《操作系统原理》实验报告

姓名	黄浩岩	学号		专业班级		时间	2020.12.15

一、实验目的

1）理解设备是文件的概念；

2）掌握 Linux 模块、驱动的概念和编程流程

3）Windows/Linux 下掌握文件读写基本操作

4）Windows/Linux 下体会缓存作用——提前读/延迟写

二、实验内容

1）编写一个 Linux 内核模块，并完成安装/卸载等操作。

2）编写 Linux 驱动程序并编程应用程序测试。功能：write 几个整数进去 read 出其和或差或最大值。

3）编写 Linux 驱动程序并编程应用程序测试。功能：有序读写内核缓冲区，返回实际读写字节数。

4）在 Win/Linux 下采用缓冲或非缓冲方式读写大文件，体会效率差异。

注：其中 1、2 必做，3、4 任选其一，此处选择了 3。

三、实验过程

1） 编写 Linux 内核模块（Task1）

该部分实验编写了一个简单的模块 myModule。

首先编写模块的 C 文件 myModule.c。文件开头包含了编写所用到的 Linux 相关的库，接着 MODULE_LICENSE 声明了模块许可证，用于模块的验证，MODULE_AUTHOR、MODULE_DESCRIPTION 声明了模块的作者和描述。随后，定义的 myModuleInit()为安装模块时调用的函数，myModuleExit()为删除模块时调用的函数，两者都使用了 printk()函数向内核缓冲区打印一段字符。此外，全局变量 test 使用 module_param()函数进行参数绑定，作为 myModuleInit()的参数，可在安装模块时进行参数传递。最后使用 module_init()函数和 module_exit()函数对上述两个函数进行注册。该部分具体代码如下：

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

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

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


MODULE_LICENSE("GPL");  //模块许可证声明
MODULE_AUTHOR("HHY");    //模块作者（可选）
MODULE_DESCRIPTION("A Simple Linux Module in OS Lab4-1");    //模块描述


int test;
module_param(test, int, 0644);  //模块参数绑定


//安装模块调用函数
static int __init myModuleInit(void) {
    printk("Hello World! This's hhy's module.\n");
    printk("test = %d\n", test);
    return 0;
}


//删除模块调用函数
static void __exit myModuleExit(void) {
    printk("Goodbye World! This's hhy's module.\n");
}


//模块函数注册
module_init(myModuleInit);
module_exit(myModuleExit);
```

然后编写用于编译该模块的 Makefile 文件，具体代码如下：

```
CONFIG_MODULE_SIG=n

# module name

obj-m:=myModule.o

# generate the path

CURRENT_PATH:=$(shell pwd)

# current linux kernel version

VERSION_NUM :=$(shell uname -r)

# linux path

LINUX_PATH    :=/usr/src/linux-headers-$(VERSION_NUM)

#compile object

all :

    make -C $(LINUX_PATH) M=$(CURRENT_PATH) modules

#clean

clean :

    make -C $(LINUX_PATH) M=$(CURRENT_PATH) clean
```

编写好上述两个文件后，在上述文件所在目录（Task1）使用命令“make”对模块进行编译，编译好后会生成 myModule.ko 等多个文件；接着使用命令“sudo insmod myModule.ko”安装模块 myModule 到内核空间。最后可以使用命令“sudo rmmod myModule”将该模块删除。过程中可以使用命令“dmesg”查看内核缓冲区中的信息，用于查看安装和删除模块时调用函数 printk 的输出结果。

2） 编写 Linux 求最大值驱动程序并测试（Task2）

该部分实验编写了一个简单的驱动程序 mydev，它的功能是可以向设备中写入两个整数然后可以读取出两个整数中的最大值。

首先编写驱动程序的 C 文件 mydev.c，其中有一部分代码与上述模块文件 myModule.c 中一致，在此不多赘述。

代码中定义了一个 file_operations 类型的结构体 mydev_fops，该函数的成员使用键值对进行赋值，表明了该驱动定义的函数与 Linux 的标准文件读取函数的对应关系（可理解为使用 Linux 文件函数操作该设备对应的文件时调用对应自定义的函数），由于该驱动程序主要涉及读写操作，因此此处仅列出了 read 和 write 与自定义函数 mydev_read 和 mydev_write 的映射关系。接下来为定义的全局变量：标识该设备的主设备号 mydev_major、设备名 mydev_name，主要用于设备的注册。int 类型数据所占字节数 int_size，方便该驱动对整数的处理。驱动程序的缓冲区大小 mydev_size，为两个整数类型的大小，因为该驱动只需要存最多 2 个整数。驱动程序缓冲区 mydev_buf，用于存取设备的数据。该部分代码具体如下：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <linux/uaccess.h> /* copy_from/to_user */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("HHY");
MODULE_DESCRIPTION("A Simple Linux Driver in OS Lab4-2");
//驱动可用函数
struct file_operations mydev_fops = {
    read: mydev_read,     //读数据
    write: mydev_write    //写数据
```

```
};
const int mydev_major = 61;          //主设备号
const char *mydev_name = "hhy's driver";       //设备名
const int mydev_size = 2 * sizeof(int);          //驱动程序缓冲区大小
const int int_size = sizeof(int);      //int 类型大小
char *mydev_buf;       //驱动程序缓冲区
```

