# Linux Kernal Lab Report 1

## 一、实验内容

此次实验需要编写四个功能模块，分别是：

- 模块一，加载和卸载模块时在系统日志输出信息
- 模块二，支持整型、字符串、数组参数，加载时读入并打印
- 模块三，在/proc路径下创建只读文件
- 模块四，在/proc路径下创建文件夹，并创建一个可读可写的文件

## 二、实验简介

**（1）Linux模块**

内核模块是Linux内核向外部提供的一个插口，其全称为动态可加载内核模块（Loadable Kernel Module，LKM），简称为模块。Linux内核之所以提供模块机制，是因为它本身是一个单内核（monolithic kernel）。单内核的最大优点是效率高，因为所有的内容都集成在一起，但其缺点是可扩展性和可维护性相对较差，模块机制就是为了弥补这一缺陷。

**（2）内核编程**

内核编程相对于普通的编程来说，需要接触到的内容更加的低层。需要编程者对于操作系统的知识有一个较为深刻的了解。不仅要考虑到写出来的模块与操作系统的兼容性，还需要考虑进程之间是如何通信的，并且有时候还要求考虑如何管理硬件。最后，接口的稳定性必不可少。

## 三、实验过程与效果截图

### 1、编写Makefile文件

```
1  obj-m:=mod{x}.o
2  KERNAL_PATH:=/lib/modules/$(shell uname -r)/build
3  all:
4      make -C $(KERNAL_PATH) M=$(shell pwd) modules
5  clean:
6      make -C $(KERNAL_PATH) M=$(shell pwd) clean
```

- **第一行**代表需要编译的目标文件是mod{x}.o 其中x=1,2,3,4，对应此次实验的4个模块
- **第二行**表示用变量KERNAL_PATH保存make时转到的路径，用处是在执行make或者make all命令时**转到这个路径下并利用该路径下的Makefile文件进行编译**。
- **第三行**的"all："的用处是指出此次编译的内容。冒号后面代表依赖文件，此处为空表示没有依赖。
- **第四行**表示运行make或者make all命令时执行的shell命令。通过阅读**KERNAL_PATH路径下**的**Makefile**文件，我找到如下解释：

```
###
# External module support.
# When building external modules the kernel used as basis is considered
# read-only, and no consistency checks are made and the make
# system is not used on the basis kernel. If updates are required
# in the basis kernel ordinary make commands (without M=...) must
# be used.
#
# The following are the only valid targets when building external
# modules.
# make M=dir clean     Delete all automatically generated files
# make M=dir modules   Make all modules in specified dir
# make M=dir           Same as 'make M=dir modules'
# make M=dir modules_install
#                      Install the modules built in the module directory
#                      Assumes install directory is already created

# We are always building only modules.
```

“M=”后面接源文件所在路径。“modules”表示会对所有模块进行编译。

- **最后两行**是运行make clean命令时执行的shell操作。表示将编译过程中产生的所有目标文件都删除。

## 2、编写实验一的源文件：mod1.c

```c
//<module1.c>
//Test for installing and removing of module.
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
static int __init hello_init(void)
{
    printk(KERN_INFO "Module1 is READY!\n"); //输出为INFO级别，第6级
    // printk(KERN_EMERG "EMERGE -- Module1 is READY!\n"); //输出为EMERG级别，第0级
    return 0;
    //TODO: 加载模块时printk输出信息
}

static void __exit hello_exit(void)
{
    printk(KERN_INFO "Module1 is REMOVED!\n");
    //TODO: 卸载模块时printk输出信息
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("xianghui");
MODULE_DESCRIPTION("EXPERIMENT 1");
```

- 入口函数需要通过**static int __init xxxx(void)**来定义。**__init**是linux 内核编程的一个特殊宏，展开是一个gcc的扩展属性语法。

**作用：**通过把init函数限制在一个固定的section，一个作用是在启动时简单遍历section调用初始化函数即可，另外一个作用是在初始化完成后，可以马上释放该section所占空间给系统用（因为初始化函数通常只在系统启动后执行一次）。下图为源代码（init.h文件）：

```
/* These are for everybody (although not all archs will actually
   discard it in modules) */
#define __init      __section(".init.text") __cold  __latent_entropy __noinitretpoline
#define __initdata  __section(".init.data")
#define __initconst __section(".init.rodata")
#define __exitdata  __section(".exit.data")
#define __exit_call __used __section(".exitcall.exit")
```

