# 广州大学学生实验报告

**开课学院及实验室：**计算机科学与教育软件学院计算机软件实验室 2020**年**4**月**23**日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | 计算机科学与教育软件学院 | **年级/专业/班** | 软件171 | **姓名** | 谢金宏 | **学号** | 1706300001 |
| **实验课**  **程名称** | Linux系统实验 | | | | | **成绩** |  |
| **实验项**  **目名称** | 设备驱动 | | | | | **指导**  **老师** | 张艳玲 |

## 实验目的

1. 通过一个简单的设备驱动的实现过程。学会Linux中设备驱动程序的编写。
2. 深入理解内核驱动模块编写和编译过程。

## 实验内容

1. 编写一个字符设备驱动程序，并在设备的打开操作中打印主次设备号；
2. 编写一个用户测试程序，实现设备的读操作。

## 实验涉及的系统调用以及内核函数

1. **分配设备号函数register\_chrdev()，用于指定设备号的情况。函数原型为:**

*int register\_chrdev(unsigned int major, const char \*name,struct file\_operations \*fops);*

major：主设备号（已知）

name：设备名

fops：设备操作方法

1. **动态申请设备号函数alloc\_chrdev\_region，函数原型：**

*int alloc\_chrdev\_region(dev\_t \*from,unsigned int firstminor,unsigned int count,char \*name)*

from：调用该函数后自动分配得到的设备号

firstminor：第一个次设备号 (一般为0)

count：要分配的设备数

name：设备名

该函数会调用\_register\_chrdev\_region这个函数。

1. **从系统注销字符设备函数unregsiter\_chrdev，函数原型为：**

*int unregister\_chrdev(unsigned int major, const char \*name)*

major：主设备号

name：设备名

unregister\_chrdev\_region是unregsiter\_chrdev的升级版，2.6以上内核使用。

void unregister\_chrdev\_region(dev\_t dev, unsigned int count);

dev为设备号，和注册时候的要一致。

count为申请的次设备个数。

1. **分配设备空间函数cdev\_alloc，函数原型：**

*struct cdev \*cdev\_alloc(void)*

主要针对需要空间申请的操作

1. **对设备空间进行初始化和赋值操作函数cdev\_init，函数原型：**

*void cdev\_init(struct cdev \*cdev, const struct file\_operations \*fops)*

cdev：函数cdev\_alloc分配的设备空间结构体指针。

fops：对设备的操作。

1. **把设备添加进系统函数cdev\_add，使用cdev\_add注册字符设备前应该先调用register\_chrdev\_region或alloc\_chrdev\_region分配设备号。函数原型：**

*int cdev\_add(struct cdev\* dev,dev\_t num,unsigned int count)*

dev：字符设备结构体

num：设备号

count：添加的设备号的数量，具体的就是minor的数量。

1. **删除设备cdev\_del，函数原型：**

*void cdev\_del(struct cdev \*dev)*

dev：字符设备结构体

## 实验原理与重点

**字符设备：**是指只能一个字节一个字节读写的设备，不能随机读取设备内存中的某一数据，读取数据需要按照先后顺序。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。

每一个字符设备都在/dev目录下对应一个设备文件。Linux用户程序通过设备文件（或称设备节点）来使用驱动程序操作字符设备。

一个字符设备都有一个主设备号和一个次设备号。主设备号用来标识与设备文件相连的驱动程序，用来反映设备类型。次设备号被驱动程序用来辨别操作的是哪个设备，用来区分同类型的设备。