与上述的模块相同，驱动程序也需要安装驱动和删除驱动时的函数 mydev_init()和 mydev_exit()，并使用 module_init()和 module_exit()进行注册。其中，在安装驱动调用的函数 mydev_init()中，使用了 register_chrdev()函数来注册一个字符型的设备，需要传入参数主设备号 mydev_major，设备名 mydev_name 以及定义的驱动函数映射的结构体 mydev_fops；此外，使用 kmalloc()函数对内核空间的驱动程序动态分配缓冲区的空间。而在删除驱动调用的函数 mydev_exit()中，使用了 unregister_chrdev()对注册的设备进行删除，同时使用 kfree()函数对缓冲区的动态内存进行释放。

```
//安装驱动
static int __init mydev_init(void) {
    int ret;
    //注册设备
    ret = register_chrdev(mydev_major, mydev_name, &mydev_fops);
    if(ret < 0) {
        printk(KERN_EMERG"mydev: cannot obtain major number %d\n",
        mydev_major);
        return ret;
    }
    //动态分配设备缓冲区
    mydev_buf = kmalloc(mydev_size, GFP_KERNEL);
    if(!mydev_buf) {
```

```
        ret = -ENOMEM;
        return ret;
    }
    //初始化缓冲区全 0
    memset(mydev_buf, 0, mydev_size);
    printk("<1>mydev: Inserting mydev\n");
    return 0;
}

//删除驱动
static void __exit mydev_exit(void) {
    //取消注册设备
    unregister_chrdev(mydev_major, mydev_name);
    //释放设备缓冲区
    if(mydev_buf) kfree(mydev_buf);
    printk("<1>mydev: Removing mydev\n");
}
//注册函数
module_init(mydev_init);
module_exit(mydev_exit);
```

然后为用于实现读写功能的函数 mydev_read()和 mydev_write()。

对于写入数据的函数 mydev_write()，形参 len 记录了从用户缓冲区需要写入驱动缓冲区的数据字节数。由于该驱动只需要最多 2 个整数的大小，因此当 len 大于驱动缓冲区的大小时要更新 len。然后使用 Linux 提供的函数 copy_from_user()将用户空间的数据复制到内核空间的驱动缓冲区。最后返回实际写入数据字节数。具体代码如下：

```
//写入数据：向设备中写入最多 2 个整数
static ssize_t mydev_write(struct file *fp, const char __user *buf,
                                        size_t len, loff_t *pos) {
    //最多向设备中写入 2 个整数大小
    if(len > mydev_size) len = mydev_size;
    //将数据从用户缓冲区复制到设备缓冲区
    if(copy_from_user(mydev_buf, buf, len)) return -EFAULT;
    printk("<1>mydev: write %d, %d\n", (int)mydev_buf[0], (int)mydev_buf[int_size]);
    //返回实际写入数据字节数
    return len;
}
```

对于读取数据函数 mydev_read()，首先通过强制类型转换得到位于驱动缓冲区中的 2 个整数 a、b，并比较大小，记录较大的数在缓冲区中的偏移量（用于后续数据复制）。然后使用 Linux 提供的函数 copy_to_user()将位于驱动缓冲区中较大的整数复制到用户空间中。最后返回实际读取的数据字节数。具体代码如下：

```
//读取数据：设备中 2 个数据中的最大值
static ssize_t mydev_read(struct file *fp, char __user *buf,
                                        size_t len, loff_t *pos) {
    //比较设备中 2 个数据的大小，并设置偏移量
    int a = (int)mydev_buf[0], b = (int)mydev_buf[int_size];
    int offset = (a >= b) ? 0 : int_size;
    //将数据从内核态复制到用户态缓冲区
    if(copy_to_user(buf, mydev_buf + offset, int_size))
        return -EFAULT;
    printk("<1>mydev: Read %d\n", (int)mydev_buf[offset]);
```

```
    //返回实际读取数据字节数
    return int_size;
}
```