- 出口函数通过**static void __exit xxxx(void)**来定义。下图是**__exit**的定义代码（init.h文件）：

```
#define __exit          __section(".exit.text") __exitused __cold notrace
```

- 输出部分采用**printk()**函数，该函数不会像printf()一样输出到命令行里，而是输出到系统log中，需要执行dmesg命令来进行查看。
- **MODULE_LICENCE("xxx")**必须加，表示该内核代码符合xxx标准，此处是采用**GPL**标准。
- 最后需要使用**module_init(xxx);** **module_exit(xxx);**来决定入口函数和出口函数分别是哪个。
- **实验1截图：**

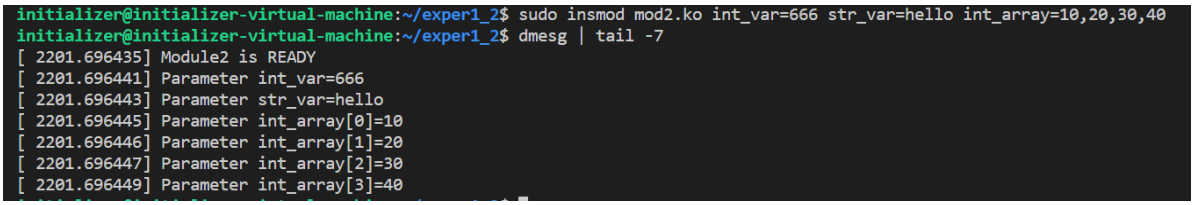
```
initializer@initializer-virtual-machine:~/exper1_1$ sudo insmod mod1.ko
initializer@initializer-virtual-machine:~/exper1_1$ sudo rmmod mod1
initializer@initializer-virtual-machine:~/exper1_1$ dmesg|tail -2
[66957.540613] Module1 is READY!
[66958.515796] Module1 is REMOVED!
initializer@initializer-virtual-machine:~/exper1_1$
```

## 3、编写实验二的源文件：mod2.c

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

static int int_var;
module_param(int_var, int, 0644); //权限是当前用户可读可写，其它用户可读
static char *str_var;
module_param(str_var, charp, 0644);
static int int_array[10];
static int num = 10;
module_param_array(int_array, int, &num, 0644);
int i = 0;
static int __init hello_init(void)
{
    printk(KERN_INFO "Module2 is READY\n");
    printk(KERN_INFO "Parameter int_var=%d\n", int_var);
    printk(KERN_INFO "Parameter str_var=%s\n", str_var);
    for (; i < num; ++i)
    {
        printk(KERN_INFO "Parameter int_array[%d]=%d\n", i, int_array[i]);
    }
    return 0;
}
static void __exit hello_exit(void)
{
    printk(KERN_INFO "Module2 is REMOVED!\n");
}
```

```
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("xianghui");
MODULE_DESCRIPTION("EXPERIMENT 2");
```

- 向模块内输入参数的做法是使用**module_param(name, type, perm)**这个函数。其中**name**对应输入的参数存储到的变量，**type**对应于该变量的类型，**perm**表示该参数的权限：所有人可读可写；还是仅当前用户可读可写、其它用户可读等等。
- 向模块内输入数组的做法是使用**module_param_array(name, type , &n_para , perm)**这个函数。其中相比于module_param(name, type, perm)函数只是多了一个 &n_para参数，用来传回用户输入数组参数时的数组长度。
- **实验2截图：**




**步骤：**1、加载模块时输入整型、字符型、数组参数。

2、利用dmesg来查看参数。

## 4、编写实验三的源文件：mod3.c

```
//<module3.c>
//read-only proc file
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/jiffies.h>
#include <linux/slab.h>

static struct proc_dir_entry *file = NULL;

static int hello_proc_show(struct seq_file *m, void *v)
{
    /* 这里不能使用printfk之类的函数，要使用seq_file输出的一组特殊函数 */
    seq_printf(m, "This is a proc message!\n");

    //必须返回0，否则什么也显示不出来
    return 0;
}

static int hello_proc_open(struct inode *inode, struct file *file)
{
    return single_open(file, hello_proc_show, NULL);
    //定义文件操作
}

const struct proc_ops hello_proc_fops = {
    // .owner = THIS_MODULE,
    .proc_open = hello_proc_open,
    .proc_release = single_release,
```

