SHANGHAI JIAOTONG UNIVERSITY

Linux 内核课程设计 - Project1

Linux 内核模块编程

姓名：薛春宇

学号：518021910698

完成时间：2021/3/18

1 实验目的

编写四个模块，分别实现以下功能：

(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、字符串和数组参数在内核加载时读入并打印

(3) 模块三：在/proc 下创建只读文件，读取该文件并返回部分信息

(4) 模块四：在/proc 下创建文件夹，并创建一个可读可写的文件

2 实验准备

在正式开始编写内核模块之前，我们需要对一些内核模块编写及 proc 文件系统操作的基本知识有所了解。

2.1 Linux 内核模块相关指令

(1) 插入模块：insmod hello.ko

(2) 删除模块：rmmod hello

(3) 列出已加载模块：lsmod（列出特定名称的已加载模块：lsmod | grep hello）

(4) 查看模块信息：modinfo hello.ko

(5) 插入模块，并自动处理存在依赖关系的模块：modprobe hello.ko

2.2 Linux 内核数据结构

这里，我们介绍本次驱动程序中要用到的三个最重要的内核数据结构，分别为 inode、file 和 file_operations。

Inode 是储存文件元信息的索引节点，用于表示文件。每一个文件都有对应的 inode，里面包含了与该文件有关的一些信息（除文件名以外的所有文件信息），包括文件字节数、拥有者 ID 和读写权限等。Linux 允许多个文件名指向同一个 inode 号码，共享一块数据块。Inode 的链接有硬链接和软链接两种方式。

File 是一种指示已打开文件的文件结构体。系统中的每个打开的文件在内核空间都有一个相应的 struct file 结构体，它由内核在打开文件时创建，并传递给在文件上进行操作的任何函数，直至文件被关闭。如果文件被关闭，内核就会释放相应的数据结构。可能会存在有多个 file 结构同时指向单个 inode 结构。

File_operations 结构体用于说明设备驱动的接口。在系统内部，I/O 设备的存取操作通过特定的入口进行，而这组特定的入口由驱动程序来提供，通常这组设备驱动的接口是由 file_operations 结构体向系统说明的。常见的成员函数包括 open、read、write、llseek

和 release 等。

2.3 /proc 文件系统

伪文件系统，追踪、记录系统状态以及进程状态，是 Linux 内核信息的统一获取接口。

实时变化，存在于虚拟内存中（不存放于任何存储介质），文件夹大小是 0（不占据硬盘空间），修改时间是上次启动的时间。每次 Linux 系统重启时，都会创建新的/proc 文件系统。

3 实验内容

本部分内容由两个部分组成，分别是四个内核模块的实现，以及 Makefile 的编写。

3.1 模块一：模块的加载/卸载

本模块主要实现 Linux 内核加载和卸载模块时，系统日志信息的输出，是内核模块化编程最为基础的框架。

3.1.1 代码结构

首先，在内核模块编程时需要引入的三个基本头文件：

```
1  // <module1.c>
2  // Test for installing and removing of module
3  #include <linux/kernel.h>
4  #include <linux/module.h>
5  #include <linux/init.h>
```

其作用分别是：

(1) <linux/kernel.h>：包含了内核打印函数 printk()等基本函数原型

(2) <linux/module.h>：作用是动态地将模块加载到内核中（必须加载）

(3) <linux/init.h>：包含了模块的初始化的宏定义，以及一些函数的初始化函数

```
7  // Enterance
8  static int __init hello_init(void) {
9      printk(KERN_INFO "Test for Module1...\n");
10     return 0;
11 }
12
13 // Exitance
14 static void __exit hello_exit(void) {
15     printk(KERN_INFO "Bye.\n");
16 }
```

其次，入口函数 hello_init 和出口函数 hello_exit 是内核模块编程两个最重要的基础，这两个函数的函数名可以进行自定义，但函数原型和参数设置必须符合规定。

```
17
18 module_init(hello_init);
19 module_exit(hello_exit);
20
```

在设置好出口和入口函数之后，我们需要将这两个函数分别设置为内核模块的入口和出口，具体的做法是使用在<linux/init.h>头文件中定义的 module_init 和 module_exit 调用，

来对内核模块的出入口进行设置。

```
21  MODULE_LICENSE("GPL");
22  MODULE_DESCRIPTION("Module1");
23  MODULE_AUTHOR("Chunyu Xue");
```