然后编写用于编译该驱动的 Makefile 文件，Makefile 中除了生成的模块名为 mydev.ko 外其余与上述模块的 Makefile 完全相同。

编写好上述两文件后，同样使用命令“make”对驱动进行编译，使用命令“sudo insmod mydev.ko”安装驱动，使用命令“sudo rmmod mydeve”将该驱动删除。过程中可以使用命令“dmesg”查看内核缓冲区中的信息。

需要注意的一点时，要真正使用驱动程序进行读写，需要建立该驱动程序设备对应的文件，使用命令“sudo mknod /dev/mydev c 61 0”，其中 mydev 为驱动模块的名称，c 表明为字符型设备，61 为主设备号，0 为次设备号。

这样就可以利用 Linux 的 write，read 接口读写设备文件“/dev/mydev”中的数据了。

3） 编写 Linux 读取字符串驱动程序并测试（Task3）

该部分实验编写了一个简单的驱动程序 mydev2，它的功能是：可以向设备中写入若干字符并在下次写入时数据紧接着上次写入的结尾；可以读取若干字符，并在下次读取时紧跟上次读取的结尾。

首先编写驱动程序的 C 文件 mydev2.c，其中全局变量设备号 mydev2_major、设备名 mydev2_name、驱动缓冲区 mydev2_buf，安装驱动函数 mydev_init()、删除驱动函数 mydev2_exit()以及相应的注册 module_init()和 module_exit()均与 Task2 中 mydev.c 中定义的一致，在此不多赘述。

根据功能要求，在全局变量中调整了驱动缓冲区大小 mydev2_max_size，设置为 64 字节；同时设置一个记录当前缓冲区中数据字节数的变量 mydev2_size，其中 loff_t 类型为 Linux 定义的类型，本质上是 long int，此处主要是为了方便和缓冲区偏移量的数据类型相统一。该部分代码如下：

```
const loff_t mydev2_max_size = 64;   //驱动程序缓冲区最大值
loff_t mydev2_size = 0;         //当前缓冲区数据字节数
```

对于向设备写入字符的函数 mydev2_write()，首先计算驱动缓冲区还能写入的字符个数 left，其中入口参数*pos 是指向设备文件指针偏移量的指针，使用缓冲区上限大小减去当前文件指针的偏移量即可计算出最多还能写入的字符个数。若 left 为 0 及缓冲区已经写满不能继续写入了，就直接返回 0，表示实际写入了 0 字节数据；若要写入设备的字符数超过了可写入的个数上限，则更新 len=left，表示只写入可写入个数的字符。接着利用 copy_from_user() 函数将字符从用户空间复制到驱动缓冲区。由于下次写入时要接着上次写入的位置，因此要更新设备文件的文件指针*pos，将其后移 len 个字节。同时，更新当前缓冲区存入数据的字节数 mydev2_size。最后返回实际写入字节数。具体代码如下：

```
//写数据：  写入字符
static ssize_t mydev2_write(struct file *fp, const char __user *buf,
                                        size_t len, loff_t *pos) {
    //计算设备缓冲区还能写入的数据大小
    int left = mydev2_max_size - *pos;
    //若文件指针已经指到缓冲区末尾则不能写入，返回
    if(left == 0) return 0;
    //若写入数据超过可写入上限，则更新写入字节数
    else if(len > left) len = left;
    //从用户缓冲区复制数据到设备缓冲区
    if(copy_from_user(mydev2_buf+(*pos), buf, len))
        return -EFAULT;
    printk("<1>mydev2: Write %ldB, left %ldB\n", len, left-len);
    //文件指针后移写入字节数个单位
    *pos += len;
    //更新当前设备缓冲区中数据字节数
    if(*pos > mydev2_size) mydev2_size = *pos;
    //返回实际写入字节数
```

```
    return len;
}
```

