# 苏州大学实验报告

| 院、系 | 计算机学院 | 年级专业 | 19 计科图灵 | 姓名 | 张昊 | 学号 | 1927405160 |
|---|---|---|---|---|---|---|---|
| 课程名称 | 操作系统课程实践 | | | | | 成绩 | |
| 指导教师 | 李培峰 | 同组实验者 | 无 | | 实验日期 | 2022 年 6 月 2 日 | |

实 验 名 称　　实验 9　Linux 内核模块

## 一. 实验目的

1. 理解针对 Linux 提出内核模块这种机制的意义。
2. 理解并掌握 Linux 实现内核模块机制的基本技术路线。
3. 运用 Linux 提供的工具和命令，掌握操作内核模块的方法。
4. 掌握将多个源文件合并到一个内核模块中的方法。
5. 掌握复杂内核模块的实现方法。

## 二. 实验内容

1. （**实验 10.1: 编写一个简单的内核模块**）
   编写一个简单的具备基本要素的内核模块，并编写这个内核模块所需要的 Makefile，最后编译内核并将其载入系统。
2. （**实验 10.2: 利用内核模块创建一个设备文件节点**）
   利用内核模块创建一个设备文件节点。

## 三. 操作方法和实验步骤

### 1. 编写一个简单的内核模块

（1）编写内核模块源代码文件 helloworld.c：

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

static int __init init_hello(void) {
  printk("<1>Hello World, by Hao Zhang!");
  return 0;
}

static void __exit cleanup_hello(void) {
  printk("<1>Goodbye, by Hao Zhang!");
}

MODULE_LICENSE("GPL");
module_init(init_hello);
module_exit(cleanup_hello);
MODULE_AUTHOR("Hao Zhang");
MODULE_DESCRIPTION("hello world");
```

（2）编写编译内核模块时要用到的 Makefile 文件：

```
obj-m := helloworld.o
```

```
KDIR := /usr/src/linux-$(shell uname -r | cut -d '-' -f1)
PWD := $(shell pwd)
all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

（3）编译 helloworld.c：

```
# make
```

```
holger@hao-zhang:/codes/exp09/helloworld$ make
make -C /usr/src/linux-4.16.10 M=/codes/exp09/helloworld modules
make[1]: Entering directory '/usr/src/linux-4.16.10'
  CC [M]  /codes/exp09/helloworld/helloworld.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /codes/exp09/helloworld/helloworld.mod.o
  LD [M]  /codes/exp09/helloworld/helloworld.ko
make[1]: Leaving directory '/usr/src/linux-4.16.10'
holger@hao-zhang:/codes/exp09/helloworld$
```

（4）执行内核模块装入命令：

```
# insmod helloworld.ko
```

（5）使用 dmesg 命令查看内核输出信息，使用 lsmod 命令查看模块信息。

（6）当不需要使用 helloworld 模块时卸载这个模块：

```
# rmmod helloworld
```

**2. 利用内核模块创建一个设备文件节点**

利用内核模块创建一个设备文件节点。

（1）编写内核模块源代码文件 procfs_example.c：

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/jiffies.h>
#include <linux/sched.h>
#include <linux/uaccess.h>
#include <linux/seq_file.h>
#include <linux/fs.h>

#define MODULE_VERS "1.0"
#define MODULE_NAME "procfs_example"
#define FOOBAR_LEN 8
struct fb_data_t {
    char name[FOOBAR_LEN + 1];
    char value[FOOBAR_LEN + 1];
};
static struct proc_dir_entry *example_dir, *foo_file,
    *bar_file, *jiffies_file, *symlink;
static int major = 255;
```

```
static int minor = 0;
struct fb_data_t foo_data, bar_data;
int foo_len, foo_temp, bar_len, bar_temp;
int jiff_temp = -1;
char tempstr[FOOBAR_LEN * 2 + 5];