最后，我们使用 MODULE_LICENSE、MODULE_DESCRIPTION 和 MODULE_AUTHOR 等宏进行模块许可证、描述和作者的声明。

3.1.2 运行效果

运行效果如图 1 所示。我们首先利用 dmesg -C 指令清空系统的日志信息，再利用 insmod module1.ko 指令将已经编译好的.ko 文件插入内核。使用 dmesg 打开系统日志后可以看到，日志输出了“Test for Module1…”的信息，再利用 lsmod | grep module1 指令查看已加载模块，发现 module1 模块已成功加载。最后，我们通过 rmmod module1 指令将 module1 模块从内核中移除。

```
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module1# dmesg -C
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module1# insmod module1.ko
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module1# dmesg
[  577.784131] Test for Module1...
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module1# lsmod | grep module1
module1                16384  0
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module1# rmmod module1
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module1# dmesg
[  577.784131] Test for Module1...
[  594.105179] Bye.
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module1#
```

图 1 Module1 的运行效果

3.2 模块二：模块的参数传递

本模块主要实现 Linux 内核加载时整型、字符串和数组参数的读入并打印，是模块化编程输入实现的重要组成部分。

3.2.1 代码结构

首先，在 3.1 节基本框架的基础上，我们额外引入了两个头文件，名称及作用分别是：

(1) <linux/moduleparam.h>：用于模块参数传递宏 module_param 的引入

(2) <linux/string.h>：用于字符串比对函数 strcmp 的引入

```
1  // <module2.c>
2  // Support for int & str & array parameter
3  #include <linux/kernel.h>
4  #include <linux/module.h>
5  #include <linux/init.h>
6  #include <linux/moduleparam.h>
7  #include <linux/string.h>
```

接下来，我们进行一些变量定义和参数传递接口的设置：

```c
// Definition for parameters
static int int_var = -9999;
static char *str_var = "Default";
static int int_array[10];
// Number of elements in array
int arrNum;

// Set interface
module_param(int_var, int, 0644);
MODULE_PARM_DESC(int_var, "An integer variable");          // Description of parameter
module_param(str_var, charp, 0644);
MODULE_PARM_DESC(str_var, "A string variable");
module_param_array(int_array, int, &arrNum, 0644);
MODULE_PARM_DESC(int_array, "An integer array");
```

可以看到，我们定义了三个变量，分别是整型变量 int_var，字符型指针 str_var 和整型数组 int_array，此外我们还设置了一个变量 arrNum 来指示数组的长度。然后，我们使用 module_param (name, type, perm) 函数来进行变量参数接口的设置，来将这三个变量与命令行参数关联起来。其中，name 是变量名，type 是变量类别，perm 是访问权限。我们使用 MODULE_PRAM_DESC 对参数添加相应的描述。

之后便是入口函数的实现，这是本模块最重要的部分：

```c
// Enterance
static int __init hello_init(void) {
    int i;

    if (int_var == -9999 && strcmp(str_var, "Default") == 0 && arrNum == 0) {
        printk(KERN_INFO "No parameters input, exit.\n");
        return 0;
    }

    printk(KERN_INFO "The parameters are:\n");
    if (int_var != -9999) {
        printk(KERN_INFO "Int: %d\n", int_var);
    }
    if (strcmp(str_var, "Default") != 0) {
        printk(KERN_INFO "Str: %s\n", str_var);
    }
    if (arrNum != 0) {
        for(i = 0; i < arrNum; i++) {
            printk(KERN_INFO "Int_array[%d]: %d\n", i, int_array[i]);
        }
    }
    return 0;
}
```

下面将对该入口函数的实现进行必要的解释。首先，为了提高函数的泛化性，我们设置了在无参数传递的情况下，会在系统日志中输出“No parameters input, exit.”的信息，并提前返回。若存在参数的输入，函数会分别利用变量的缺省值或数组长度作为判据，来判断应该输出哪类参数。对于数组参数，我们采用每行一个元素的方式进行输出。

最后，同 3.1 节中的基本框架相同，我们进行模块出入口的设置及版本信息的声明。

3.2.2 运行效果

我们首先演示在输入全部三种类型参数情况下的运行效果，通过在执行 insmod 内核插入操作的指令时添加 int_var、str_var 及 int_array 参数传递的方式，我们能够将相应的参数传入内和空间。需要注意的是，命令行参数需要同在模块代码中定义的变量名称保持一致。运行效果如图 2 所示。

```
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module2# insmod module2.ko int_var=12345 str_var=helloWorld int_a
rray=5,4,3,2,1,0
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module2# lsmod | grep module2
module2                16384  0
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module2# rmmod module2
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module2# dmesg
[16500.216510] The parameters are:
[16500.216511] Int: 12345
[16500.216511] Str: helloWorld
[16500.216512] Int_array[0]: 5
[16500.216512] Int_array[1]: 4
[16500.216512] Int_array[2]: 3
[16500.216513] Int_array[3]: 2
[16500.216513] Int_array[4]: 1
[16500.216513] Int_array[5]: 0
[16527.334761] Bye.
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module2#
```