对于从设备读取字符的函数 mydev2_read()，首先将利用当前设备中数据字节数减去当前设备文件指针偏移量得到还能从设备中读取的字节数 left。若 left 为 0 则表明文件指针已经指到数据末尾，不能读取了，返回 0，表明读取了 0 个字节。若用户要读取的长度 len 大于可读取上限，则更读取长度为读取上限 len=left。接着利用函数 copy_to_user()将数据从驱动缓冲区赋值到用户空间。由于下次读取时要接着上次读取的位置，因此要更新设备文件的文件指针*pos，将其后移 len 个字节。最后返回实际读取的字节数。具体代码如下：

```
static ssize_t mydev2_read(struct file *fp, char __user *buf,
                                    size_t len, loff_t *pos) {
    //计算设备中数据还能读取的大小
    int left = mydev2_size - *pos;
    //若文件指针已经指到数据末尾则不能读取，返回
    if(left == 0) return 0;
    //若读取数据超过可读取上限，则更新读取字节数
    else if(len > left) len = left;
    //从设备缓冲区复制数据到用户缓冲区
    if(copy_to_user(buf, mydev2_buf+(*pos), len))
        return -EFAULT;
    printk("<1>mydev2: Read %ldB, left %ldB\n", len, left-len);
    //文件指针后移读取字节数个单位
    *pos += len;
    //返回实际读取字节数
    return len;
}
```

由于读写操作文件指针都是紧跟在上次的位置之后，因此为了方便调整文件指针，此处又添加了一个函数 mydev2_llseek()，它对应 Linux 的 lseek()函数，用于重新设置文件指针的偏移。在 mydev2_fops 结构体中加入 llseekd 到 mydev2_llseek 的映射。该函数有三个形参，fp 表示设备文件结构体，offset 为要设置的偏移量，whence 为偏移的参照位置。在函数中，定义 newpos 作为新的文件指针的偏移量，先初始化为 0。然后判断偏移量的参照位置，根据 Linux 定义的规则，分三种情况：SEEK_SET 表示以文件开头为基准，因此此时偏移量 offset 的值即为新指针的偏移量；SEEK_CUR 表示当前文件指针的位置，因此新偏移量为原偏移量加上 offset 的值；SEEK_END 表示文件结尾（即存入驱动缓冲区数据的末尾），新偏移量为结尾位置加上偏移量。若新的文件指针偏移量小于 0 或者超过当前数据长度，则返回错误码，否则更新文件指针偏移量并返回。该部分具体代码如下：

```
//文件指针移动
loff_t mydev2_llseek(struct file *fp, loff_t offset, int whence) {
    //新文件指针位置
    loff_t newpos = 0;
    //判断起点
    switch (whence) {
    case SEEK_SET:   //文件头
        newpos = offset;
        break;
    case SEEK_CUR:   //文件指针当前位置
        newpos = fp->f_pos + offset;
        break;
    case SEEK_END:   //文件尾
        newpos = mydev2_size + offset;
        break;
    default:
        return -EINVAL;
```

```
    }
    //若文件指针位置小于 0 或大于数据长度则报错
    if(newpos < 0 || newpos >= mydev2_size) return -EINVAL;
    //重置文件指针位置并返回
    fp->f_pos = newpos;
    printk("<1>mydev2: f_pos set to %lld\n", newpos);
    return newpos;
}
```

随后的编写 Makefile 文件，安装、删除驱动已经建立设备对应的文件“/dev/mydev2”均和上述 Task2 中 mydev 驱动的相同，在此不多赘述。

这样就可以利用 Linux 的 write，read 接口向设备文件“/dev/mydev2”中读写字符，同时可以利用 lseek 函数调整文件指针的偏移位置。

四、实验结果

1） 编写 Linux 内核模块（Task1）

使用命令“sudo insmod myModule.ko test=123”安装模块 myModule（其中 test=123 为传递的参数）后使用命令“sudo rmmod myModule”将该模块删除，然后使用命令“dmesg”查看内核缓冲区中的信息，结果如图 4-1 所示，可以看到安装和删除模块时输出的信息。

```
[   29.499269] rfkill: input handler disabled
[  342.430059] myModule: loading out-of-tree module taints kernel.
[  342.430087] myModule: module verification failed: signature and/or required k
ey missing - tainting kernel
[  342.430419] Hello World! This's hhy's module.
[  342.430420] test = 123
[  399.268569] Goodbye World! This's hhy's module.
```

