# 《操作系统》课第十二次实验报告

| | |
|---|---|
| 学院: | 软件学院 |
| 姓名: | 杨万里 |
| 学号: | 2013774 |
| 邮箱: | 2013774@mail.nankai.edu.cn |
| 时间: | 2022/12/7 |

## 0. 开篇感言

我们不应该害怕未知，更应该努力把未知变为已知。

刚开始进行本次实验时，我对于字符设备的了解几乎为 0，因此对于本次实验的要求与用意感到一头雾水，经过不断地摸索与查询资料，逐渐建立起认知，最终完成实验。实验开始时觉得的不可能，最终也变成了可能。

## 1. 实验题目

Linux 内核模块驱动之读写特定文件

## 2. 实验目标

（1） 写一个 C/C++程序；
（2） 实现一个可以读写特定文件的内核模块；
（3） 具体要求：

- 可以选择一个具体文件（设备文件），通过 cat xxxfile 的形式实现对该文件的读操作；
- 可以通过 echo “hello” > xxxfile 和 echo “hello” >> xxxfile 的方式实现对该文件的写操作。

# 3. 原理方法

（1） 内核模块的概念/原理：

在上一次实验中初步探索了内核模块的编写。

Linux 内核模块是一些在**已经启动**的操作系统内核**需要时**可以**直接载入内核执行**的代码块，不需要时由操作系统卸载。它实现了在扩展内核功能的同时**避免重新编译、安装内核以及重启系统**的麻烦。同时也避免了操作系统过度的臃肿。

（2） 字符设备的概念/原理：

本次实验中内核模块读写文件要求的目标文件是 Linux 当中的字符设备文件，而非普通的文件。

Linux 是文件型系统，所有硬件、软件都会在对应的目录下面有相应的文件表示。对于 dev 这个目录，我们知道它下面的文件表示的是 Linux 的设备。在文件系统的 Linux 下面，都有对应文件与磁盘、硬盘等设备关联。访问它们就可以访问到实际硬件。直接读文件，写文件就可以向设备发送读或者写操作了。按照读写存储数据方式，我们可以把设备分为以下几种：字符设备，块设备，伪设备。

字符设备是指只能逐个字符读写的设备，不能随机读取设备内存中的某一数据。字符设备按照字符流的方式被有序访问。字符设备是面向流的设备，常见的字符设备有鼠标、键盘、串口、控制台和 LED 设备等。

本次实验的目标就是通过编写一个内核模块，通过重写 device_write、device_read 等函数，实现对字符设备文件的读写。

（3） 内核模块程序编写原理：

1）引入相关头文件：

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/kernel.h>
4 #include <linux/fs.h>
5 #include <linux/uaccess.h>
```

2）注册相关信息（内核模块编写的特点）：

```
7 MODULE_LICENSE("GPL");
8 MODULE_AUTHOR("YangWanli");
9 MODULE_DESCRIPTION("A simple example of linux module.");
10 MODULE_VERSION("0.01");
```

3）相关宏定义：

```
#define DEVICE_NAME "lkm_ywl_device"  //设备名称
#define EXAMPLE_MSG "Hello World YangWanli 2013774!\n"  //设备初始信息
#define MSG_BUFFER_LEN 128  //设备文件自定义存储大小
```

4）打开、释放、读、写设备的函数声明：

```
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);
```

5）相关的全局变量声明：

```
static int major_num;  //设备号
static int device_open_count = 0;  //设备打开次数
static char msg_buffer[MSG_BUFFER_LEN];  //设备文件的缓冲区（存储区）
static char *msg_ptr;  //读文件的指针
static int public_offset;  //标识写文件的offset
```

6）结构体 file_operations 变量内 read、write 等相关函数的覆盖：

```
static struct file_operations file_ops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

