《操作系统原理》实验报告(四)

姓名		学号		专业班级		时间	

一、实验目的

1）掌握Linux驱动概念和编程流程，理解设备是文件的概念。
2）掌握Linux下文件读写操作
3）了解索引文件系统和索引节点的概念

二、实验内容

1）编写Linux驱动程序（字符类型或杂项类型）和相应的测试程序。驱动程序的功能：(a)写入一个整数表示设备的状态；(b)读出设备的状态；(c)输入两个整数和要做的操作的指示码（用1,2,3分别表示求和，求差，求最大值），分别返回和、差和最大值。要求使用IOCTL接口。
2）编写Linux驱动程序（字符类型或杂项类型）和相应的测试程序。测试程序的功能：使用open函数先后打开该设备和2~5个文本文件（自己编造2~5个文本文件）。驱动程序的功能：打印测试程序进程PCB的成员信息，越多越好，其中必须包括：内核版本，进程ID，程序名，打开的文件列表，占用内存信息等。
3）编造若干大小不等的文本文件并创建它们的软链接和硬链接，自己设计各种验证过程，感性认识inode索引节点和索引文件系统。

三、实验环境和核心代码

3.1 编写Linux驱动及测试程序

> 开发环境：Ubuntu 20.04，内核版本：5.15.71，编辑工具：vim & vscode，编译器：gcc

3.1.1 驱动程序代码：

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <linux/ioctl.h>

#define DEV_NAME "my_dev"
#define MAGIC_NUM 'k'
#define IOCTL_SET_ARGS _IOW(MAGIC_NUM, 1, int *)
#define IOCTL_GET_RESULT _IOR(MAGIC_NUM, 2, int *)

static dev_t dev;
static struct cdev cdev;
static int state;

static int my_open(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "my_dev: Device opened\n");
```

```
    return 0;
}

static int my_release(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "my_dev: Device released\n");
    return 0;
}

static ssize_t my_write(struct file *file, const char __user *buf, size_t count,
loff_t *ppos)
{
    int ret;
    if (count != sizeof(int))
    {
        return -EINVAL;
    }
    ret = copy_from_user(&state, buf, count);
    if (ret)
    {
        return -EFAULT;
    }
    printk(KERN_INFO "my_dev: State set to %d\n", state);
    return count;
}

static ssize_t my_read(struct file *file, char __user *buf, size_t count, loff_t
*ppos)
{
    int ret;
    if (count != sizeof(int))
    {
        return -EINVAL;
    }
    ret = copy_to_user(buf, &state, count);
    if (ret)
    {
        return -EFAULT;
    }
    printk(KERN_INFO "my_dev: State read as %d\n", state);
    return count;
}

static long my_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    int args[3], ret;
    int op_code = (cmd == IOCTL_SET_ARGS) ? 1 : 2;

    switch (cmd)
    {
    case IOCTL_SET_ARGS:
    case IOCTL_GET_RESULT:
```

```
        ret = copy_from_user(args, (void __user *)arg, sizeof(args));
        if (ret)
        {
            return -EFAULT;
        }
        printk(KERN_INFO "my_dev: IOCTL cmd=%u, arg1=%d, arg2=%d, op=%d\n",
               cmd, args[0], args[1], args[2]);
        break;
    default:
        return -ENOTTY;
    }

    switch (op_code)
    {
    case 1: /* perform the operation */
        switch (args[2])
        {
        case 1: /* add */
            state = args[0] + args[1];
            break;
        case 2: /* subtract */
            state = args[0] - args[1];
            break;
        case 3: /* maximum */
            state = (args[0] > args[1]) ? args[0] : args[1];
            break;
        default:
            return -EINVAL;
        }
        break;
    case 2: /* return the result */
        ret = copy_to_user((void __user *)arg, &state, sizeof(state));
        if (ret)
        {
            return -EFAULT;
        }
        break;
    }
    return 0;
}

static struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .release = my_release,
    .write = my_write,
    .read = my_read,
    .unlocked_ioctl = my_ioctl,
};

static int __init my_init(void)
{
```

```
    int ret;
    ret = alloc_chrdev_region(&dev, 0, 1, DEV_NAME);
    if (ret < 0)
    {
        printk(KERN_ERR "my_dev: Failed to allocate chrdev region\n");
        return ret;
    }

