

# H1 CS353 Linux内核 Project1 报告

517030910214 刘宏洲

## H2 0. 简介

本Project的主要内容是熟悉Linux内核的模块编程以及proc文件系统。按照要求，完成以下四个模块：

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

本次Project的环境配置如下：

- Ubuntu 18.04 LTS with Linux 5.5.8
- GNU Make 4.1
- gcc 7.5.0

## H2 1. 模块一

### H3 1.1 实现

在本模块中，主要熟悉模块初始化函数、退出函数的编写及注册。主要使用了在linux/module.h中定义的module\_init()以及module\_exit()两个宏来注册这两个函数。代码如下：

```
1  #include <linux/kernel.h>
2  #include <linux/module.h>
3  #include <linux/init.h>
4
5  static int __init M1_init(void)
6  {
7      printk(KERN_INFO "Hello Linux Kernel!\n");
8      return 0;
9  }
10
11 static void __exit M1_exit(void)
12 {
13     printk(KERN_INFO "Goodbye Linux Kernel!\n");
14 }
15 MODULE_LICENSE("GPL");
16 MODULE_DESCRIPTION("Module2");
17 MODULE_AUTHOR("Hongzhou Liu");
18 module_init(M1_init);
19 module_exit(M1_exit);
```

### H3 1.2 结果

编译模块后使用insmod插入module1.ko，利用dmesg查看系统日志（以上命令均写入脚本test1.sh）即可看到实验结果：

```

dean@dean ~/CS353/A1/M1 <master>
$ ./test1.sh
[ 1480.084565] Hello Linux Kernel!
[ 1480.103531] Goodbye Linux Kernel!

```

图1. 模块一 结果

## H2 2. 模块二

### H3 2.1 实现

本模块需要实现模块初始化时传入参数的功能。必须使用 `linux/moduleparam.h` 中定义的宏 `module_param()` 注册参数，也可使用 `module_param_array()` 注册数组参数。而 `module_param_array()` 注册的是定长数组，如果输入数组元素超出限制，则会导致输入错误。还可以使用 `char*` 类型的参数实现“伪”不定长数组。以下是参数的注册代码：

```

1 static int int_param;
2 static char* str_param;
3 static char* arr_param_fake;
4 static int arr_num_fake = 0;
5 static int* arr_fake;
6 static int arr_param[5];
7 static int arr_argc = 0;
8 module_param(int_param, int, 0644);
9 module_param(str_param, charp, 0644);
10 module_param_array(arr_param, int, &arr_argc, 0644);
11 module_param(arr_param_fake, charp, 0644);

```

其中 `arr_param_fake`、`arr_num_fake` 以及 `arr_fake` 分别是“不定长数组”的字面值、长度以及指向真实数据的指针，后两者将在初始化函数中计算得到。需要注意的是，如果参数是字符串，注册时宏的类型参数应该传入 `charp`，而 `module_param_array` 还需要传入一个指针 `arr_argc` 表示实际得到的数组元素个数。传入参数后即可在初始化函数 `M2_init()` 中打印，而“不定长数组”则需要转换为 `int` 类型的数组，这个转换的实现如下：

```

1 static int __init M2_init(void)
2 {
3     int i = 0, tmp = 0;
4     char* cp = arr_param_fake;
5     .....
6     if(arr_param_fake)
7     {
8         arr_fake = (int*)kmalloc_array(strlen(arr_param_fake), sizeof(int),
9 GFP_KERNEL);
10         while(*cp)
11         {
12             while(*cp >= '0' && *cp <= '9')
13             {
14                 tmp *= 10;
15                 tmp += *cp - '0';
16                 cp++;
17             }
18             arr_fake[arr_num_fake++] = tmp;
19         }
20     }
21 }

```

```

18     tmp = 0;
19     cp++;
20 }
21 .....
22 kfree(arr_fake);
23 }
24 else
25 {
26     printk(KERN_INFO "arr_fake = (null)\n");
27 }
28 return 0;
29 }

```

首先要确定是否传入了“不定长数组”，若忽略此判断，不传入这个参数，就会解引空指针造成内核panic。如果传入了参数，需要为数组申请一块内核空间。`kmalloc`是定义在`linux/slab.h`中的函数，用于申请内核空间。保险起见，我们申请一块足够容纳所有元素的空间，然后将字符串中的每个代表数字的子串转换为int类型的数据放入数组即可。打印过程省略，打印过后使用`kfree`释放这一空间即可。