1. **描述字符设备的数据结构**

在Linux 2.6内核中的字符设备用cdev结构来描述，其定义如下：

*struct cdev*

*{*

*struct kobject kobj; //类似对象类，驱动模块的基础对象*

*struct module \*owner; //所属内核模块，一般为THIS\_MODULE*

*const struct file\_operations \*ops; //文件操作结构*

*struct list\_head list;*

*dev\_t dev; //设备号，int 类型，高12位为主设备号，低20位为次设备号*

*unsigned int count;*

*};*

1. **字符设备驱动模块的编写**

实现一个基本的字符驱动设备需要以下几个部分：字符设备驱动模块的加载、卸载函数和file\_operations结构中的成员函数。具体步骤如下：

* 1. **分配和释放设备号**

在设备驱动程序中，注册设备前首先要向系统申请设备号，分配设备号有静态和动态的两种方法：静态分配(register\_chrdev\_region()函数)或动态分配（alloc\_chrdev\_region()）。通过 unregister\_chrdev\_region()函数释放已分配的（无论是静态的还是动态的）设备号。

* 1. **定义并初始化一个struct file\_operations结构，并实现其中的操作函数**

*static struct file\_operations cdrv\_fops = {*

*.owner=THIS\_MODULE, /\* 这是一个宏，推向编译模块时自动创建的\_\_this\_module变量 \*/*

*.open=cdrv\_open,*

*.read=cdrv\_read,*

*.write=cdrv\_write,*

*};*

*static int cdrv\_open(struct inode \*inode, struct file \*filp)*

*static ssize\_t cdrv\_read(struct file \*filp, char \_\_user \*buf, size\_t size, loff\_t \*ppos)*

*static ssize\_t cdrv\_write(struct file \*filp, const char \_\_user \*buf, size\_t size, loff\_t \*ppos)*

* 1. **字符设备的注册（重点）**

首先定义该结构体：*struct cdev my\_cdev;*

然后即可初始化该结构体，使用如下的函数初始化：

*int cdev\_init(struct cdev \*dev, struct file\_operations \*fops)*

然后定义该结构体中的一个所有者字段：

*my\_cdev\_owner = THIS\_MODULE*

最后即可向该模块添加结构体：

*int cdev\_add(struct cdev \*dev, dev\_t dev\_num, unsigned int count)*

其中dev是cdev结构体，dev\_num是该设备对应的第一个设备编号，count则是与该设备关联的设备编号数量。

* 1. **删除字符设备**

*void cdev\_del(struct cdev \*dev);*

* 1. **注销设备号**

*unregister\_chrdev\_region(dev\_t first, unsigned int count);*

以上两个函数一般用于模块的出口函数。

* 1. **模块声明**

*MODULE\_LICENSE("GPL");*

* 1. **加载模块**

*module\_init(cdrv\_init);*

* 1. **卸载模块**

*module\_exit(cdrv\_exit);*

1. **编译模块Makefile文件**
2. **利用mknod命令在/dev目录下为字符设备生成对应的节点**

## 实验原始数据、结果与分析

下面给出一个实现了读操作和写操作的字符设备驱动的内核模块代码：

*// simple\_char\_dev\_driver.c*

#include <linux/kernel.h>

#include <linux/vmalloc.h>

#include <linux/module.h>

#include <linux/fs.h>

#include <linux/string.h>

#include <linux/uaccess.h>

#include <linux/init.h>

#include <linux/cdev.h>

#define DEVICE\_NAME "simple\_char\_dev"

#define BUFFER\_SIZE 4096

#define NEXT\_INDEX(idx) ((idx + 1) % BUFFER\_SIZE)

#define LOG(string) printk(DEVICE\_NAME ": [%s] " string, \_\_func\_\_)

#define LOGF(format, ...) printk(DEVICE\_NAME ": [%s] " format, \_\_func\_\_, \_\_VA\_ARGS\_\_)

static dev\_t dev;

static struct cdev \*simple\_cdev;

static signed count = 1;

static char \*buffer;

static int write\_index = 0;

static int read\_index = 0;

static int simple\_char\_dev\_open(struct inode \*inode, struct file \*file)

{

    int major = MAJOR(inode->i\_rdev);

    int minor = MINOR(inode->i\_rdev);

    LOGF("Device opened, major = %d, minor = %d\n", major, minor);

    return 0;

}

static ssize\_t simple\_char\_dev\_read(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*offset)