    cdev_init(&cdev, &my_fops);
    cdev.owner = THIS_MODULE;

    ret = cdev_add(&cdev, dev, 1);
    if (ret < 0)
    {
        printk(KERN_ERR "my_dev: Failed to add cdev\n");
        unregister_chrdev_region(dev, 1);
        return ret;
    }

    printk(KERN_INFO "my_dev: Initialized\n");
    return 0;
}

static void __exit my_exit(void)
{
    cdev_del(&cdev);
    unregister_chrdev_region(dev, 1);
    printk(KERN_INFO "my_dev: Exited\n");
}

module_exit(my_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("dekrt");
MODULE_DESCRIPTION("A simple character driver for testing ioctl");
```

3.1.2 测试程序代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#define DEV_PATH "/dev/my_dev"
#define IOCTL_MAGIC 'k'
#define IOCTL_SET_ARGS _IOW(IOCTL_MAGIC, 1, int *)
#define IOCTL_GET_RESULT _IOR(IOCTL_MAGIC, 2, int *)

int main(int argc, char *argv[])
```

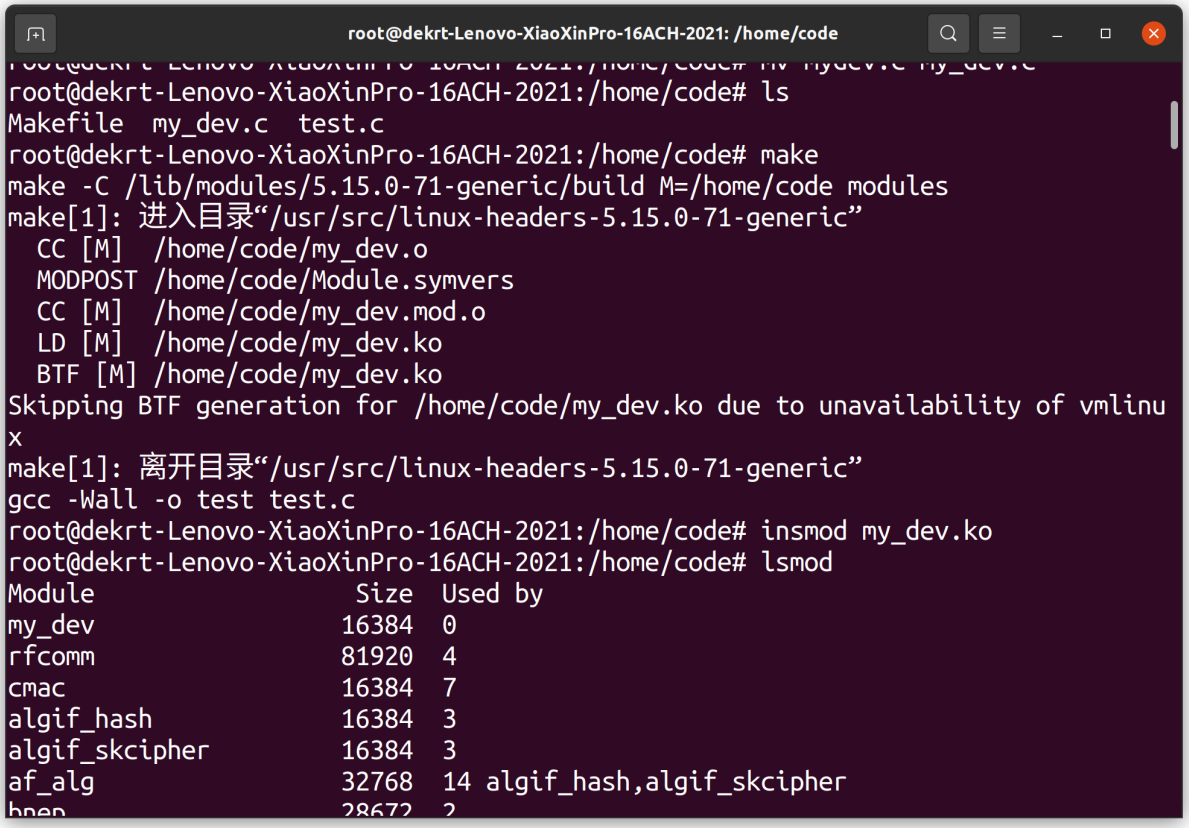
```
{
    int fd, ret, args[3], result;
    puts(">>> please input num_1, num_2 and the operation number:");
    puts(">>> operation number: 1 for add, 2 for minus, 3 for maximize");
    for(int i = 0; i < 3; i++)
    {
        scanf("%d", &args[i]);
    }
    fd = open(DEV_PATH, O_RDWR);
    if (fd < 0)
    {
        perror("open");
        exit(EXIT_FAILURE);
    }

    ret = ioctl(fd, IOCTL_SET_ARGS, args);
    if (ret < 0)
    {
        perror("ioctl set args");
        goto error;
    }

    ret = ioctl(fd, IOCTL_GET_RESULT, &result);
    if (ret < 0)
    {
        perror("ioctl get result");
        goto error;
    }

    printf("Result: %d\n", result);

    close(fd);
    exit(EXIT_SUCCESS);

error:
    close(fd);
    exit(EXIT_FAILURE);
}
```