其他的打印语句、模块退出时的函数以及初始化、退出函数的注册便不再赘述。

### H3 2.2 结果

测试模块二的命令如下（摘取自test2.sh）：

```

1  sudo -S insmod module2.ko int_param=10 str_param=hello arr_param=1,2,3
   arr_param_fake=1,2,3,4,5,6,7,8,9,10,11,12,13
2  sudo -S rmmod module2
3
4  sudo -S insmod module2.ko int_param=20 arr_param=4,5,6,7,8
5  sudo -S rmmod module2

```

使用`dmesg`打印系统日志可以看见结果：

```

dean@dean ~/CS353/A1/M2 <master>
$ ./test2.sh
[ 4488.769175] int_param = 10
[ 4488.769180] str_param = hello
[ 4488.769182] arr_param[0] = 1
[ 4488.769184] arr_param[1] = 2
[ 4488.769186] arr_param[2] = 3
[ 4488.769187] arr_param[3] = 0
[ 4488.769189] arr_param[4] = 0
[ 4488.769191] got 3 input array elements
[ 4488.769194] arr_fake[0] = 1
[ 4488.769196] arr_fake[1] = 2
[ 4488.769198] arr_fake[2] = 3
[ 4488.769199] arr_fake[3] = 4
[ 4488.769201] arr_fake[4] = 5
[ 4488.769203] arr_fake[5] = 6
[ 4488.769205] arr_fake[6] = 7
[ 4488.769208] arr_fake[7] = 8
[ 4488.769210] arr_fake[8] = 9
[ 4488.769212] arr_fake[9] = 10
[ 4488.769214] arr_fake[10] = 11
[ 4488.769215] arr_fake[11] = 12
[ 4488.769217] arr_fake[12] = 13
[ 4488.769219] got 13 input array elements using char* as input
[ 4488.789401] Exiting Module2...

```

图2. 模块二 结果1

```

[ 4488.843780] int_param = 20
[ 4488.843783] str_param = (null)
[ 4488.843785] arr_param[0] = 4
[ 4488.843787] arr_param[1] = 5
[ 4488.843788] arr_param[2] = 6
[ 4488.843789] arr_param[3] = 7
[ 4488.843790] arr_param[4] = 8
[ 4488.843792] got 5 input array elements
[ 4488.843793] arr_fake = (null)
[ 4488.863501] Exiting Module2...

```

图3. 模块二 结果2

可见，对于字符串参数，若不传入参数，则自动显示 `(null)`，而数组参数没有元素的位置自动初始化为0。而“不定长数组”参数通过传入字符串再转换的方式，也实现了相应的功能。

## H2 3. 模块三

### H3 3.1 实现

在本模块中，需要在 `/proc` 文件夹下创建一个只读文件。调用在 `linux/proc_fs.h` 头文件中定义的 `proc_create()` 函数可以返回一个指向 `proc_dir_entry` 结构体的指针，如果指针非空，表明在指定路径创建了一个文件。具体的创建方式如下：

```

1 struct proc_dir_entry *entry = NULL;
2 static int __init M3_init(void)
3 {
4     entry = proc_create("M3_proc", 0444, NULL, &proc_fops);

```

```

5     if(!entry)
6     {
7         printk(KERN_ERR "Unable to create /proc/M3_proc\n");
8         return -EINVAL;
9     }
10    printk(KERN_INFO "/proc/M3_proc successfully created\n");
11    msg = "Hello /proc!\n";
12    strcpy(proc_buf, msg);
13    proc_buf_size = strlen(msg);
14    return 0;
15 }

```

`proc_create()` 的第一个参数是文件名，第二个参数为访问权限，`0444` 代表无论根用户、拥有者和其用户组都只有读的权限，第三个参数代表父文件夹对应的 `proc_dir_entry` 指针，而这里父文件夹为 `/proc`，则为 `NULL`。最后一个参数是指向 `file_operations` 结构体的指针，在这个结构体中定义操作文件是需要调用的函数。这里我们需要注册读文件和写文件时调用的函数：

```

1 struct file_operations proc_fops = { .read = read_proc, .write = write_proc };

