memory management
#include <linux/uaccess.h> // user <-> kernel space
int len, temp;
char* msg;
struct proc_dir_entry* parent;
static ssize_t myread(struct file* f, char __user* buf, size_t count, loff_t* pos) {
  printk(KERN_INFO "[mod_4] myread\n");
  if (count > temp) count = temp;
  temp = temp - count;
  printk(KERN_INFO "[mod_4] count = %d, temp = %d, len = %d\n", count, temp, len);
  copy_to_user(buf, msg, count);
  if (count == 0) temp = len;
  printk(KERN_INFO "[mod_4] count = %d, temp = %d, len = %d\n", count, temp, len);
  return count;
}
static ssize_t mywrite(struct file* f, const char __user* buf, size_t count, loff_t* pos) {
  printk(KERN_INFO "[mod_4] mywrite\n");
  copy_from_user(msg, buf, count);
  printk(KERN_INFO "[mod_4] count = %d, temp = %d, len = %d\n", count, temp, len);
  len = count;
  temp = len;
  return count;
}
struct file_operations myops = {
  .read = myread,
  .write = mywrite
};

static int __init myinit(void) {
  printk(KERN_INFO "*****[mod_4 is installed]*****\n");
  parent = proc_mkdir("myproc_folder", NULL);
  proc_create("myproc", 0600, parent, &myops);
  msg = kmalloc(GFP_KERNEL, 100*sizeof(char));
  printk(KERN_INFO "[mod_4] created /proc/myproc_folder/myproc\n");
  return 0;
}
static void __exit myexit(void) {
  proc_remove(parent);
  kfree(msg);
  printk(KERN_INFO "*****[mod_4 is removed]*****\n");
}

module_init(myinit);
module_exit(myexit);
```

mod_4.c

```
M_NAME := mod_4
obj-m := ${M_NAME}.o
KERNEL := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
  make -C ${KERNEL} M=${PWD} modules

clean:
  make -C ${KERNEL} M=${PWD} clean

insert:
  insmod ${M_NAME}.ko

remove:
  rmmod ${M_NAME}
```

Makefile

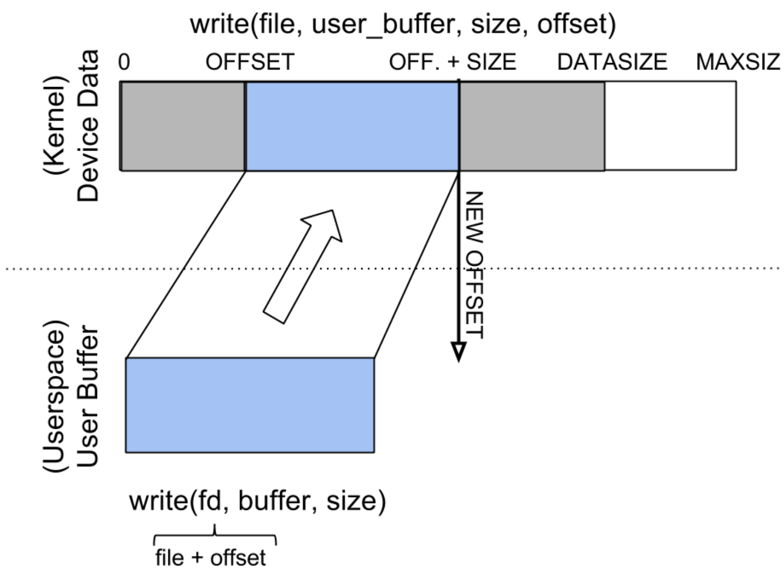
b. Explanation

i. myread

Same logic as the previous module, but this time global variables are used to share information with write function.

ii. mywrite

The following picture describes the relationship between [write(2)]() and the write function in the module:



The following operation should be performed when a write is triggered:

- transfer maximum number of possible bytes between source and target (copy_from_user())
- update readable length and new offset for next write (len=count; temp=len;)
- return the number of bytes written (return count;)

iii. initialization

This time proc_dir_entry* parent is pointed to folder created by proc_mkdir(), and the parent folder of create_proc() is pointed to it. Also, kmalloc() is used to allocate memory for char* msg.

iv. clean up

Use remove_proc() to remove the folder and its children. Also, kfree() is used to free the memory allocated during initialization.

c. Module in action

```
root@nick-ubuntu-vm:~/linux_kernel_module/mod_4# cat /proc/myproc_folder/myproc
root@nick-ubuntu-vm:~/linux_kernel_module/mod_4# echo TESTMSG > /proc/myproc_folder/myproc
root@nick-ubuntu-vm:~/linux_kernel_module/mod_4# cat /proc/myproc_folder/myproc
TESTMSG
```

read / write test, starting with an empty proc file

```
[  795.036112] *****[mod_4 is installed]*****
[  795.036117] [mod_4] created /proc/myproc_folder/myproc
[  801.506886] [mod_4] myread
[  801.506891] [mod_4] count = 0, temp = 0, len = 0
[  801.506893] [mod_4] count = 0, temp = 0, len = 0
[  811.711739] [mod_4] mywrite
[  811.711745] [mod_4] count = 8, temp = 0, len = 0
[  818.072383] [mod_4] myread
[  818.072388] [mod_4] count = 8, temp = 0, len = 8
[  818.072395] [mod_4] count = 8, temp = 0, len = 8
[  818.073070] [mod_4] myread
[  818.073073] [mod_4] count = 0, temp = 0, len = 8
[  818.073075] [mod_4] count = 0, temp = 8, len = 8
[  823.320682] *****[mod_4 is removed]*****
```

dmesg output

5. Final thoughts

Developers should be very careful when writing kernel modules, otherwise the system may crash during testing. I once forget to return 0 in the initialization function, which make insmod and rmmod dead (unable to insert / remove any module) and I have to restart the virtual machine to make things work again.

Also, intellicode in vscode doesn't seem to work very well when working with kernel modules. I never successfully configure it to recognize all included headers, which isn't a big problem because the code will compile if the compiler sees the included files, but it's annoying, I have to say.

The git repo can be found at https://git.sjtu.edu.cn/nickchen120235/linux-kernel-module.