3.1.3 Makefile

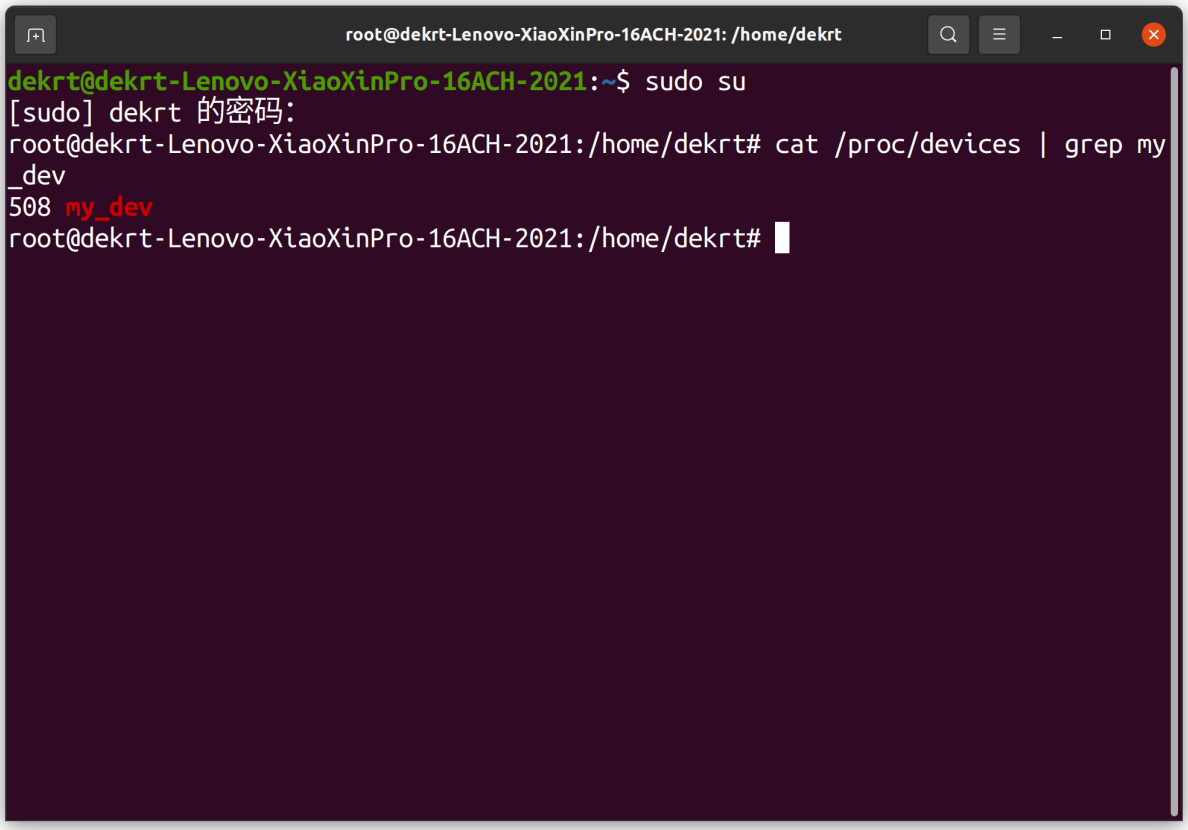
```
obj-m += my_dev.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
    gcc -Wall -o test test.c

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
    rm -f test
```