图 4-1 安装删除模块的内核缓冲区信息

2） 编写 Linux 求最大值驱动程序并测试（Task2）

对于该驱动程序 mydev，编写了如下的代码 test.c 进行测试。主要是使用 Linux 提供的文件函数接口进行操作：使用 open()函数打开设备文件“/dev/mydev”，然后接受用户输入的 2 个整数，使用 write()函数将这两个整数到设备，最后使用 read()函数读取到两个数中的较大者并输出。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
	int a[2], maxx;
	int fd = open("/dev/mydev", O_RDWR);
	int len;
	if(fd < 0) {
		perror("Open dev fail\n");
		return 0;
	}
	printf("Input 2 int:");
	scanf("%d%d", &a[0], &a[1]);
	len = write(fd, a, 2*sizeof(int));
	if(len == 2*sizeof(int)) printf("Write: %d, %d\n", a[0], a[1]);
	len = read(fd, &maxx, sizeof(int));
	if(len == sizeof(int)) printf("Read max one: %d\n", maxx);
	close(fd);
	return 0;
}
```

编译测试代码后使用 root 权限运行，运行结果如图 4-2。然后使用“dmesg”查看内核缓冲区的输出信息，结果如图 4-3。可以看到安装、删除驱动后的输出，以及写入了两个整数并输出了最大值。

```
hhy@hhy-virtual-machine:~/os/lab4-2$ gcc -o test test.c
hhy@hhy-virtual-machine:~/os/lab4-2$ sudo ./test
Input 2 int:88 100
Write: 88, 100
Read max one: 100
```

图 4-2 测试驱动 mydev 的输出结果

```
[31741.067389] IPv6: ADDRCONF(NETDEV_CHANGE): ens33: link becomes read
[33608.158711] <1>mydev: Inserting mydev
[33649.211963] <1>mydev: write 88, 100
[33649.211978] <1>mydev: Read 100
[33679.848589] <1>mydev: Removing mydev
hhy@hhy-virtual-machine:~/os/lab4-2$
```

图 4-3 测试驱动 mydev 的内核缓冲区信息

3） 编写 Linux 读取字符串驱动程序并测试（Task3）

对于该驱动程序 mydev2 编写了如下的代码 test.c 进行测试。同样是使用 Linux 提供的文件函数接口进行操作：使用 open()函数打开设备文件“/dev/mydev2”，然后不断接受用户输入的字符串，并使用 open()读入到设备文件中，直到用户终止（遇到 EOF）。然后使用 lseek()函数将文件指针偏移恢复到设备文件头，接着允许用户不断输入一个整数，表示从设备文件中要读取的字符数，接着使用 write()函数从设备文件中读取字符，直到遇到 EOF 终止。

```
#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <fcntl.h>


char src[256];

char dest[256];

int pc = 0;
```

```
int main() {
	int len, cnt;
	int pos;
	int fd = open("/dev/mydev2", O_RDWR);
	if(fd < 0) {
		printf("Open dev2 fail\n");
		return 0;
	}
	printf("Write string:\n");
	while(scanf("%s", src) != EOF) {
		len = strlen(src);
		cnt = write(fd, src, len);
		printf("Write to dev2 %dB\n", cnt);
	}
	printf("pos: %ld\n", lseek(fd, 0, SEEK_SET));
	printf("Read bytes of string:\n");
	while(scanf("%d", &len) != EOF) {
		cnt = read(fd, dest, len);
		dest[cnt] = 0;
		printf("Read from dev2 %dB: %s\n", cnt, dest);
	}
	close(fd);
	return 0;
}
```

编译测试代码后使用 root 权限运行，运行结果如图 4-4，可以看到当输入的字符超过了 64 字节后写入到设备中的字符数就变为了 0，而同样读取的字符数超过 64 字节后则不能再从设备中读取到字符。最后使用“dmesg”查看内核缓冲区的输出信息，结果如图 4-5。