7）实现打开、释放、读、写文件的函数：

- 打开设备的函数 device_open:

```
//打开设备
static int device_open(struct inode *inode, struct file *file) {
    printk(KERN_INFO "[2013774ywl] Open /dev/%s c %d 0\n", DEVICE_NAME,
major_num);
    //已经打开则返回 -EBUSY
    if (device_open_count) {
        return -EBUSY;
    }
    device_open_count++;
    try_module_get(THIS_MODULE);
    return 0;
}
```

- 释放设备的函数 device_release:

```
//关闭设备
static int device_release(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "[2013774ywl] Close /dev/%s c %d 0\n",
DEVICE_NAME,major_num);
    device_open_count--;
    module_put(THIS_MODULE);
    return 0;
}
```

- 读设备文件的函数 device_read:

```
38 //读设备文件，每次命令行执行cat /dev/lkm_ywl_device时调用该函数
39 static ssize_t device_read(struct file *flip, char *buffer, size_t len, loff_t
   *offset)
40 {
41     //参数说明：调用该函数时，传入的len非常大(131072)，也就是默认读取所有内容
42     int bytes_read = 0;  //记录读取的字符/字节数目
43     printk(KERN_INFO "[2013774ywl] Read from /dev/%s c %d 0\n", DEVICE_NAME,
   major_num);
44     //该函数的作用就是把内核中msg_buffer中的内容读取到用户空间的buffer中
45     //注意，每次读完文件时，读文件的指针都指向文件的末尾，实际指向的内容是\0
46     //因此需要重新把读文件指针msg_ptr指向缓存区
47     if (*msg_ptr == 0) {
48         msg_ptr = msg_buffer;
49     }
50     while (len && *msg_ptr) {
51         //逐个字符读取
52         put_user(*(msg_ptr++), buffer++);
53         len--;
54         bytes_read++;
55     }
56     printk("[2013774ywl] %d bytes read\n", bytes_read);
57     return bytes_read;
58 }
```

不过该函数实现的弊端就是因为有了读文件指针 msg_ptr 的重定位，函数会不断调用，因此读设备文件时需要 Ctrl+c 终止。

- 写入设备文件的函数 device_write:

```
60 //写设备文件
61 static ssize_t device_write(struct file *flip, const char *buffer, size_t len,
   loff_t *offset)
62 {
```

每次命令行执行 echo “xxx” > lkm_ywl_device 或者 echo “xxx” >> lkm_ywl_device 时，都会调用 device_write 函数，其主要功能就是把用户空间的 buffer 的内容写入内核空间的 msg_buffer 中。**其中 echo >代表的是覆盖，也就是清空文件原有内容，并写入新的内容；而 echo >>则代表向文件末尾添加内容。**

首先遇到的一个问题就是，**当两种指令都调用同一函数，并需要执行不同功能时，如何区分这两种指令？**

可以注意到传入的文件指针参数 flip，file 结构体当中有一个参数 **f_flags**，将其与常量 **O_APPEND（1024）**相与可以得到一个数值，若该**数值等于 O_APPEND**，则代表 **echo >>**的打开方式，若该**数值为 0** 则代表 **echo >**的打开方式。

```
{
    int value;  //写文件返回值
    int flag;  //标识是>>还是>打开文件
    flag = O_APPEND & flip->f_flags;
    if(O_APPEND == flag){ //说明是>>打开方式，需要添加文件内容
        printk("[2013774ywl] append\n");
        //public_offset是全局变量，标志当前文件的末尾位移
        value = copy_from_user(msg_buffer + public_offset, buffer, len);
        public_offset += len;
    }
    else if(flag == 0){  //说明是>打开方式，需要覆盖文件内容
        printk("[2013774ywl] overlay\n");
        memset(msg_buffer,'\0',MSG_BUFFER_LEN);  //清空文件内容
        value = copy_from_user(msg_buffer, buffer, len);
        public_offset = len;
    }
    printk(KERN_INFO "[2013774ywl] length of message:%d\n", (int)len);
    return len;
}
```

8）加载模块函数 lkm_example_init:

该函数在将该内核模块载入内核时调用，主要是进行初始化的工作。

```
103 static int __init lkm_example_init(void)
104 {
105     //把我们宏定义的helloworld字符串写到msg_buffer当中
106     strncpy(msg_buffer, EXAMPLE_MSG, 31);
107     public_offset = 31;
108     //msg_ptr指向msg_buffer
109     msg_ptr = msg_buffer;
110     //注册字符设备
111     major_num = register_chrdev(0, DEVICE_NAME, &file_ops);
112     if (major_num < 0) {
113         printk(KERN_ALERT "[2013774ywl] Could not register device: %d\n",
    major_num);
114         return major_num;
115     } else {
116         printk(KERN_INFO "[2013774ywl] lkm_ywl_device module loaded with device
    major number %d\n", major_num);
117         return 0;
118     }
119 }
```

9）卸载模块函数 lkm_example_exit:

卸载模块函数在将模块移出内核时调用，主要作用就是进行一些注销工作。

```
121 static void __exit lkm_example_exit(void)
122 {
123     unregister_chrdev(major_num, DEVICE_NAME);
124     printk(KERN_INFO "Goodbye World YangWanli 2013774 !\n");
125 }
```

10） 正式将加载/卸载模块函数注册:

```
127 /* Register module functions */
128 module_init(lkm_example_init);
129 module_exit(lkm_example_exit);
```

以上则为该内核模块程序的全部内容。其大致思路就是，加载内核模块的时候申请一个字符设备，并且初始化缓冲区的内容和相关指针，同时重写

了该字符设备的打开、释放、读、写函数。因此操作该字符设备时，就会调用相关的自定义函数完成操作。当卸载内核模块时，释放该设备。

（4） Makefile 文件编写：

```
ModuleName = lkm_ywl_device
obj-m +=${ModuleName}.o
all:${ModuleName}.ko
${ModuleName}.ko:${ModuleName}.c
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
test:${ModuleName}.ko
	echo make test_ins	加载内核模块
	echo make test_mk	注册字符设备文件
	echo make test_read	读取文件内容
	echo make test_rm	卸载模块
test_ins:${ModuleName}.ko
	sudo dmesg -C
	sudo insmod ${ModuleName}.ko
	sudo dmesg | grep [2013774ywl]
test_mk:${ModuleName}.ko
	sudo bash -c "sudo mknod /dev/${ModuleName} c 237 0"
test_read:${ModuleName}.ko
	cat /dev/${ModuleName}
test_rm:${ModuleName}.ko
	sudo rmmod ${ModuleName}.ko
	sudo dmesg | grep [2013774ywl]
```

# 4. 具体步骤

（1） 按照上一小节的原理方法编写 lkm_ywl_device.c 文件和 Makefile 文件。

（2） 执行 make test：

```
wanliyang2013774@wanliyang2013774-virtual-machine:~/OSWorkSpace/lab12_code/template$ make test
make -C /lib/modules/6.0.9/build M=/home/wanliyang2013774/OSWorkSpace/lab12_code/template modules
make[1]: Entering directory '/home/wanliyang2013774/linux-6.0.9'
  CC [M]  /home/wanliyang2013774/OSWorkSpace/lab12_code/template/lkm_ywl_device.o
  MODPOST /home/wanliyang2013774/OSWorkSpace/lab12_code/template/Module.symvers
  CC [M]  /home/wanliyang2013774/OSWorkSpace/lab12_code/template/lkm_ywl_device.mod.o
  LD [M]  /home/wanliyang2013774/OSWorkSpace/lab12_code/template/lkm_ywl_device.ko
make[1]: Leaving directory '/home/wanliyang2013774/linux-6.0.9'
echo make test_ins    加载内核模块
make test_ins 加载内核模块
echo make test_mk     注册字符设备文件
make test_mk 注册字符设备文件
echo make test_read   读取文件内容
make test_read 读取文件内容
echo make test_rm     卸载模块
```