3.1.4 安装过程

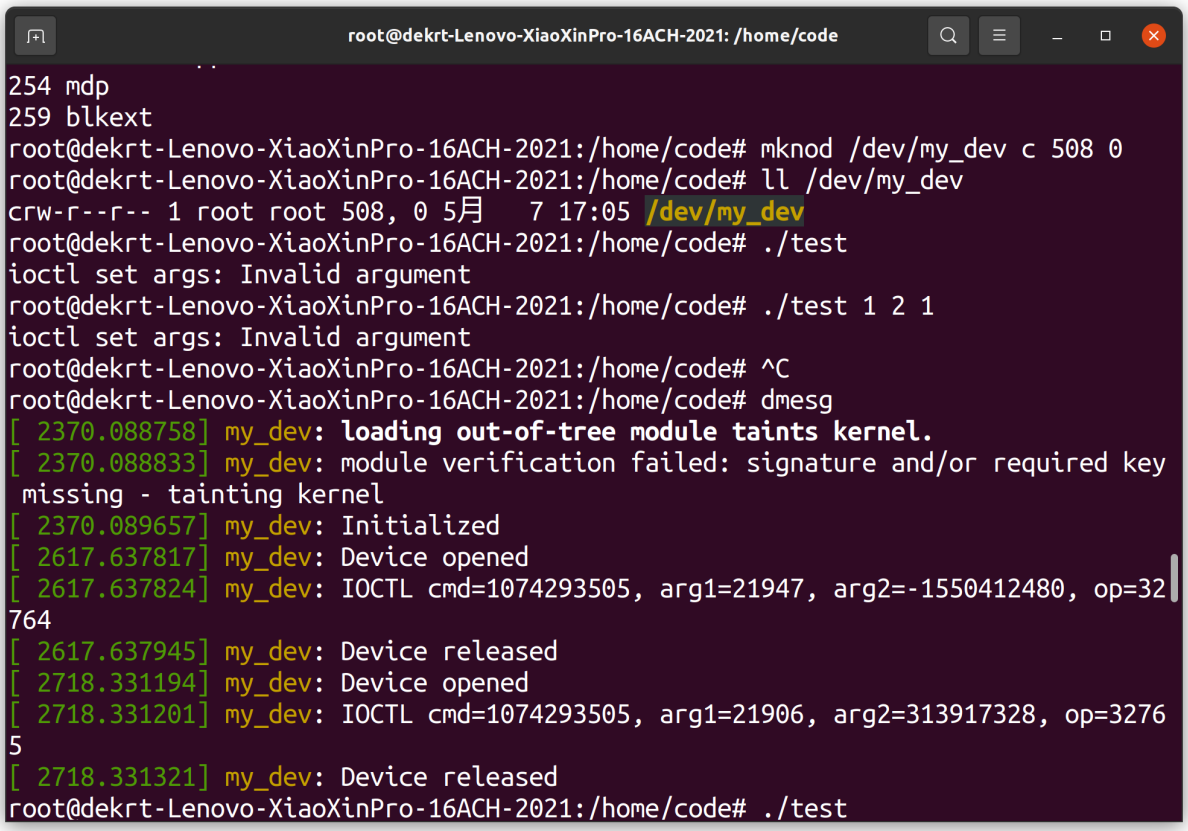
使用`sudo su`命令进入root模式，在代码目录使用`make`命令进行编译，编译完成后使用`insmod my_dev.ko`命令安装编译好的模块：