{

    char \*read\_buffer;

    int read\_count = 0;

    read\_buffer = vmalloc(PAGE\_SIZE);

    if (read\_buffer == NULL) {

        return 0;

    }

    while (read\_index != write\_index) {

        read\_buffer[read\_count++] = buffer[read\_index];

        read\_index = NEXT\_INDEX(read\_index);

    }

    copy\_to\_user(buf, read\_buffer, read\_count);

    vfree(read\_buffer);

    LOGF("%d bytes read.\n", read\_count);

    return read\_count;

}

static ssize\_t simple\_char\_dev\_write(struct file \*file, const char \_\_user \*buf, size\_t count, loff\_t \*offset)

{

    char \*write\_buffer;

    int write\_count = 0;

    write\_buffer = vmalloc(PAGE\_SIZE);

    if (write\_buffer == NULL) {

        return 0;

    }

    if (count > BUFFER\_SIZE) {

        count = BUFFER\_SIZE;

    }

    copy\_from\_user(write\_buffer, buf, count);

    do {

        buffer[write\_index] = write\_buffer[write\_count++];

        write\_index = NEXT\_INDEX(write\_index);

    } while (--count > 0 && write\_index != read\_index);

    if (count > 0) {

        LOGF("%ld bytes from user is not written due to full buffer.\n", count);

    }

    vfree(write\_buffer);

    LOGF("%d bytes written.\n", write\_count);

    return write\_count;

}

static const struct file\_operations simple\_char\_dev\_fops = {

    .owner = THIS\_MODULE,

    .open = simple\_char\_dev\_open,

    .read = simple\_char\_dev\_read,

    .write = simple\_char\_dev\_write,

};

static int \_\_init simple\_char\_dev\_init(void)

{

    int ret = 0;

    const char \*welcome\_message = "The simple char device is initialized.\n";

    buffer = vmalloc(BUFFER\_SIZE);

    if (buffer == NULL) {

        ret = -ENOMEM;

        goto out;

    }

    strcpy(buffer, welcome\_message);

    write\_index += strlen(welcome\_message) + 1;

    ret = alloc\_chrdev\_region(&dev, 0, count, DEVICE\_NAME);

    if (ret)

    {

        LOG("Fail to allocate char device region.\n");

        goto out;

    }

    simple\_cdev = cdev\_alloc();

    if (!simple\_cdev)

    {

        LOG("Fail to cedv\_alloc().\n");

        goto unregister\_chrdev;

    }

    cdev\_init(simple\_cdev, &simple\_char\_dev\_fops);

    ret = cdev\_add(simple\_cdev, dev, count);

    if (ret)

    {

        LOG("Fail to cdev\_add().\n");

        goto cdev\_fail;

    }

    LOGF("Sucessfully register char device: %s.\n", DEVICE\_NAME);

    LOGF("Major number = %d, Minor number = %d.\n", MAJOR(dev), MINOR(dev));

    goto out;

cdev\_fail:

    cdev\_del(simple\_cdev);

unregister\_chrdev:

    unregister\_chrdev\_region(dev, count);

out:

    return ret;

}

static void \_\_exit simple\_char\_dev\_exit(void)

{

    vfree(buffer);

    cdev\_del(simple\_cdev);

    unregister\_chrdev\_region(dev, count);

    LOG("Device removed.\n");

}

MODULE\_LICENSE("GPL");

module\_init(simple\_char\_dev\_init);

module\_exit(simple\_char\_dev\_exit);

在驱动的入口函数中，驱动使用vmalloc向系统申请一块缓冲区用于保存用户程序写入的数据，并向这块缓冲区中写入预先定义的欢迎消息。这块缓冲区在read操作和write操作中被维护为环形缓冲区。驱动还使用alloc\_chrdev\_region注册一个字符设备编号，使用cdev\_alloc和cdev\_init初始化字符设备结构体，并使用cdev\_add将字符设备结构体与编号关联，并添加到系统中。在模块的出口函数中，驱动归还在入口函数中申请的系统资源。