图 2(1) Module2 的运行效果 (1)

通过 lsmod 及 dmesg 相关信息可以看到，我们成功完成了 Linux 内核加载时整型、字符串和数组参数的读入和打印目标。当我们在内核加载过程中传入的参数不包含甚至没有参数传递时，仍然支持输出的自主选择：

```
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module2# insmod module2.ko int_var=12345
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module2# dmesg
[16572.868562] The parameters are:
[16572.868563] Int: 12345
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module2#
```

图 2(2) Module2 的运行效果 (2)

```
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module2# insmod module2.ko
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module2# dmesg
[16825.638781] No parameters input, exit.
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module2#
```

图 2(3) Module2 的运行效果 (3)

完成实验后，通过 rmmod 指令将 module2 模块移出内核即可。

3.3 模块三：模块创建只读文件

本模块主要实现 Linux 在/proc 文件系统下创建只读文件，读取该文件并返回部分信息，是基于 proc 文件系统编程的基本操作。

3.3.1 代码结构

在 3.1 节基本框架的基础上，本模块额外引入了三个头文件，其作用分别是：

(1) <linux/proc_fs.h>：包含一些 proc 文件系统读、写和创建操作的基本函数

(2) <linux/seq_file.h>：包含 seq-read、seq_lseek 等顺序文件处理的函数

(3) <linux/sched.h>：任务调度相关，包含对内核系统时间进行读取的全局变量 jiffies

```
4   #include <linux/proc_fs.h>
5   #include <linux/seq_file.h>
6   #include <linux/sched.h>
```

接下来，我们首先介绍模块的入口函数，hello_proc_init：

```
40  static int __init hello_proc_init(void) {
41      printk(KERN_INFO "Test for module3...\n");
42      // Create proc file
43      proc_create("helloworld", 0444, NULL, &hello_proc_fops);
44
45      return 0;
46  }
```

在入口函数中，调用了在<linux/proc_fs.h>中定义的 proc_create (name, perm, parent, fops)函数，其作用是根据 fops 文件操作结构体创建一个名称为 name，访问权限为 perm，父目录为 parent 的 proc 文件。上述的 fops 文件操作结构体实现如下：

```
// Specify file operations
static const struct proc_ops hello_proc_fops = {
    .proc_open = hello_proc_open,
    .proc_read = seq_read,
    .proc_lseek = seq_lseek,
    .proc_release = single_release,
};
```

这里，我们定义了打开、读取、定位和释放四个文件操作函数，其中仅.proc_open 被定义为自定义的函数，另外三个全部为封装好的库函数。需要注意的是，我们在这里不使用 file_operations 结构体的原因是，在 Linux 内核 5.6 或更高版本中，我们需要将 file_operations 结构体更换为新的 proc_ops 结构体。

当 proc 文件被成功创建后，会调用 proc_ops 结构体中的.proc_open 成员函数，即我们自定义的 hello_proc_open 函数：

```
// Definition of file operations
static int hello_proc_open(struct inode *inode, struct file *file) {
    return single_open(file, hello_proc_show, NULL);
}
```

注意到，这里的 hello_proc_open 函数只是通过 single_open 库函数来对 hello_proc_show 函数进行了封装，后者则是通过 seq_printf 函数来进行了内核信息的读取和输出：

```
static char *str = "Successfully read content from proc file!";

// Obtain info
static int hello_proc_show(struct seq_file *m, void *v) {
    seq_printf(m, "%s\n", str);
    seq_printf(m, "Current kernel time is: %ld\n", jiffies);

    return 0;
}
```