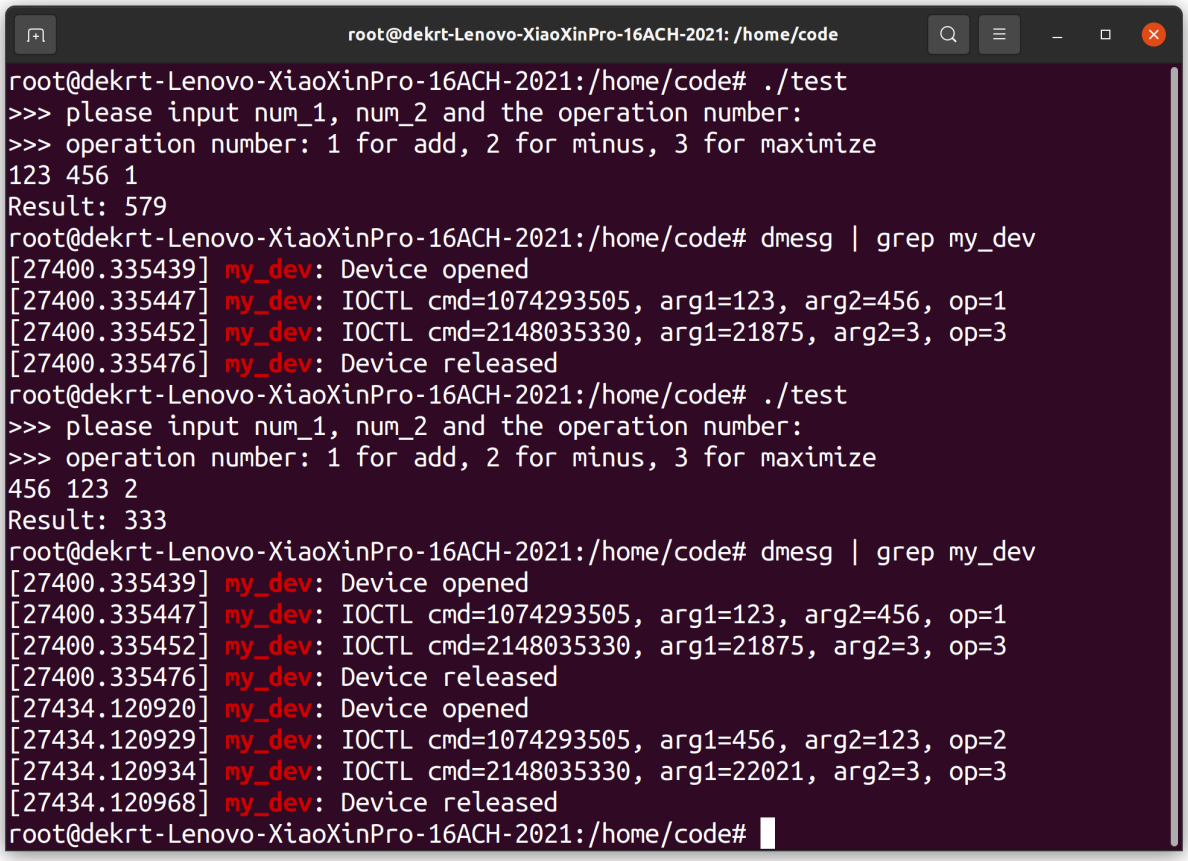
使用`lsmod`命令，可以看到my_dev模块被成功安装。再使用`cat /proc/devices`命令查看设备调用的的设备号，可以看到`my_dev`的设备号是508



使用`mknod /dev/my_dev c 508 0`命令创建相应设备，使用`ll`命令验证设备创建成功。



运行测试程序，可以发现其成功运行



3.2 Linux下创建软连接与硬连接

> 开发环境：Ubuntu 20.04，内核版本：5.15.71，编辑工具：vim & Vscode

1. 首先在命令行中创建一个名为"Test"的目录:

```
mkdir Test
```

2. 进入"Test"目录，使用以下命令创建一个名为"file1"的文件，并写入一些内容：

```
cd Test
echo "This is file 1" > file1
```

3. 使用以下命令创建一个名为"hardlink1"的硬链接：

```
ln file1 hardlink1
```

4. 以下命令创建一个名为"symlink1"的软链接：

```
ln -s file1 symlink1
```

5. 用以下命令查看"file1"、"hardlink1"和"symlink1"的详细信息：

```
ls -li
```

这个命令会输出包含文件索引节点号（inode）的列表，以及每个文件的权限、链接数、所有者和大小等信息。我们可以发现，"file1"、"hardlink1"和"symlink1"的inode号是一样的，说明它们都指向同一个数据块。