在驱动的read操作中，程序从环形缓冲区读出数据，并写入用户空间的缓冲区；在驱动的write操作中，程序从用户空间缓存区中拷贝需要写入的数据，并将数据写入环形缓存中。程序维护read\_index和write\_index，每读出（写入）一个字节，read\_index（write\_index）在对缓冲区大小的取模的意义下向前递增1。当index\_index与write\_index相等时，视为缓冲区空（所有已经写入的数据已被读出）。

编写Makefile如下：

obj-m := simple\_char\_dev\_driver.o

KERNELDIR ?= /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

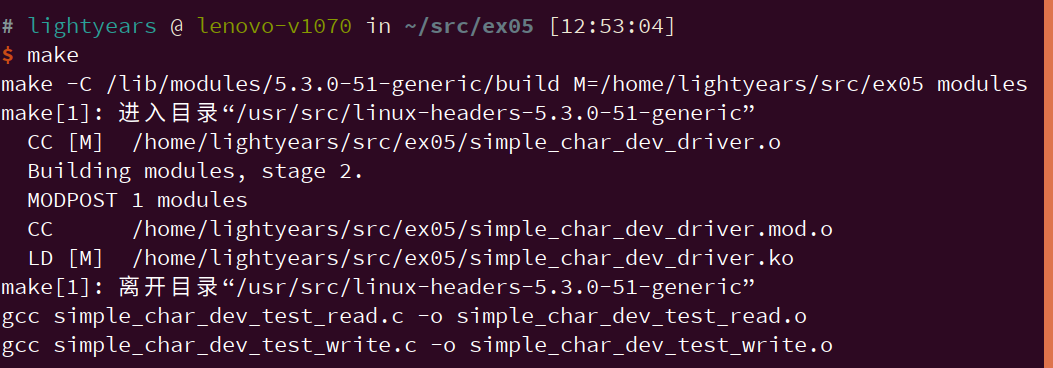
default:

    make -C $(KERNELDIR) M=$(PWD) modules

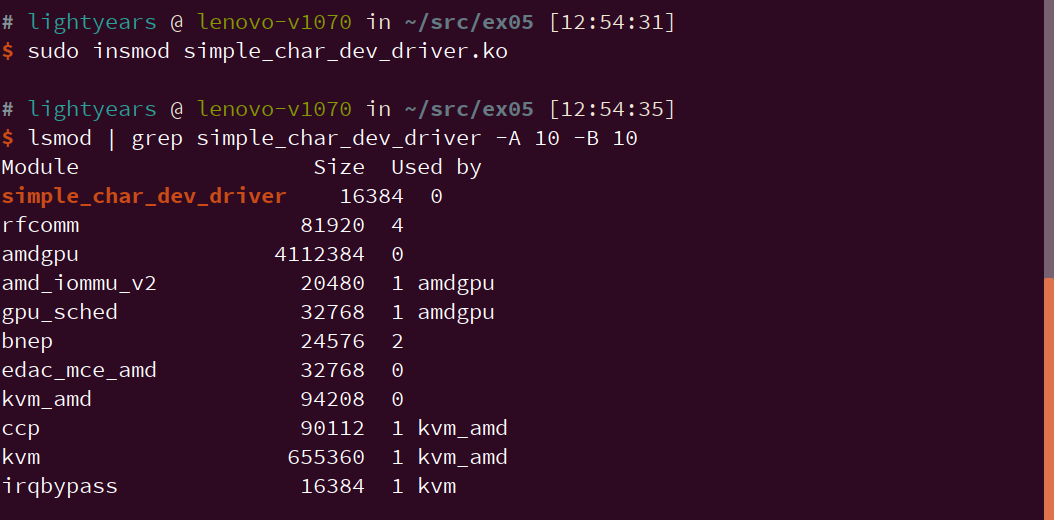
clean:

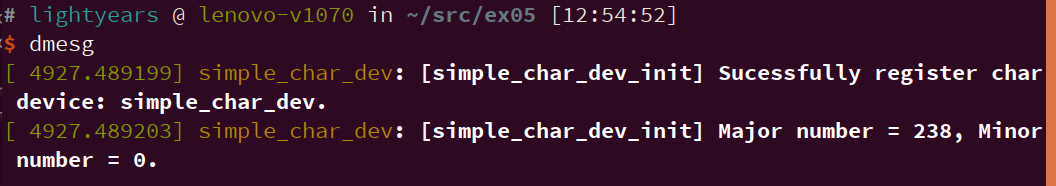
    make -C $(KERNELDIR) M=$(PWD) clean

使用make编译字符设备驱动内核模块。

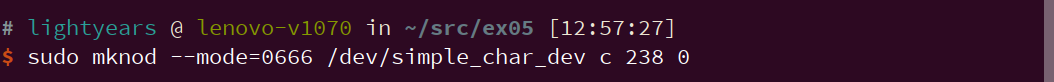


插入内核模块并查看内核日志。

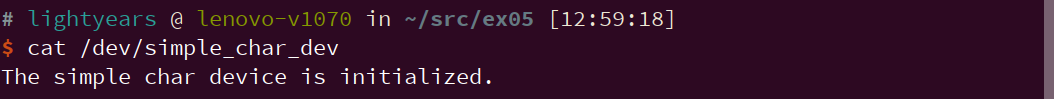


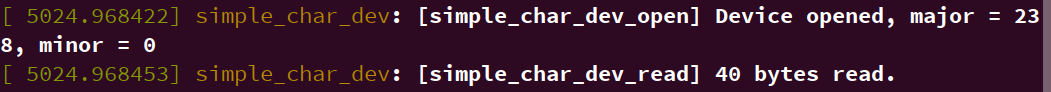


使用mknod创建设备文件节点。

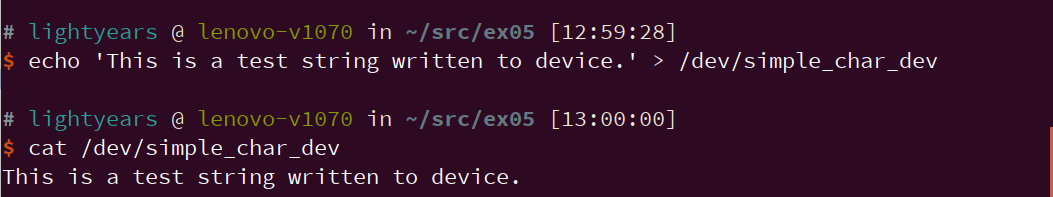


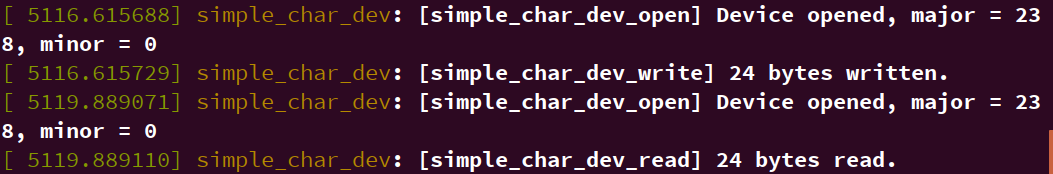
测试读操作。





测试写操作。





下面给出用编程方式测试写操作的代码：

*// simple\_char\_dev\_test\_write.c*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <fcntl.h>

#define BUFFER\_SIZE 128

#define DEV\_NAME "/dev/simple\_char\_dev"

int main()

{

    char buffer[BUFFER\_SIZE];

    int fd;

    fd = open(DEV\_NAME, O\_WRONLY);

    if (fd < 0)

    {

        printf("Open device %s failed\n", DEV\_NAME);

        return EXIT\_FAILURE;

    }

    const char \*message = "Twinkle, twinkle, little bat!\n"

                          "How I wonder what you’re at!\n"

                          "Up above the world you fly,\n"