```
hhy@hhy-virtual-machine:~/os/lab4-3$ sudo ./test
Write string:
abcdef
Write to dev2 6B
1234567890
Write to dev2 10B
aafjoifj&(&F)^^^F^))Faf5468af48afaf77efa89(&^%**
Write to dev2 48B
444
Write to dev2 0B
pos: 0
Read bytes of string:
4
Read from dev2 4B: abcd
10
Read from dev2 10B: ef12345678
20
Read from dev2 20B: 90aafjoifj&(&F)^^^F^
20
Read from dev2 20B: ))Faf5468af48afaf77e
20
Read from dev2 10B: fa89(&^%**
20
Read from dev2 0B:
```

图 4-4 测试驱动 mydev2 的输出结果

```
[35418.240451] <1>mydev2: Inserting mydev2
[35434.835763] <1>mydev2: Write 6B, left 58B
[35438.162011] <1>mydev2: Write 10B, left 48B
[35452.215207] <1>mydev2: Write 48B, left 0B
[35461.655545] <1>mydev2: f_pos set to 0
[35463.315763] <1>mydev2: Read 4B, left 60B
[35465.794540] <1>mydev2: Read 10B, left 50B
[35468.412681] <1>mydev2: Read 20B, left 30B
[35469.371167] <1>mydev2: Read 20B, left 10B
[35471.172362] <1>mydev2: Read 10B, left 0B
[35503.007220] <1>mydev2: Removing mydev2
hhy@hhy-virtual-machine:~/os/lab4-3$
```

图 4-5 测试驱动 mydev2 的内核缓冲区信息

五、体会

1. 本次实验学习并实践了如何编写 Linux 的模块。包括如何自定义安装和删除模块的函数并使用 module_int()和 module_exit()函数进行注册，以及使用 module_param()函数将参数绑定到模块来为模块传递参数。此外，也学习了编写 Makefile 来编译当前模块。

2. 学习并实践了如何编写 Linux 驱动程序，通过任务二、三简单熟悉了 Linux 驱动的编写规则及相关的 Linux 函数，包括使用 file_operations 类型结构体定义 Linux 接口函数与自定义驱动函数的映射关系；使用 register_chrdev()函数注册字符型设备、unregister_chrdev()函数取消注册字符型设备；使用 kmalloc()和 kfree()进行内核空间的动态内存分配和释放，编写自定义函数 mydev_read()和 mydev_write()对设备进行读写，其中使

用了 copy_to_user()和 copy_from_user()来进行内核空间中驱动缓冲区和用户空间进行交互，以及如何处理缓冲区的数据。此外在任务三进一步实践了如何以文件的角度进行驱动函数的编写，包括如何更新设备文件指针的偏移量，如何计算出实际的读写字节数，以及如何调整文件偏移指针的位置。

3．通过实验及对驱动设备的测试更加深刻的理解到设备被抽象成了文件这一概念。在对驱动设备进行测试时，基本上就是将设备作为文件来处理，使用 Linux 提供的文件函数 open()、read()、write()及 lseek()等函数对设备中的数据进行处理，与普通文件并无太大区别。而通过测试也让我在编写驱动自定义函数时以一个文件的视角进行编写，使得编写时如何与普通文件保持一致。

4．遇到问题及解决：

1）问题：编写模块 C 文件时显示“linux/init.h”等 linux 相关头文件不存在。

解决：使用命令“sudo apt-get install linux-headers-`uname -r`”来下载当前 Linux 内核相关的头文件即可。虽然之后 VScode 中可能仍然显示部分 Linux 头文件不存在，但在实际编译中并未出现问题。

2）问题：安装好编写的驱动后并不能打开驱动对应的设备文件。

解决：在安装模块后 Linux 并不会创建对应的设备文件，需要我们自己创建，使用命令“sudo mknod /dev/<设备名> <设备类型> <主设备号> <次设备号>”进行设备文件的创建，之后就可以正常读取设备文件了。

3）问题：在任务三中，当向设备中写入了 64 字节的字符后，无法继续从设备读取数据。

解决：出现上述问题的原因经过调查发现是因为文件指针偏移已经指到了设备文件结尾（即定义的驱动缓冲区的最后），而读取时文件指针不能再向后移动了，因此无法从设备中进行字符读取。解决方法就是从重新更新设备文件指针的偏移量，使其移动到文件头部，即在测试代码中使用 lseek()函数，但是对于设备文件而言，该函数需要用户自己进行编写定义，因此在 mydev2 驱动中加入了函数 mydev2_llseek()用于移动文件偏移指针。最终在测试时，读取字符后，使用 lseek()函数重置一下文件偏移指针，即可正常从设备文件中读取数据。