（3） 执行 make test_ins（加载内核模块）：

```
wanliyang2013774@wanliyang2013774-virtual-machine:~/OSWorkSpace/lab12_code/template$ make test_ins
sudo dmesg -C
[sudo] password for wanliyang2013774:
sudo insmod lkm_ywl_device.ko
sudo dmesg | grep [2013774ywl]
[31807.214983] [2013774ywl] lkm_ywl_device module loaded with device major number 237
```

（4） 执行 make test_mk（创建字符设备文件）：

```
wanliyang2013774@wanliyang2013774-virtual-machine:~/OSWorkSpace/lab12_code/template$ make test_mk
sudo bash -c "sudo mknod /dev/lkm_ywl_device c 237 0"
wanliyang2013774@wanliyang2013774-virtual-machine:~/OSWorkSpace/lab12_code/template$
```

在/dev 目录下，通过 ls 命令，可以看到确实有/lkm_ywl_device

```
wanliyang2013774@wanliyang2013774-virtual-machine:/dev$ ls
autofs           dmmidi      initctl         loop2          midi     rtc0      sr1      tty15  tty
block            dri         input           loop3          mqueue   sda       stderr   tty16  tty
bsg              ecryptfs    kmsg            loop4          net      sda1      stdin    tty17  tty
bus              fb0         lkm_ywl_device  loop5          null     sda2      stdout   tty18  tty
cdrom            fd          log             loop6          port     sda3      tty      tty19  tty
char             fd0         loop0           loop7          ppp      sg0       tty0     tty2   tty
console          full        loop1           loop8          psaux    sg1       tty1     tty20  tty
core             fuse        loop10          loop9          ptmx     sg2       tty10    tty21  tty
cpu              hidraw0     loop11          loop-control   pts      shm       tty11    tty22  tty
cpu_dma_latency  hpet        loop12          mapper         random   snapshot  tty12    tty23  tty
disk             hugepages   loop13          mcelog         rfkill   snd       tty13    tty24  tty
dma_heap         hwrng       loop14          mem            rtc      sr0       tty14    tty25  tty
wanliyang2013774@wanliyang2013774-virtual-machine:/dev$
```

（5） 执行 make test_read（读取字符设备的内容）：

```
Hello World YangWanli 2013774!
Hello World YangWanli 2013774!
Hello World YangWanli 2013774!
Hello World YangWanli 2013774!
Hello World YangWanli 2013774!
Hello World YangWanli 2013774!
Hello World YangWanli 2013774!
Hello World YangWanli 2013774!
Hello World YangWanli 2013774!
^Cmake: *** [Makefile:18: test_read] Interrupt

wanliyang2013774@wanliyang2013774-virtual-machine:~/OSWorkSpace/lab12_code/template$
```

循环打印设备内容。

（6） 通过 echo >>这种在末尾添加的方式向设备写入信息：

```
~/OSWorkSpace/lab12_code/template$ sudo bash -c "echo test echo way1 >> /dev/lkm_ywl_device"
```

```
Hello World YangWanli 2013774!
test echo way1
Hello World YangWanli 2013774!
test echo way1
Hello World YangWanli 2013774!
test echo way1
Hello World YangWanli 2013774!
test echo way1
Hello World Yan^Cmake: *** [Makefile:18: test_read] Interrupt

wanliyang2013774@wanliyang2013774-virtual-machine:~/OSWorkSpace/lab12_code/template$
```

可以发现确实在文件末尾写入信息。

再测试一次。

```
Hello World YangWanli 2013774!
test echo way1
test echo way1
Hello World YangWanli 2013774!
test echo way1
test echo way1
Hello World YangWanli 2013774!
test echo way1
test echo way1
^Cmake: *** [Makefile:18: test_read] Interrupt

wanliyang2013774@wanliyang2013774-virtual-machine:~/OSWorkSpace/lab12_code/template$
```