6. 接下来，我们尝试在"file1"中添加一些内容，然后再次查看这三个文件的信息：

```
echo "This is some more content" >> file1
ls -li
```

现在我们会发现，三个文件的大小都增加了，并且它们的inode号仍然相同。

7. 接着，我们使用以下命令创建一个名为"file2"的新文件：

```
echo "This is file 2" > file2
```

8. 然后，我们使用以下命令删除"file1"：

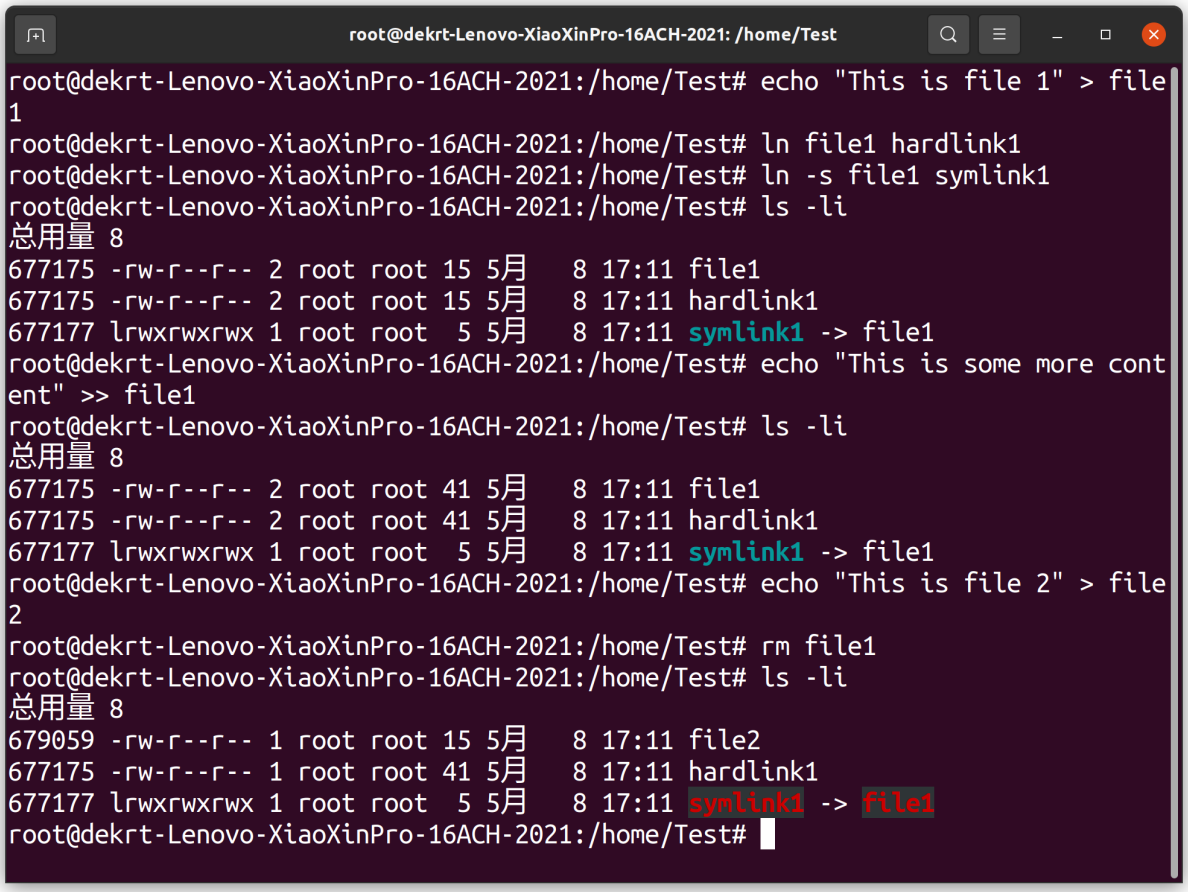
```
rm file1
```

9. 最后，我们再次查看"hardlink1"和"symlink1"的详细信息：

```
ls -li