我们通过图表的方式来展现代码中几个重要模块的关系：

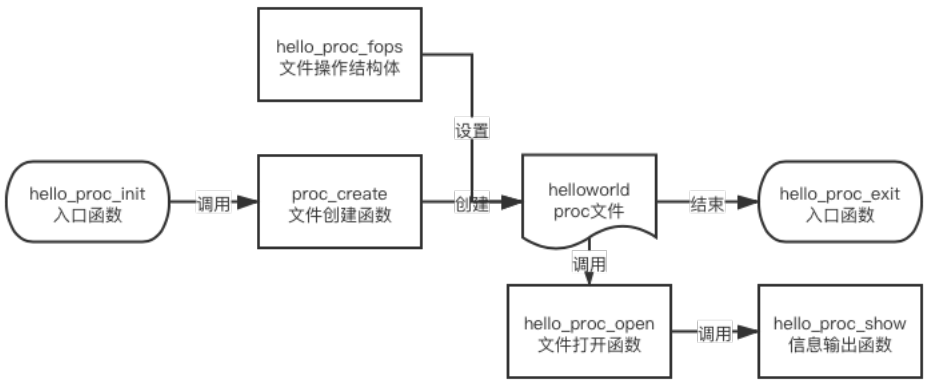

图 3 Module3 的模块关系

3.3.2 运行效果

运行效果如图 4 所示。在和之前一样成功插入内核模块 module3 之后，通过指令 cat /proc/helloworld 来读取新创建的 proc 文件的内容。可以看到，该操作成功输出了成功读取的信息字符串，以及当前内核的系统时间。同时，通过运行 ls -l /proc/helloworld 指令可以看到，该 proc 文件的读写权限为只读。最后通过 rmmod 将 module3 模块卸载。

```
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module3# dmesg -C
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module3# insmod module3.ko
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module3# lsmod | grep module3
module3                16384  0
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module3# dmesg
[  258.000093] Test for module3...
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module3# cat /proc/helloworld
Successfully read content from proc file!
Current kernel time is: 4294969851
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module3# ls -l /proc/helloworld
-r--r--r-- 1 root root 0 Mar 17 22:22 /proc/helloworld
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module3# rmmod module3
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module3# dmesg
[  258.000093] Test for module3...
[  359.318411] Bye.
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module3#
```

图 4 Module3 的运行效果

3.4 模块四：模块创建文件夹及读写文件

本模块主要实现 Linux 在/proc 文件系统下创建文件夹，并创建一个可读可写的文件，同样是基于 proc 文件系统编程的基本操作。

3.4.1 代码结构

在 3.3 节代码的基础上，本模块额外引入了两个头文件，其作用分别是：

(1) <linux/slab.h>：包含了 kzalloc、kfree 等内存分配函数

(2) <linux/uaccess.h>：包含了从用户空间到内核空间的拷贝函数 copy_from_user

该模块在除了 write 函数之外的其他部分的相对逻辑关系与 3.3 节的 module3 大致相同，不同之处在于在入口和出口函数中添加了 proc 文件系统下的目录创建和删除：

```
63  // module init
64  static int __init hello_init(void) {
65      printk(KERN_INFO "Test for module4...\n");
66      // Create proc directory
67      helloworldDir =  proc_mkdir("helloworldDir", NULL);
68      if (!helloworldDir) {
69          return -ENOMEM;
70      }
71      // Create proc file
72      helloworld = proc_create("helloworld", 0644, helloworldDir, &hello_proc_fops);
73      if (!helloworld) {
74          return -ENOMEM;
75      }
76
77      return 0;
78  }
```

```
80  // module exit
81  static void __exit hello_exit(void)
82  {
83      // Remove file
84      remove_proc_entry("helloworld", helloworldDir);
85      // Remove dir
86      remove_proc_entry("helloworldDir", NULL);
87
88      printk(KERN_INFO "Bye.\n");
89  }
```

需要注意的是，在创建及删除的过程中要注意目录与文件的相对顺序，在创建时必须先创建目录，再在目录下创建文件；同样的，在删除时必须要先删除文件，再删除目录。且在 proc_create()和 remove_proc_entry()的过程中，helloworld 文件的父目录 parent 需要设置成创建的目录 helloworldDir。

在文件操作结构体的设置中，我们相对于 3.3 节添加了.proc_write=hello_proc_write 的成员函数设置，而 hello_proc_open 的设置与 3.3 节相同。hello_proc_show 的输出信息更改为：

```
14  // Message
15  static char *message = NULL;
16  // Directory and file
17  struct proc_dir_entry *helloworldDir, *helloworld;
18
19
20  // Show
21  static int hello_proc_show(struct seq_file *m, void *v) {
22      seq_printf(m, "Successfully read content from proc file!\n");
23      seq_printf(m, "The message is: %s\n", message);
24      return 0;
25  }
```

其中，message 为我们写入 helloworld 文件的字符串，缺省值为 null。

接下来，我们重点看一下 hello_proc_write 函数的代码实现：

```
28  // file_operations -> write
29  static ssize_t hello_proc_write(struct file *file, const char __user *buffer, size_t count, loff_t *f_pos)
30  {
31      // Create user buffer
32      char *userBuffer = kzalloc((count + 1), GFP_KERNEL);
33      if (!userBuffer) {
34          return -ENOMEM;
35      }
36      // Copy the data in user space to kernel space
37      if(copy_from_user(userBuffer, buffer, count)) {
38          kfree(userBuffer);
39          return EFAULT;
40      }
41      // Release the original string space
42      kfree(message);
43      // Redirect
44      message = userBuffer;
45      return count;
46  }
```