写入功能正常。

（7） 通过 echo >这种覆盖的方式向设备写入信息：

```
~/OSWorkSpace/lab12_code/template$ sudo bash -c "echo test echo way2 > /dev/lkm_ywl_device"
~/OSWorkSpace/lab12_code/template$
```

```
test echo way2
test echo way2
test echo way2
test echo way2
test echo way2
^Cmake: *** [Makefile:18: test_read] Interrupt

wanliyang2013774@wanliyang2013774-virtual-machine:~/OSWorkSpace/lab12_code/template$
```

确实实现了覆盖写入。

以上则测试了所需要实现的读写特定文件的功能。

（8） 卸载模块：

```
wanliyang2013774@wanliyang2013774-virtual-machine:~/OSWorkSpace/lab12_code/template$ make test_rm
sudo rmmod lkm_ywl_device.ko
```

……中间有大量输出信息

```
[32872.439726] Goodbye World YangWanli 2013774 !
wanliyang2013774@wanliyang2013774-virtual-machine:~/OSWorkSpace/lab12_code/template$
```

以上则完成了本次实验的所有内容。

## 5. 总结心得

（1） 通过本次实验，我进一步理解了操作系统内核模块的运行原理；

（2） 通过本次实验，我了解到了 Linux 系统的设备、文件的关联，加深了我对于 Linux 这种文件型系统的理解；

（3） 通过编写内核模块，操作内核空间和用户空间的变量，让我对用户态和内核态有了更深入的掌握；

（4） 通过本次实验，我对 Linux 系统的 cat 和 echo 命令有了更深刻的认知；

（5） 同时，本次实验让我更为了解 Makefile 文件的编写。

## 6. 参考资料

实验指导书《oslab12_README_modulechardevice.pdf》

# 7. 实验源代码

（1） lkm_ywl_device.c 文件：

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("YangWanli");
MODULE_DESCRIPTION("A simple example of linux module.");
MODULE_VERSION("0.01");

#define DEVICE_NAME "lkm_ywl_device"  //设备名称
#define EXAMPLE_MSG "Hello World YangWanli 2013774!\n"  //设备初始信息
#define MSG_BUFFER_LEN 128  //设备文件自定义存储大小

static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

static int major_num;  //设备号
static int device_open_count = 0;  //设备打开次数
static char msg_buffer[MSG_BUFFER_LEN];  //设备文件的缓冲区（存储区）
static char *msg_ptr;  //读文件的指针
static int public_offset;  //标识写文件的 offset

//声明相关的结构体，把四个函数组合起来
/* This structure points to all of the device functions */
static struct file_operations file_ops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};

//具体实现上面声明的函数

```

```
38. //读设备文件，每次命令行执行 cat /dev/lkm_ywl_device 时调用该函数
39. static ssize_t device_read(struct file *flip, char *buffer, size_t le
    n, loff_t *offset)
40. {
41.     //参数说明：调用该函数时，传入的 len 非常大(131072)，也就是默认读取所有
    内容
42.     int bytes_read = 0;  //记录读取的字符/字节数目
43.     printk(KERN_INFO "[2013774ywl] Read from /dev/%s c %d 0\n", DEVIC
    E_NAME, major_num);
44.     //该函数的作用就是把内核中 msg_buffer 中的内容读取到用户空间的 buffer
    中
45.     //注意，每次读完文件时，读文件的指针都指向文件的末尾，实际指向的内容是
    \0
46.     //因此需要重新把读文件指针 msg_ptr 指向缓存区
47.     if (*msg_ptr == 0) {
48.         msg_ptr = msg_buffer;
49.     }
50.     while (len && *msg_ptr) {
51.     //逐个字符读取
52.     put_user(*(msg_ptr++), buffer++);
53.         len--;
54.         bytes_read++;
55.     }
56.     printk("[2013774ywl] %d bytes read\n", bytes_read);
57.     return bytes_read;
58. }
59.
60. //写设备文件
61. static ssize_t device_write(struct file *flip, const char *buffer, si
    ze_t len, loff_t *offset)
62. {
63.     int value;  //写文件返回值
64.     int flag;  //标识是>>还是>打开文件
65.     flag = O_APPEND & flip->f_flags;
66.     if(O_APPEND == flag){ //说明是>>打开方式，需要添加文件内容
67.         printk("[2013774ywl] append\n");
68.         //public_offset 是全局变量，标志当前文件的末尾位移
69.         value = copy_from_user(msg_buffer + public_offset, buffer, le
    n);
70.         public_offset += len;
71.     }
72.     else if(flag == 0){  //说明是>打开方式，需要覆盖文件内容
73.         printk("[2013774ywl] overlay\n");
74.         memset(msg_buffer,'\0',MSG_BUFFER_LEN);  //清空文件内容
```