```

而读文件时，我们像将内核中的信息显示给用户看。在模块初始化时，我们已经将 `proc_buf` 填入了信息，并计算了对应的 `proc_buf_size`。我们只需利用 `linux/uaccess.h` 中定义的 `copy_to_user()` 函数，将数据拷贝至用户空间的缓存 `usr_buf` 即可。值得注意的是，若 `read_proc()` 的返回值不为 0，那么它将被一直调用。但如果第一次调用即返回 0，那么用户也不会看到输出。因此，我们需要控制此函数恰好被调用 2 次，第一次返回拷贝至用户空间的数据大小，第二次返回 0。利用 `static` 变量的特性，引入 `finished` 变量即可实现这个功能。

```

1 static ssize_t read_proc(struct file *filp, char *usr_buf, size_t count, loff_t
  *offp)
2 {
3     static int finished = 0;
4     if(finished)
5     {
6         printk(KERN_INFO "read_proc: END\n");
7         finished = 0;
8         return 0;
9     }
10    finished = 1;
11    if(copy_to_user(usr_buf, proc_buf, proc_buf_size))
12    {
13        printk(KERN_ERR "Copy to user unfinished\n");
14        return -EFAULT;
15    }
16    printk(KERN_INFO "read_proc: read %lu bytes\n", proc_buf_size);
17    return proc_buf_size;
18 }

```

具体的 `write_proc()` 函数在下节详细介绍，这里只是为了体现创建的文件的只读性质。最后我们还需要在 `M3_exit()` 函数中调用 `proc_remove(entry)` 删除文件。

### H3 3.2 结果

编译模块后执行下列命令，可以看到M3\_proc是一个只读文件，普通用户不可写入（即使定义了写入时调用的函数）

```
dean@dean ~/CS353/A1/M3 <master*>
$ sudo insmod module3.ko
dean@dean ~/CS353/A1/M3 <master*>
$ cat /proc/M3_proc
Hello /proc!
dean@dean ~/CS353/A1/M3 <master*>
$ echo 1 > /proc/M3_proc
zsh: 权限不够: /proc/M3_proc
dean@dean ~/CS353/A1/M3 <master*>
$ sudo rmmod module3
dean@dean ~/CS353/A1/M3 <master*>
$ dmesg | tail -4
[27683.862641] /proc/M3_proc successfully created
[27687.453066] read_proc: read 13 bytes
[27687.453073] read_proc: END
[27700.701830] Exiting Module3...
```

图4. 模块三 结果

实验过程中还发现一个现象，若执行以下指令：

```
1 sudo echo 1 > /proc/M3_proc
```

无论文件的读写权限是0444还是0644，均提示权限不足。而0644权限下，超级用户应该可以写入。查询资料发现sudo指令只对离它最近的命令起作用，而管道>也算作命令，因此使用sudo命令作为超级用户写入应该执行：

```
1 sudo sh -c "echo 1 > /proc/M3_proc"
```

这时候，处于0444权限下的文件居然也可以被写入了。根据网上的资料，即使root权限为0，超级用户下，也可以对文件进行读写。

## H2 4. 模块四

### H3 4.1 实现

在本模块中需要在/proc文件夹下建立一个文件夹，然后在文件夹中创建一个可读可写的文件。

首先，我们需要利用proc\_mkdir()函数创建文件夹，再利用proc\_create()函数创建文件。在M4\_init()函数中：



```

1  base = proc_mkdir("M4_proc_dir", NULL);
2  if(!base)
3  {
4      printk(KERN_ERR "Unable to create /proc/M4_proc_dir/\n");
5      return -EINVAL;
6  }
7  entry = proc_create("M4_proc", 0666, base, &proc_fops);
8  if(!entry)
9  {
10     printk(KERN_ERR "Unable to create /proc/M4_proc_dir/M4_proc\n");
11     proc_remove(base);
12     return -EINVAL;
13 }

```

`proc_mkdir()` 函数只需指定文件夹名称以及父文件夹的 `proc_dir_entry` 结构体指针即可。`proc_create()` 函数此时需要把父文件夹指针设置为 `base`，并指定权限为 0666。为了实现向文件中写数据的功能，定义写函数：

```

1  static ssize_t write_proc(struct file *filp, const char *usr_buf, size_t count,
2  loff_t *offp)
3  {
4      proc_buf_size = count;
5
6      if(copy_from_user(proc_buf, usr_buf, proc_buf_size))
7      {
8          printk(KERN_ERR "Copy from user unfinished\n");
9          return -EFAULT;
10     }
11     printk(KERN_INFO "write_proc: write %lu bytes\n", proc_buf_size);
12     return proc_buf_size;
13 }