```
    .proc_read_iter = seq_read_iter,
    .proc_lseek = seq_lseek,
    //TODO: 指定文件操作
};
static int __init hello_proc_init(void)
{

    file = proc_create("hello_proc", 0400, NULL, &hello_proc_fops);
    printk(KERN_INFO "/proc/hello_proc has been created!\n");
    return 0;
    //TODO: 加载模块时printk输出信息
}
static void __exit hello_proc_exit(void)
{
    proc_remove(file);
    printk(KERN_INFO "Module3 is REMOVED!\n");
    //TODO: 卸载模块时printk输出信息，删除创建的proc文件
}
module_init(hello_proc_init);
module_exit(hello_proc_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("xianghui");
MODULE_DESCRIPTION("EXPERIMENT 3");
```

- 使用proc_create(name, mode, parent, proc_ops)接口来创建文件。其中name参数表示文件名，mode参数表示创建的文件的权限（由于实验要求创建只读文件，因此不会给写权限），parent表示**父文件夹**的proc_dir_entry对象，最后的proc_ops代表创建文件进行操作。由于创建只读文件，因此只需要自己写.proc_open函数即可。proc_release是用来释放内存的操作，此处直接调用seq_file.h文件中给的single_release接口，来释放顺序文件的内存。同样的，利用seq_read_iter来为hello_proc_fops提供读顺序文件的迭代器。proc_lseek用来寻找文件开始的偏移量，同样可以利用seq_lseek接口来定义。
- **hello proc open**实现方法：利用接口single_open，它只有**show**函数需要参数提供，而start，stop， next 这三个用的是默认接口single_xxx。调用顺序为：start->show->next->...->stop。可以看见single_open 只会调用一次show函数。因此主要实现部分就是show函数。该函数调用了seq_printf（）接口，从而让顺序文件在被读时出现参数中的内容。
- **实验三截图：**

```
initializer@initializer-virtual-machine:~/exper1_3$ sudo cat /proc/hello_proc
cat: /proc/hello_proc: No such file or directory
initializer@initializer-virtual-machine:~/exper1_3$ sudo insmod mod3.ko
initializer@initializer-virtual-machine:~/exper1_3$ sudo cat /proc/hello_proc
This is a proc message!
initializer@initializer-virtual-machine:~/exper1_3$ sudo rmmod mod3
initializer@initializer-virtual-machine:~/exper1_3$ sudo cat /proc/hello_proc
cat: /proc/hello_proc: No such file or directory
initializer@initializer-virtual-machine:~/exper1_3$
```

**步骤：**1、第一次读取/proc/hello_proc时，发现文件不存在。

2、然后加载模块，再次读取则会看到该文件以及文件中的内容。

3、随后删除模块，再次读取发现文件已经被删除。

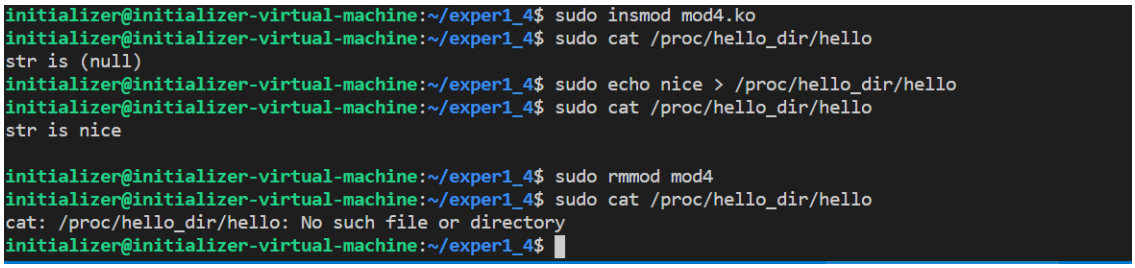
## 5、编写实验三的源文件：mod4.c

```
//<module4.c>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/jiffies.h>
```

```c
#include <linux/slab.h>
#include <asm/uaccess.h>

static char *str = NULL;
static struct proc_dir_entry *file = NULL;
static struct proc_dir_entry *director = NULL;

static int hello_proc_show(struct seq_file *m, void *v)
{
    /* 这里不能使用printfk之类的函数，要使用seq_file输出的一组特殊函数 */
    seq_printf(m, "str is %s\n", str);