```
75.         value = copy_from_user(msg_buffer, buffer, len);
76.         public_offset = len;
77.     }
78.     printk(KERN_INFO "[2013774ywl] length of message:%d\n", (int)len)
    ;
79.     return len;
80. }
81.
82. //打开设备
83. static int device_open(struct inode *inode, struct file *file) {
84.     printk(KERN_INFO "[2013774ywl] Open /dev/%s c %d 0\n", DEVICE_NAM
    E, major_num);
85.     //已经打开则返回 -EBUSY
86.     if (device_open_count) {
87.         return -EBUSY;
88.     }
89.     device_open_count++;
90.     try_module_get(THIS_MODULE);
91.     return 0;
92. }
93.
94. //关闭设备
95. static int device_release(struct inode *inode, struct file *file)
96. {
97.     printk(KERN_INFO "[2013774ywl] Close /dev/%s c %d 0\n", DEVICE_NA
    ME,major_num);
98.     device_open_count--;
99.     module_put(THIS_MODULE);
100.       return 0;
101.   }
102.
103.   static int __init lkm_example_init(void)
104.   {
105.       //把我们宏定义的 helloworld 字符串写到 msg_buffer 当中
106.       strncpy(msg_buffer, EXAMPLE_MSG, 31);
107.       public_offset = 31;
108.       //msg_ptr 指向 msg_buffer
109.       msg_ptr = msg_buffer;
110.       //注册字符设备
111.       major_num = register_chrdev(0, DEVICE_NAME, &file_ops);
112.       if (major_num < 0) {
113.           printk(KERN_ALERT "[2013774ywl] Could not register device:
    %d\n", major_num);
114.           return major_num;
```

```
115.       } else {
116.           printk(KERN_INFO "[2013774ywl] lkm_ywl_device module loade
    d with device major number %d\n", major_num);
117.           return 0;
118.       }
119.   }
120.
121.   static void __exit lkm_example_exit(void)
122.   {
123.       unregister_chrdev(major_num, DEVICE_NAME);
124.       printk(KERN_INFO "Goodbye World YangWanli 2013774 !\n");
125.   }
126.
127.   /* Register module functions */
128.   module_init(lkm_example_init);
129.   module_exit(lkm_example_exit);
```

（2） Makefile 文件：

```
1.  ModuleName = lkm_ywl_device
2.  obj-m +=${ModuleName}.o
3.  all:${ModuleName}.ko
4.  ${ModuleName}.ko:${ModuleName}.c
5.      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
6.  test:${ModuleName}.ko
7.      echo make test_ins     加载内核模块
8.      echo make test_mk      注册字符设备文件
9.      echo make test_read    读取文件内容
10.     echo make test_rm      卸载模块
11. test_ins:${ModuleName}.ko
12.     sudo dmesg -C
13.     sudo insmod ${ModuleName}.ko
14.     sudo dmesg | grep [2013774ywl]
15. test_mk:${ModuleName}.ko
16.     sudo bash -c "sudo mknod /dev/${ModuleName} c 237 0"
17. test_read:${ModuleName}.ko
18.     cat /dev/${ModuleName}
19. test_rm:${ModuleName}.ko
20.     sudo rmmod ${ModuleName}.ko
21.     sudo dmesg | grep [2013774ywl]
```