```

我们会发现，"hardlink1"仍然存在，而"symlink1"则已经无法找到目标文件，因为它是指向"file1"的软链接，而"file1"已经被删除了。

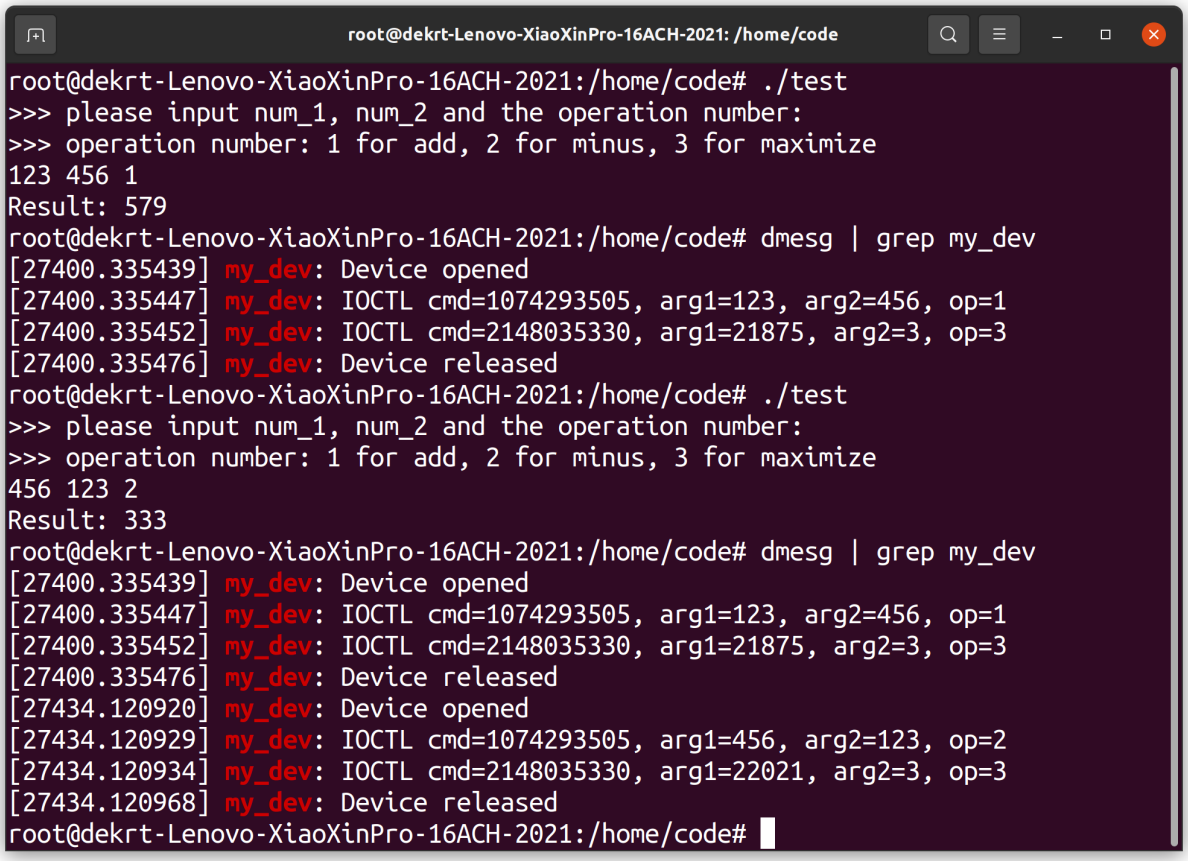
通过这个实验，我们可以感性认识索引文件和索引节点的概念。在Linux中，每个文件都有一个唯一的inode号，它包含了文件的元数据（如文件所有者、文件权限、文件大小等）以及指向数据块的指针。当我们创建一个硬链接时，实际上是在文件系统中创建了一个新的目录项，它指向相同的inode。这个新的目录项与原始文件目录项没有区别，它们是等价的。而软链接则是一个特殊的文件，它包含了指向目标文件的路径，当我们访问软链接时，实际上是通过路径找到了目标文件。