    //必须返回0，否则什么也显示不出来
    return 0;
}

static int hello_proc_open(struct inode *inode, struct file *file)
{
    return single_open(file, hello_proc_show, NULL);
    //定义文件操作
}

static ssize_t
hello_proc_write(struct file *file, const char __user *buffer, size_t count,
loff_t *f_pos)
//buffer表示要写入的缓冲区，count表示要写入的信息长度,f_pos为当前的偏移位置，这个值通常是用来判断写文件是否越界
//如果返回值非负，则代表成功写的字节数。如果写入失败，则返回-EFAULT
{
    char *tmp = kzalloc((count + 1), GFP_KERNEL);
    if (!tmp)
        return -ENOMEM;
    if (copy_from_user(tmp, buffer, count))
    {
        kfree(tmp);
        return -EFAULT;
    }
    if (str)
        kfree(str);
    str = tmp;
    return count;
}

const struct proc_ops hello_proc_fops = {
    // .owner = THIS_MODULE,
    .proc_open = hello_proc_open,
    .proc_release = single_release,
    .proc_lseek = seq_lseek,
    .proc_read_iter = seq_read_iter,
    .proc_write = hello_proc_write,
    //TODO: 指定文件操作
};

static int __init hello_proc_init(void)
{
    director = proc_mkdir("hello_dir", NULL);
    file = proc_create("hello", 0777, director, &hello_proc_fops);
    printk(KERN_INFO "/proc/hello_dir/hello has been created!\n");
```

```
    return 0;
    //TODO: 加载模块时printk输出信息
}
static void __exit hello_proc_exit(void)
{
    proc_remove(file);
    proc_remove(director);
    printk(KERN_INFO "Module4 is REMOVED!\n");
    //TODO: 卸载模块时printk输出信息，删除创建的proc文件
}
module_init(hello_proc_init);
module_exit(hello_proc_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("xianghui");
MODULE_DESCRIPTION("EXPERIMENT 4");
```

- 相比于模块3，模块4多了一个创建文件夹、删除文件夹、写文件的函数。前面两个分别调用了proc_mkdir(name, parent)接口和proc_remove(proc_dir_entry)接口。
- 写文件的函数为hello_proc_write(file, buffer, count, f_pos)，四个参数分别代表：被写入文件的proc_dir_entry，存有写入的数据的缓存buffer，要写入的信息长度count，当前的偏移位置f_open。该函数的思想为先申请一块大小为(count + 1)的临时内存tmp，然后将buffer中的内容利用copy_from_user来传给这块临时内存，最终让文件的show函数中输出的str指针指向这块内存。
- **实验四截图：**




**步骤**：1、加载模块后，读取创建好的/proc/hello_dir/hello文件，里面的str指针为空。

2、然后利用echo将nice字符串写入文件中。

3、再次读取发现文件中内容变为str is nice。

4、删除模块后文件也随之被删除。

# 四、反思与总结

**思考：**

1、对于操控模块进行操作，由于是直接对内核的代码进行增改，因此用户常常具有较高的权限。而这时候如果不对操作者加以限制，常常会产生系统安全问题。比如写操作时，如果不对输入的数据大小加以限制，则可能会出现缓冲区溢出的危险，这常常是黑客发现漏洞的绝佳来源。所以为了避免缓冲区溢出，我的做法是对输入的数据大小在传给缓冲区时就做一个判断，让这个数据量大小不会超过某个特定值，这样就避免了缓冲区溢出的危险。

2、如果在卸载模块时没有将模块生成的文件删除，那么很可能再也没法删除掉。这个问题是让我不断回退虚拟机快照的罪魁祸首。而我对其的猜测是这个文件是某个模块产生，那么只有这个模块拿到了该文件的文件描述符，才能对其有操作。如果模块卸载，这个文件描述符也没有被保存到系统中，很可能就像野指针一样被丢失了。因此这个文件很难再被删除掉。

**总结：**

这次实验是我第一次接触到Linux内核源码与Linux模块的安装与卸载。这让我有了较为新奇的感受。刚刚了解到Linux模块的时候，我不禁有种熟悉的感觉。转念一想，发现这难道不是类似插件的理念吗？后来又想到，按照时间线来说，Linux模块才应该是插件的前辈。

在实验过程中，最常见的问题是不知道某个功能该使用内核提供的哪些接口函数。除了在网上对其搜索之外，我更喜欢直接到内核代码中查看相关的源码，找到从名字和描述上看比较接近我需要的功能的代码。这个过程虽然是一个艰难的过程，但是对我的能力有着极大的提高，我自己也乐在其中。