```

其中，`proc_buf` 是一个定长数组，它的最大长度为 `MAX_BUF_SIZE`，实际长度为 `proc_buf_size`。在写文件时，`write_proc()` 函数会收到来自用户的数据 `usr_buf` 以及数据的长度 `count`。简单的想法就是利用 `copy_from_user()` 函数，将用户数据拷贝到内核中来。

编写以下测试脚本：

```

1  sudo -S insmod module4.ko
2  cat /proc/M4_proc_dir/M4_proc
3  echo 1 > /proc/M4_proc_dir/M4_proc
4  cat /proc/M4_proc_dir/M4_proc
5  echo helloworldhelloworld > /proc/M4_proc_dir/M4_proc
6  cat /proc/M4_proc_dir/M4_proc
7  sudo -S rmmod module4.ko

```

运行，发现出现错误。错误原因是在第二次写入文件时，缓冲区溢出了。打印内核日志发现确实如此。

```
dean@dean ~/CS353/A1/M4-Exp <master*>
$ ./test4.sh
Hello /proc!
1
sh: echo: I/O error
cat: /proc/M4_proc_dir/M4_proc: 错误的地址
```

图5. 模块四 缓冲区溢出1

```
[ 5663.450983] Buffer overflow detected (16 < 21)!
[ 5663.451024] WARNING: CPU: 5 PID: 16311 at ./include/linux/thread_info.h:134 read_proc+0x92/0xc3 [module4]
```

图6. 模块四 缓冲区溢出2

为了解决这一问题，我们判断用户输入的长度是否超过了内核中 `proc_buf` 的最大长度，如果超过，则只写入前缀部分，并在内核日志中输出警告：

```
1  if(count > MAX_BUF_SIZE)
2  {
3      proc_buf_size = MAX_BUF_SIZE;
4      printk(KERN_WARNING "write_proc: overflow detected\n");
5      printk(KERN_WARNING "write_proc: input size %lu\n", count);
6      printk(KERN_WARNING "write_proc: buffer size %lu\n", proc_buf_size);
7  }
8  else
9  {
10     proc_buf_size = count;
11 }
```

再次运行测试脚本，溢出的问题被解决了。但由于第二次写入超过了 `proc_buf` 的大小，用户缓冲区中的数据没有被读完，因此 `write_proc()` 再一次被调用，原来写入的数据被覆盖了。

```
dean@dean ~/CS353/A1/M4-Exp <master*>
$ ./test4.sh
Hello /proc!
1
orld
[ 5534.832517] /proc/M4_proc_dir/M4_proc successfully created
[ 5534.836965] read_proc: read 13 bytes
[ 5534.836994] read_proc: END
[ 5534.837401] write_proc: write 2 bytes
[ 5534.839389] read_proc: read 2 bytes
[ 5534.839414] read_proc: END
[ 5534.839824] write_proc: overflow detected
[ 5534.839827] write_proc: input size 21
[ 5534.839829] write_proc: buffer size 16
[ 5534.839830] write_proc: write 16 bytes
[ 5534.839835] write_proc: write 5 bytes
[ 5534.841657] read_proc: read 5 bytes
[ 5534.841681] read_proc: END
[ 5534.857913] Exiting Module4...
```