                          "Like a tea-tray in the sky.\n";

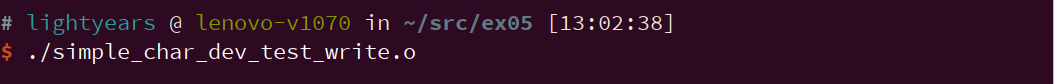
    write(fd, message, strlen(message) + 1);

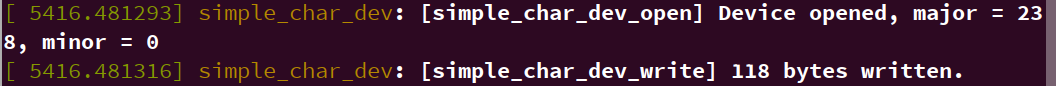
    close(fd);

    return 0;

}

以编程方式测试写操作。





以编程方式测试读操作。

*// simple\_char\_dev\_test\_read.c*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#define BUFFER\_SIZE 128

#define DEV\_NAME "/dev/simple\_char\_dev"

int main(void)

{

    char buffer[BUFFER\_SIZE];

    int fd;

    fd = open(DEV\_NAME, O\_RDONLY);

    if (fd < 0)

    {

        printf("Open device %s failed\n", DEV\_NAME);

        return EXIT\_FAILURE;

    }

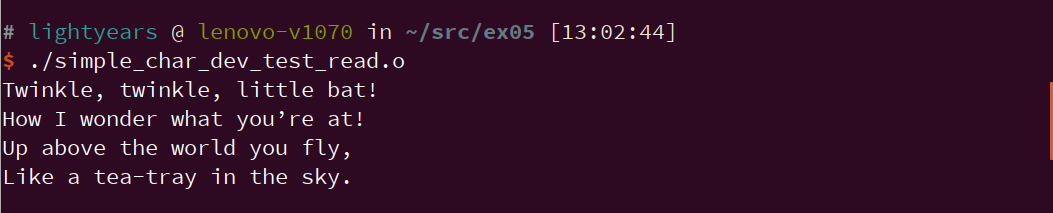
    read(fd, buffer, 64);

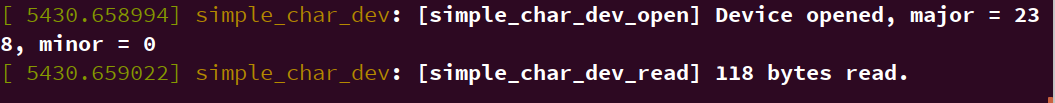
    printf("%s", buffer);

    close(fd);

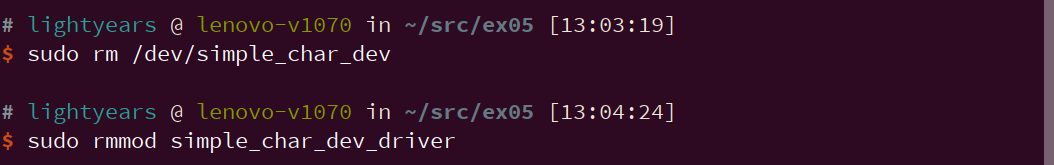
    return 0;