//jiffies 文件操作函数
static ssize_t read_jiffies_proc(struct file *filp, char __user *buf,
                                 size_t count, loff_t *offp) {
   printk(KERN_INFO"count=%d  jiff_temp=%d\n", count, jiff_temp);
   char tempstring[100] = "";
   if (jiff_temp != 0)
      jiff_temp = sprintf(tempstring, "jiffies=%ld\n", jiffies);
      //jiffies 为系统启动后经过的时间戳
   if (count > jiff_temp)
      count = jiff_temp;
   jiff_temp = jiff_temp - count;
   printk(KERN_INFO"count=%d  jiff_temp=%d\n", count, jiff_temp);
   copy_to_user(buf, tempstring, count);
   if (count == 0)
      jiff_temp = -1;    //读取结束后 temp 变回-1
   return count;
}
static const struct file_operations jiffies_proc_fops = {
   .read = read_jiffies_proc
};
//foo 文件操作函数
static ssize_t read_foo_proc(struct file *filp, char __user *buf,
                             size_t count, loff_t *offp ) {
   printk(KERN_INFO"count=%d\n", count);
   //调整 count 与 temp 的值，具体过程自行查看 printk 输出的信息来分析
   if (count > foo_temp)
      count = foo_temp;
   foo_temp = foo_temp - count;
   //拼接 tempstr 字符串
   strcpy(tempstr, foo_data.name);
   strcat(tempstr, "='");
   strcat(tempstr, foo_data.value);
   strcat(tempstr, "'\n");
   printk(KERN_INFO"count=%d  length(tempstr)=%d\n",
          count, strlen(tempstr));
   //向用户空间写入 tempstr
   copy_to_user(buf, tempstr, count);
   //如果 count=0，读取结束，temp 回归为原来的值
   if (count == 0)
```

```
        foo_temp = foo_len + 4;
    return count;
}
static ssize_t write_foo_proc(struct file *filp, const char __user *buf,
                              size_t count, loff_t *offp) {
    int len;
    if (count > FOOBAR_LEN)
        len = FOOBAR_LEN;
    else
        len = count;
    //将数据写入 foo_data 的 value 字段
    if (copy_from_user(foo_data.value, buf, len))
        return -EFAULT;
    foo_data.value[len-1]='\0';    //减 1 是因为除去输入的回车
    //更新 len 与 temp 值
    foo_len = strlen(foo_data.name)+strlen(foo_data.value);
    foo_temp = foo_len + 4;
    return len;
}
static const struct file_operations foo_proc_fops={
    .read = read_foo_proc,
    .write = write_foo_proc
};
//bar 文件操作函数
static ssize_t read_bar_proc(struct file *filp,char __user *buf,
                             size_t count,loff_t *offp ) {
    printk(KERN_INFO"count=%d\n", count);
    if (count > bar_temp)
        count = bar_temp;
    bar_temp = bar_temp - count;
    strcpy(tempstr, bar_data.name);
    strcat(tempstr, "='");
    strcat(tempstr, bar_data.value);
    strcat(tempstr, "'\n");
    printk(KERN_INFO"count=%d  length(tempstr)=%d\n",
           count, strlen(tempstr));
    copy_to_user(buf, tempstr, count);
    if (count == 0)
        bar_temp = bar_len + 4;
    return count;
}
static ssize_t write_bar_proc(struct file *filp,const char __user *buf,
                              size_t count, loff_t *offp) {
    int len;
    if (count > FOOBAR_LEN)
```

```
        len = FOOBAR_LEN;
    else
        len = count;
    if (copy_from_user(bar_data.value, buf, len))
        return -EFAULT;
    bar_data.value[len-1] = '\0';
    bar_len = strlen(bar_data.name) + strlen(bar_data.value);
    bar_temp = bar_len + 4;
    return len;
}
static const struct file_operations bar_proc_fops = {
    .read = read_bar_proc,
    .write = write_bar_proc
};
//模块 init 函数
static int __init init_procfs_example(void) {
    int rv = 0;
    //创建目录
    example_dir = proc_mkdir(MODULE_NAME, NULL);
    if (example_dir == NULL) {
        rv = -ENOMEM;
        goto out;
    }
    //创建 jiffies(只读)
    jiffies_file = proc_create("jiffies", 0444,
                               example_dir, &jiffies_proc_fops);
    if (jiffies_file == NULL) {
        rv = -ENOMEM;
        goto no_jiffies;
    }
    //创建 foo
    strcpy(foo_data.name, "foo");
    strcpy(foo_data.value, "foo");
    foo_len = strlen(foo_data.name) + strlen(foo_data.value);
    foo_temp = foo_len + 4; //加 4 是因为拼接 tempstr 字符串时多了=''\n"四个字符
    foo_file = proc_create("foo", 0, example_dir, &foo_proc_fops);
    if (foo_file == NULL) {
        rv = -ENOMEM;
        goto no_foo;
    }
    //创建 bar
    strcpy(bar_data.name, "bar");
    strcpy(bar_data.value, "bar");
    bar_len = strlen(bar_data.name) + strlen(bar_data.value);
    bar_temp = bar_len + 4;
```