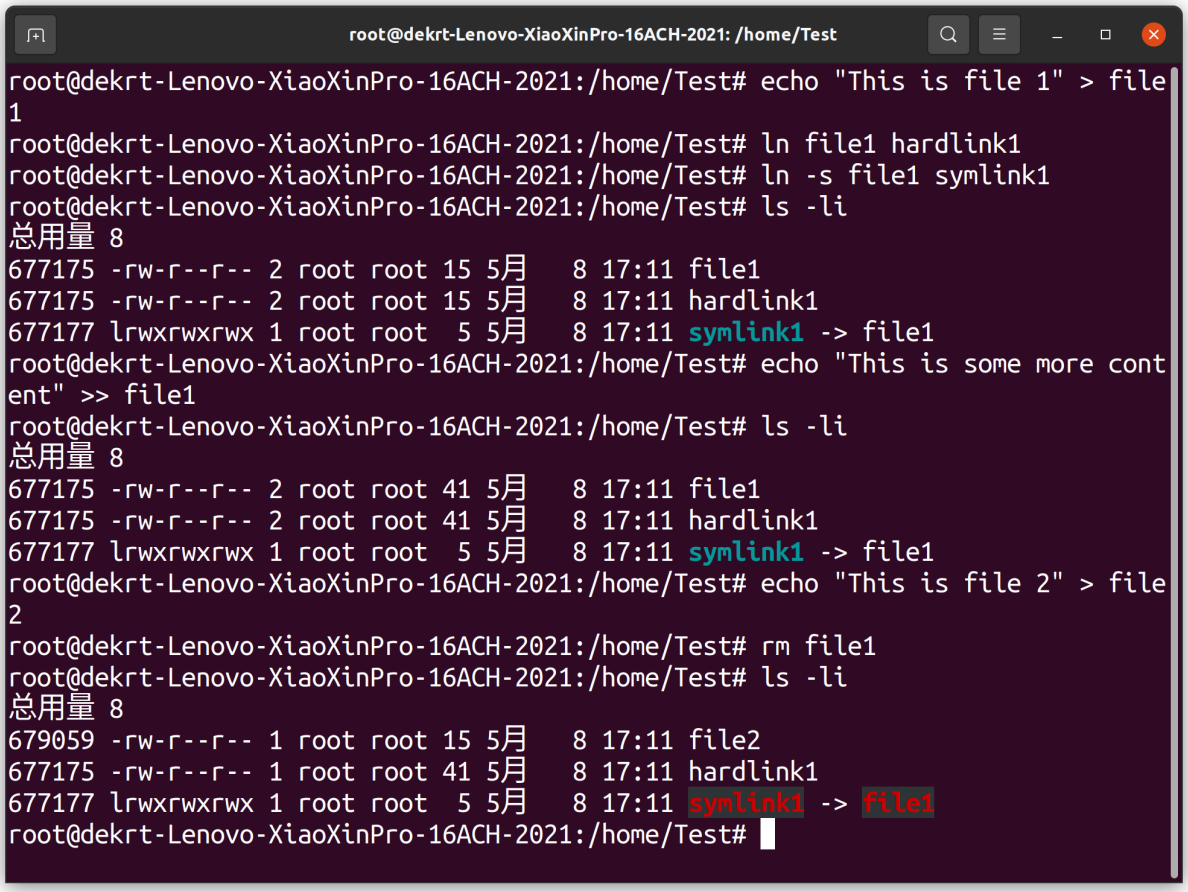
四、实验结果

4.1 编写Linux驱动及测试程序

运行测试程序，可以发现其成功运行



4.2 Linux下创建软连接与硬连接



五、实验错误排查和解决方法

5.1 编写Linux驱动及测试程序

错误1：注册设备后/dev 下没有对应的节点文件

字符设备注册函数不会自动创建设备节点文件，需要使用 mknod 命令手动创建，或者使用相关 API 函数（见三）。使用杂项设备的方式来注册可以避免这个复杂的过程。

错误2：测试程序无法打开设备文件/测试程序执行结果不正确

测试程序没有访问设备的权限
很多同学都会遇到这个问题。使用超级用户权限来执行测试程序：

```
$ sudo ./test
```

六、实验参考资料和网址

- 教学课件
- Google
- Stackoverflow
- CSDN