可以看到，在 hello_proc_write 函数中，我们首先利用 kzalloc 函数动态分配了一块空间，再利用 copy_from_user 系统调用将处于 buffer 中的命令行参数读入内核空间中新开辟的空间里，利用 kfree 将 message 指针原本指向的内容所占的空间释放，最后将其指向新开辟的含有命令行参数的空间。

3.4.2 运行效果

运行效果如图 5 所示。在和之前一样成功插入内核模块 module4 之后，通过指令 echo 1234567 > /proc/helloworldDir/helloworld 来向 proc 文件中写入内容，再通过 cat /proc/helloworldDir/helloworld 来读出 proc 文件中刚刚写入的字符串。同时，通过运行 ls -l /proc/helloworldDir/helloworld 指令可以看到，该 proc 文件的读写权限为可读可写。最后通过 rmmod 将 module4 模块卸载。

```
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module4# dmesg -C
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module4# insmod module4.ko
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module4# lsmod | grep module4
module4                16384  0
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module4# dmesg
[ 3050.533331] Test for module4...
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module4# cat /proc/helloworldDir/helloworld
Successfully read content from proc file!
The message is: (null)
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module4# echo 1234567 > /proc/helloworldDir/helloworld
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module4# cat /proc/helloworldDir/helloworld
Successfully read content from proc file!
The message is: 1234567

root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module4# ls -l /proc/helloworldDir/helloworld
-rw-r--r-- 1 root root 0 Mar 18 20:48 /proc/helloworldDir/helloworld
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module4# rmmod module4
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module4# dmesg
[ 3050.533331] Test for module4...
[ 3208.721788] Bye.
root@ubuntu:/home/dicardo/Desktop/LinuxKernel/Project1/Module4#
```

图 5 Module4 的运行效果

3.4 Makefile 文件的编写

为了编译上述四个模块的源码以生成可加载的.ko 文件，我们需要编写特定的 Makefile 文件，并在命令行中运行 make 指令进行编译。Makefile 文件如下所示：

```
obj-m := module1.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

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

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

图 6 Makefile 文件

注意到，在编译不同的模块时，只需要将该 Makefile 文件同源码放在一个目录下，再修改第一行中的中间文件名，即可完成编译。需要特别注意的是，在进行第四个模块的编译时，在系统日志中输出了如下错误：

```
[89.203313] module4: loading out-of-tree module taints kernel.
[89.203350] module4: module verification failed: signature and/or required key missing - tainting kernel
```

在查阅资料后发现，是内核模块的签名在动态加载时出现了验证的问题，因此在 Makefile 的开头处加入“CONFIG_MODULE_SIG=n”来关闭内核的签名功能，成功解决该问题。

4 实验结果

在解决遇到的一系列问题，经过一天的不断尝试之后，终于顺利在 ubuntu20.04 上实现并验证了几个具有特定功能的内核模块。

5 实验心得

本次实验是 Linux 内核课程的第一次正式 project，旨在掌握面向内核模块化编程及 proc 文件系统编程的基本知识。由于之前在操作系统的课程中学习过一点关于内核编程的知识，这次实验的整体难度尚在我的能力范围之内。前两个模块的实现较为简单，在第三个模块中实现 proc 文件系统下的文件创建时，由于内核版本的问题，demo 中给出的 file_operations 文件操作结构体已经不再适用，且目前网上相关问题的解决方案少之又少。在经过一系列的查阅资料之后，我终于找到了问题的解决办法[1]：将 file_operations 结构体更换为 proc_ops 结构体，其成员函数也发生相应的改变：

```
// Specify file operations, we need to replace struct file_operations with struct proc_ops for kernel version 5.6 or later
static const struct proc_ops hello_proc_fops = {
    .proc_open = hello_proc_open,
    .proc_read = seq_read,
    .proc_lseek = seq_lseek,
    .proc_release = single_release,
};

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

此外，在第四个模块的 make 过程中，我遇到了内核模块签名的问题，在查阅资料后我发现，只需要在 Makefile 最开始加上一行“CONFIG_MODULE_SIG=n”来关闭内核的签名功能，就能够成功解决该问题。

总的来说，我在本次实验中成功完成了四个内核模块的编写及运行，在增强了内核编程知识的同时，也提高了自身发现问题、解决问题的能力。希望能在下面的实验中收获满满！

[1] https://stackoverflow.com/questions/64931555/how-to-fix-error-passing-argument-4-of-proc-create-from-incompatible-pointer