```
    bar_file = proc_create("bar", 0, example_dir, &bar_proc_fops);
    if (bar_file == NULL) {
        rv = -ENOMEM;
        goto no_bar;
    }
    //创建 symlink
    symlink = proc_symlink("jiffies_too", example_dir, "jiffies");
    if (symlink == NULL){
        rv = -ENOMEM;
        goto no_symlink;
    }
    //all ok
    printk(KERN_INFO"%s%s initialised\n", MODULE_NAME, MODULE_VERS);
    return 0;

no_symlink:
    remove_proc_entry("jiffies_too", example_dir);
no_bar:
    remove_proc_entry("bar", example_dir);
no_foo:
    remove_proc_entry("foo", example_dir);
no_jiffies:
    remove_proc_entry("jiffies", example_dir);
out:
    return rv;
}
//模块 cleanup 函数
static void __exit cleanup_procfs_example(void) {
    remove_proc_entry("jiffies_too", example_dir);
    remove_proc_entry("bar", example_dir);
    remove_proc_entry("foo", example_dir);
    remove_proc_entry("jiffies", example_dir);
    remove_proc_entry(MODULE_NAME, NULL);
    printk(KERN_INFO"%s%s removed\n", MODULE_NAME, MODULE_VERS);
}
MODULE_LICENSE("GPL");
module_init(init_procfs_example);
module_exit(cleanup_procfs_example);
MODULE_DESCRIPTION("proc filesystem example");
```

（2）编写 Makefile 文件：

```
CONFIG_MODULE_SIG=n
obj-m := procfs_example.o
KDIR := /usr/src/linux-$(shell uname -r | cut -d '-' -f1)
PWD := $(shell pwd)
all:
```

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

（3）编译：

```
# make
```

（4）执行内核模块装入命令：

```
# insmod procfs_example.ko
```

（5）当不需要使用时卸载这个模块：

```
# rmmod procfs_example
```

## 四. 实验结果和分析

### 1. 编写一个简单的内核模块

（1）执行内核模块装入命令：

```
# insmod helloworld.ko
```

（2）使用 lsmod 命令查看模块信息：

```
# lsmod | grep helloworld
```

```
holger@hao-zhang:/codes/exp09/helloworld$ lsmod | grep helloworld
helloworld             16384  0
holger@hao-zhang:/codes/exp09/helloworld$
```

（3）使用 dmesg 命令查看内核输出信息：

```
# dmesg
```

```
holger@hao-zhang:/codes/exp09/helloworld$ sudo dmesg | tail
[   34.882598] perf: interrupt took too long (2704 > 2500), lowering kernel.perf_event_max_sample_rate to 73750
[   36.392286] Bluetooth: RFCOMM TTY layer initialized
[   36.392295] Bluetooth: RFCOMM socket layer initialized
[   36.392306] Bluetooth: RFCOMM ver 1.11
[  102.292320] perf: interrupt took too long (3850 > 3380), lowering kernel.perf_event_max_sample_rate to 51750
[  102.912784] helloworld: loading out-of-tree module taints kernel.
[  102.912853] helloworld: module verification failed: signature and/or required key missing - tainting kernel
[  102.919114] <1>Hello World, by Hao Zhang!
[  109.153408] e1000: ens33 NIC Link is Down
[  115.201924] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
holger@hao-zhang:/codes/exp09/helloworld$
```

（4）卸载模块:

```
# rmmod helloworld
```

（5）使用 dmesg 命令查看内核输出信息：

```
# dmesg
```

```
holger@hao-zhang:/codes/exp09/helloworld$ sudo dmesg | tail
[   36.392286] Bluetooth: RFCOMM TTY layer initialized
[   36.392295] Bluetooth: RFCOMM socket layer initialized
[   36.392306] Bluetooth: RFCOMM ver 1.11
[  102.292320] perf: interrupt took too long (3850 > 3380), lowering kernel.perf_event_max_sample_rate to 51750
[  102.912784] helloworld: loading out-of-tree module taints kernel.
[  102.912853] helloworld: module verification failed: signature and/or required key missing - tainting kernel
[  102.919114] <1>Hello World, by Hao Zhang!
[  109.153408] e1000: ens33 NIC Link is Down
[  115.201924] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[  206.713045] <1>Goodbye, by Hao Zhang!
holger@hao-zhang:/codes/exp09/helloworld$
```

（6）使用 lsmod 命令查看模块信息：