图7. 模块四 写覆盖

为了解决这个问题，我们直接在输入长度超出 `proc_buf` 最大长度时将它增长以容纳所有数据。我们将 `proc_buf` 改为动态数组，在 `M4_init()` 中申请，在 `M4_exit()` 中释放。而出现缓冲区即将溢出的情况时再释放、重新申请：



```

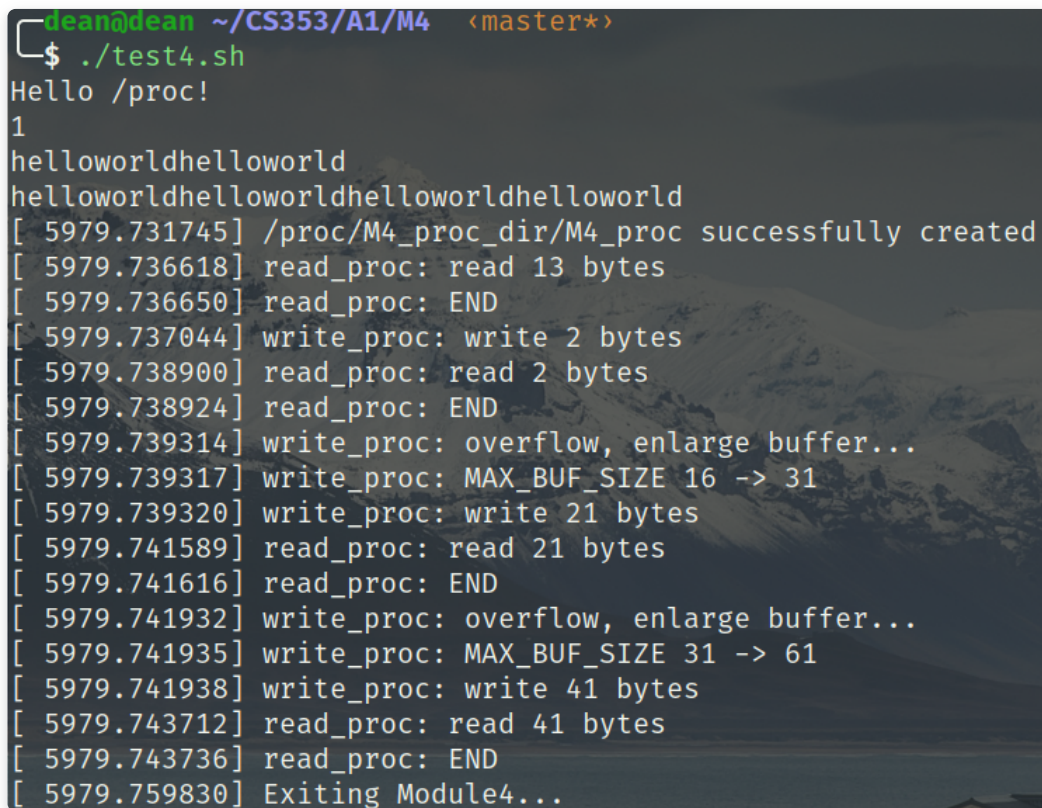
1  if(count > MAX_BUF_SIZE)
2  {
3      printk(KERN_WARNING "write_proc: overflow, enlarge buffer...\n");
4      printk(KERN_WARNING "write_proc: MAX_BUF_SIZE %d -> %ld\n",
5              MAX_BUF_SIZE, count + count / 2);
6      MAX_BUF_SIZE = count + count / 2;
7      kfree(proc_buf);
8      proc_buf = (char*)kmalloc(MAX_BUF_SIZE * sizeof(char), GFP_KERNEL);
9  }
10 proc_buf_size = count;

```

这样就解决了以上问题。

### H3 4.2 结果

解决了缓冲区溢出问题和写覆盖问题后，我们向文件写入超出缓冲区原大小的数据时，也能获得正确的结果。观察系统日志就可以看出，缓冲区的最大长度也是在不断变化的。



```

dean@dean ~/CS353/A1/M4 <master*>
$ ./test4.sh
Hello /proc!
1
helloworldhelloworld
helloworldhelloworldhelloworldhelloworld
[ 5979.731745] /proc/M4_proc_dir/M4_proc successfully created
[ 5979.736618] read_proc: read 13 bytes
[ 5979.736650] read_proc: END
[ 5979.737044] write_proc: write 2 bytes
[ 5979.738900] read_proc: read 2 bytes
[ 5979.738924] read_proc: END
[ 5979.739314] write_proc: overflow, enlarge buffer...
[ 5979.739317] write_proc: MAX_BUF_SIZE 16 -> 31
[ 5979.739320] write_proc: write 21 bytes
[ 5979.741589] read_proc: read 21 bytes
[ 5979.741616] read_proc: END
[ 5979.741932] write_proc: overflow, enlarge buffer...
[ 5979.741935] write_proc: MAX_BUF_SIZE 31 -> 61
[ 5979.741938] write_proc: write 41 bytes
[ 5979.743712] read_proc: read 41 bytes
[ 5979.743736] read_proc: END
[ 5979.759830] Exiting Module4...

```

图8. 模块四 结果

## H2 5. 总结与感想

在本Project中，我对Linux内核的模块编程有了一定的了解，并熟悉了proc文件系统的运作方式，这为接下来的学习打下了一个好的基础。