}





移除设备文件节点，并卸载模块。

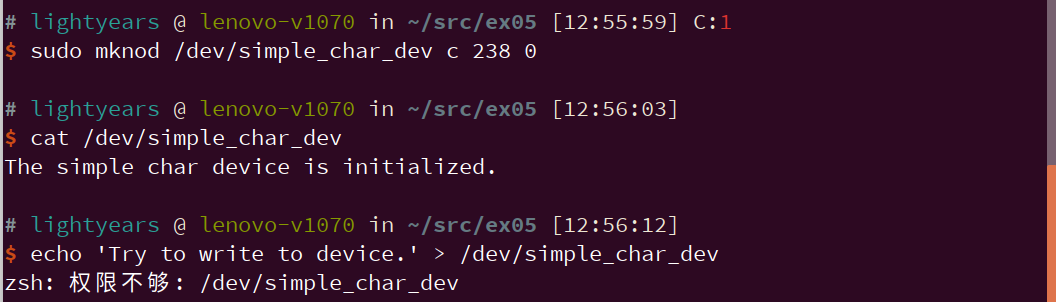




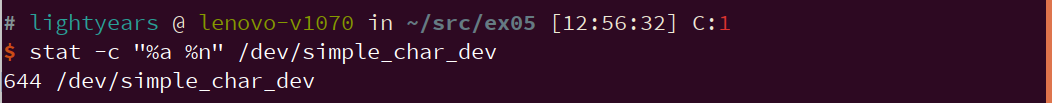
## 实验思考题

1. 对自己编写的字符设备驱动模块进行测试，对实验过程中出现的错误进行截图，分析错误原因，并写出修改代码和修改理由。

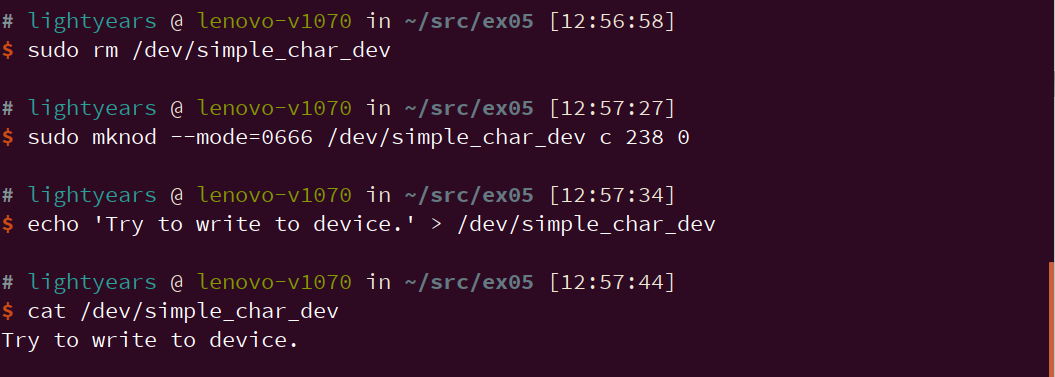
在写设备文件时，shell提示权限不足。



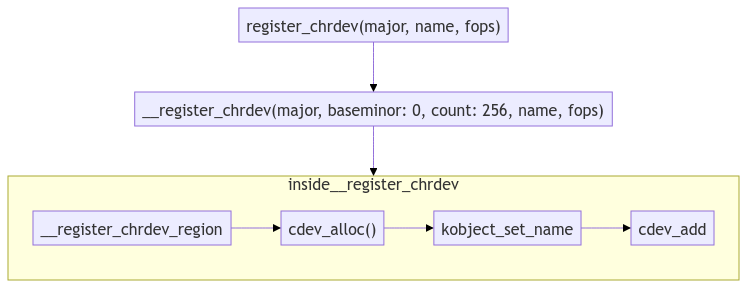
通过stat可查看设备文件的权限，发现以默认参数的mknod创建的设备文件，普通用户只有设备文件的读权限而没有写权限。



如果希望普通用户也具有写设备的权限，需要在mknod使用mode参数指定文件权限。随后可验证普通用户也能对设备进行写操作。



1. 根据实验过程和阅读源代码，画出实现register\_chrdev()函数的流程图



Register\_chrdev是定义在include/fs.h的内联函数，它调用一个参数更多的\_\_register\_chrdev函数完成实际的工作。在\_\_register\_chrdev函数中，首先是\_\_register\_chrdev\_region被调用从而在系统中注册一组设备编号，并得到一个char\_device\_struct。然后是cdev\_alloc函数被调用以得到一个cdev。接下来根据参数设置cdev的各个字段，在设置各个字段的过程中会调用kobject\_set\_name以设置cdev中的kobject字段。最后调用cdev\_add，将字符设备真正地加入系统中。