```
# lsmod | grep helloworld
```

```
holger@hao-zhang:/codes/exp09/helloworld$ lsmod | grep helloworld
holger@hao-zhang:/codes/exp09/helloworld$
```

**2. 利用内核模块创建一个设备文件节点**

（1）执行内核模块装入命令：

```
# insmod procfs_example.ko
```

```
root@hao-zhang:/codes/exp09/procfs# insmod procfs_example.ko
root@hao-zhang:/codes/exp09/procfs# lsmod | grep procfs
procfs_example         16384  0
root@hao-zhang:/codes/exp09/procfs#
```

（2）查看/proc/porcfs_example 目录中各文件及属性：

```
# cd /proc/procfs_example
# cat bar foo jiffies jiffies_too
# ls -l
```

```
root@hao-zhang:/codes/exp09/procfs# cd /proc/procfs_example
root@hao-zhang:/proc/procfs_example# cat bar foo jiffies jiffies_too
bar='bar'
foo='foo'
jiffies=4295470542
jiffies=4295470542
root@hao-zhang:/proc/procfs_example# ls -l
total 0
-r--r--r-- 1 root root 0 Jun  2 07:13 bar
-r--r--r-- 1 root root 0 Jun  2 07:13 foo
-r--r--r-- 1 root root 0 Jun  2 07:13 jiffies
lrwxrwxrwx 1 root root 7 Jun  2 07:13 jiffies_too -> jiffies
root@hao-zhang:/proc/procfs_example#
```

bar、foo 和 jiffies 为只读文件，jiffies_too 为 jiffies 的链接文件（属性为可读可写）。

（3）尝试修改 jiffies 文件：

Vim 提示只读：

```
~
"jiffies" [readonly] 1L, 19C
```

强制写入修改：

```
~
"jiffies"
WARNING: The file has been changed since reading it!!!
Do you really want to write to it (y/n)?y
"jiffies" E667: Fsync failed
WARNING: Original file may be lost or damaged
don't quit the editor until the file is successfully written!
Press ENTER or type command to continue
```

发现修改不成功，jiffies 文件为只读文件。

尝试修改 jiffies_too 文件，依旧提示为只读：

```
"jiffies_too" [readonly] 1L, 19C
```

强制修改不成功：

```
~
"jiffies_too"
WARNING: The file has been changed since reading it!!!
Do you really want to write to it (y/n)?y
"jiffies_too" E667: Fsync failed
WARNING: Original file may be lost or damaged
don't quit the editor until the file is successfully written!
Press ENTER or type command to continue
```

尝试使用重定向写入，仍不成功：

```
root@hao-zhang:/proc/procfs_example# echo 'new line' >> jiffies
-bash: echo: write error: Input/output error
root@hao-zhang:/proc/procfs_example# echo 'new line' >> jiffies_too
-bash: echo: write error: Input/output error
```

jiffies_too 虽然为可读可写的链接文件，但其目标文件不可写，实际表现上 jiffies_too 文件仅可读。

（4）查看/dev 目录下设备文件 tty 的类型、内容、属性：

```
# ls -lh /dev/tty
# cat /dev/tty
# stat /dev/tty
# file -b /dev/tty
```

```
root@hao-zhang:/proc/procfs_example# ls -lh /dev/tty
crw-rw-rw- 1 root tty 5, 0 Jun  2 07:11 /dev/tty
root@hao-zhang:/proc/procfs_example# cat /dev/tty
^C
root@hao-zhang:/proc/procfs_example# stat /dev/tty
  File: '/dev/tty'
  Size: 0               Blocks: 0          IO Block: 4096   character special file
Device: 6h/6d   Inode: 14          Links: 1     Device type: 5,0
Access: (0666/crw-rw-rw-)  Uid: (    0/    root)   Gid: (    5/     tty)
Access: 2022-06-02 07:11:44.878886283 -0700
Modify: 2022-06-02 07:11:44.878886283 -0700
Change: 2022-06-02 06:34:56.878886283 -0700
 Birth: -
root@hao-zhang:/proc/procfs_example# file -b /dev/tty
character special (5/0)
root@hao-zhang:/proc/procfs_example#
```

可见，/dev/tty 文件为字符设备文件（character special），可读可写。

（5）卸载模块:

```
# rmmod procfs_example
```

（6）使用 dmesg 命令查看内核输出信息：

```
# dmesg
```

```
[ 2313.060591] count=131072
[ 2313.060593] count=10  length(tempstr)=10
[ 2313.060638] count=131072
[ 2313.060639] count=0  length(tempstr)=10
[ 2313.060670] count=131072
[ 2313.060671] count=10  length(tempstr)=10
[ 2313.060675] count=131072
[ 2313.060676] count=0  length(tempstr)=10
[ 2313.060682] count=131072  jiff_temp=-1
[ 2313.060683] count=19  jiff_temp=0
[ 2313.060685] count=131072  jiff_temp=0
[ 2313.060686] count=0  jiff_temp=0
[ 2313.062582] count=131072  jiff_temp=-1
[ 2313.062584] count=19  jiff_temp=0
[ 2313.062624] count=131072  jiff_temp=0
[ 2313.062625] count=0  jiff_temp=0
[ 2736.897884] count=8192  jiff_temp=-1
[ 2736.897886] count=19  jiff_temp=0
[ 2736.897905] count=65536  jiff_temp=0
[ 2736.897905] count=0  jiff_temp=0
[ 2749.808730] count=8192  jiff_temp=-1
[ 2749.808732] count=19  jiff_temp=0
[ 2749.808745] count=65536  jiff_temp=0
[ 2749.808746] count=0  jiff_temp=0
[ 2785.377580] count=8192  jiff_temp=-1
[ 2785.377582] count=19  jiff_temp=0
[ 2785.377605] count=8192  jiff_temp=0
[ 2785.377605] count=0  jiff_temp=0
[ 2789.340148] perf: interrupt took too long (6024 > 5747), lowering kernel.perf_event_max_sample_rate to 33000
[ 2792.027107] count=8192  jiff_temp=-1
[ 2792.027109] count=19  jiff_temp=0
[ 2792.027134] count=8192  jiff_temp=0
[ 2792.027134] count=0  jiff_temp=0
[ 3001.268079] count=8192  jiff_temp=-1
[ 3001.268082] count=19  jiff_temp=0
[ 3001.268114] count=65536  jiff_temp=0
[ 3001.268115] count=0  jiff_temp=0
[ 3016.249893] count=8192  jiff_temp=-1
[ 3016.249896] count=19  jiff_temp=0
[ 3016.249920] count=8192  jiff_temp=0
[ 3016.249921] count=0  jiff_temp=0
[ 3940.593884] count=8192  jiff_temp=-1
[ 3940.593885] count=19  jiff_temp=0
[ 3940.593899] count=65536  jiff_temp=0
[ 3940.593899] count=0  jiff_temp=0
[ 3960.585348] count=8192  jiff_temp=-1
[ 3960.585350] count=19  jiff_temp=0
[ 3960.585363] count=65536  jiff_temp=0
[ 3960.585364] count=0  jiff_temp=0
[ 4603.872699] procfs_example1.0 removed
```

## 五. 讨论、心得

1. 通过本次实验我理解了 Linux 内核模块这一机制。运用 Linux 提供的工具和命令，我掌握了编写、操作内核模块的方法。
2. Linux 内核模块这一机制的意义在于允许开发者动态的向内核添加功能，通过模块的方式添加到内核，无需重新编译内核，从而减少复杂度。
3. 实验过程中遇到了加载和卸载内核模块时 dmesg 信息输出不及时的现象，查阅资料得知内核日志是有缓冲区的，经检查发现使用 printk 函数输出信息时忽略了行尾的换行符，从而导致缓冲区没有及时刷新。
4. 如果一个模块由多个文件组成，可以采用模块名加-objs 后缀或者-y 后缀的形式来定义模块的组成文件，如：

```
obj-m = hello.o
hello-objs = helloworld.o  hello_suda.o
```

其中 hello 是希望生成的内核模块名称，整个内核模块包含 helloworld.c、hello_suda.c 两个 C 文件。

5. 在内核源代码中的输出函数选用了 printk()而不是常用的 printf()，这是因为在编译内核时还没有 C 的库函数可以供调用。因此，内核提供了专门用于在内核空间中信息打印的 printk()函数，来代替 C 标准库中的 printf()函数。printf()在终端打印，而 printk()函数为内核空间中打印。