

基础题目实验报告

1.Shell编程

第一行表示是一个shell文件，当bash解释器解释的时候读到这句就知道是shell文件。

\$1是指在命令行输入的的第一个参数（这里shell文件的名称算是第0个参数）

\$2同理就是第二个参数

根据需求我们得到要读写的文件名称filename和要进行的操作limit

解析shell中的变量的值就要用\${variable}

本题关键是如何去读文件。

我这里使用了**管道法**（|这就是管道，管道左边的输出会沿着管道流入管道右边作为输入），cat filename将输入的文件展开之后读每一行，

然后读到的就将这一行echo也就是输出到控制台上。

否则的话就是写操作，将事先准备好的学号写入文件中

这里的 > 是重定向符号，表示将左边的内容输出到右边中去（这个会覆盖数据，>>表示追加）

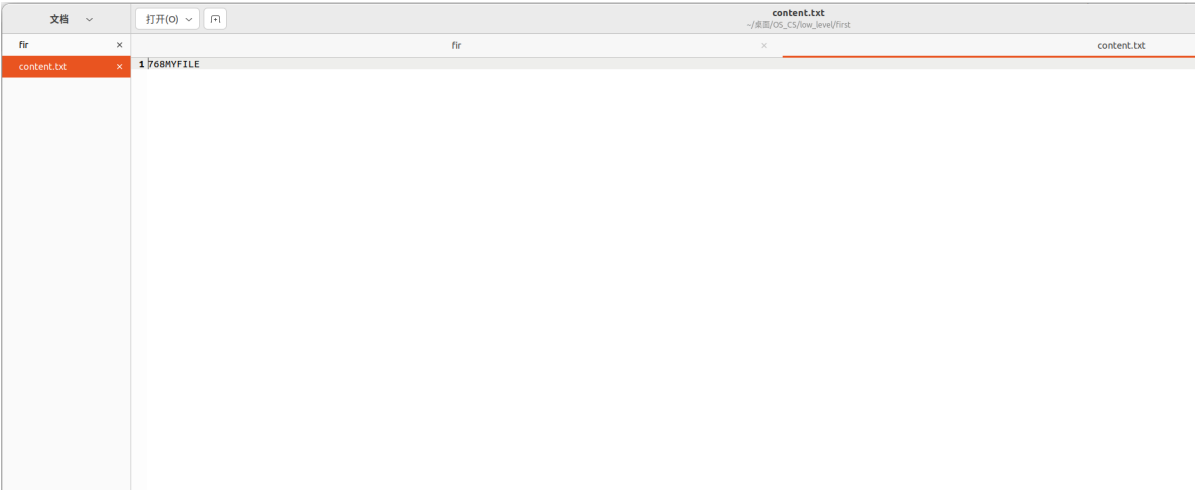
```
1 #!/bin/bash
2 # @author papa
3 filename=$1
4 limit=$2
5 content="768MYFILE"
6 if [ $limit == "read" ]
7 then
8     cat ${filename} | while read line
9 do
10     echo ${line}
11 done
12 else
13     echo "${content}">${filename}
14 fi
15
```

执行一下脚本看看结果

先执行写操作，再去读发现写入成功，读操作也成功了。

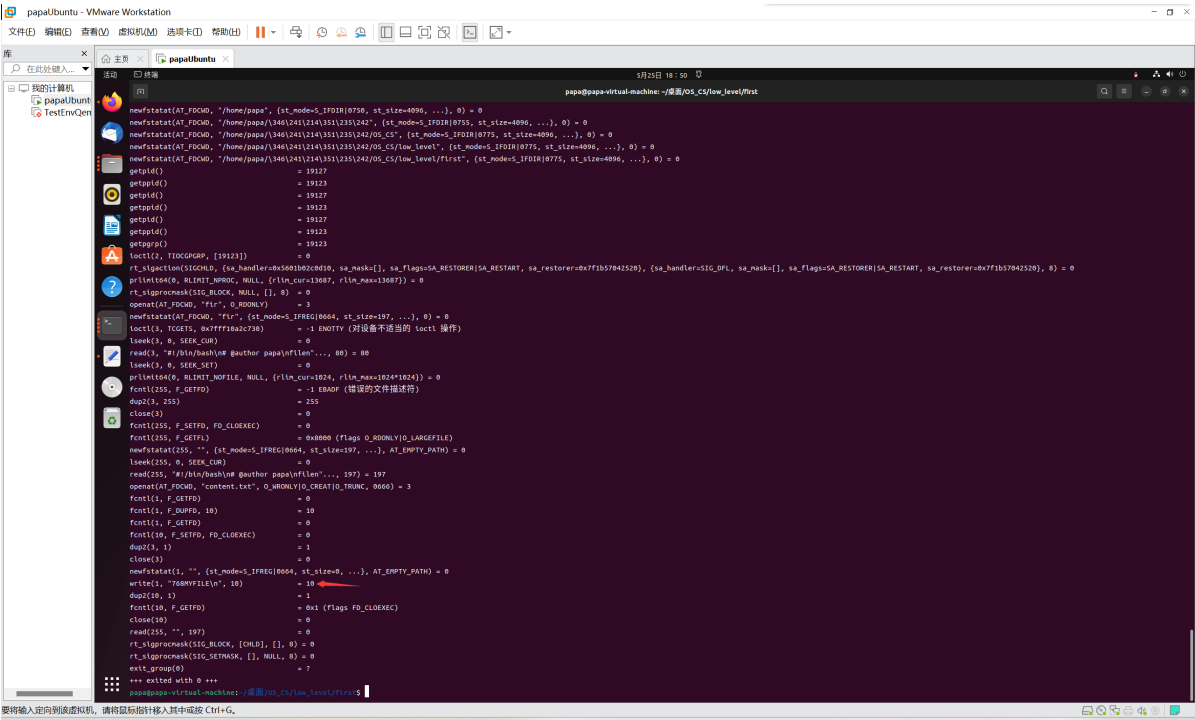
```
papa@papa-virtual-machine:~/桌面/OS_CS/low_level/first$ /bin/bash fir content.txt write
papa@papa-virtual-machine:~/桌面/OS_CS/low_level/first$ /bin/bash fir content.txt read
768MYFILE
```

打开content.txt也发现已经写入



下面我们验证shell脚本在重定向输出时用到了write系统调用，我们使用strace命令得到

可以看到我们向1，1代表的文件描述符是stdout标准输出，但是我们前面肯定是使用了dup2进行了重定向，此时的文件描述符1指向了我们要输出到的文件。



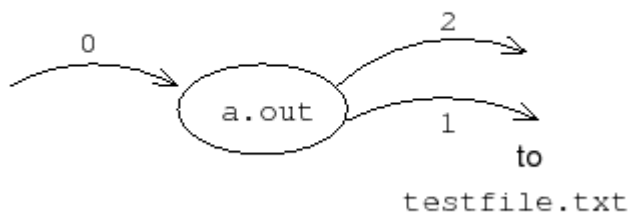
before redirection



file descriptor table

0	standard input
1	standard output
2	standard error

after redirection



file descriptor table

0	standard input
1	write to testfile.txt
2	standard error

2.系统调用编程

(1)

第一题就是熟悉几个系统调用的使用，open，write，read

open第一个参数是要打开的文件，我这里是相对路径，就是和这个c程序处于同一级下的文件

第二个参数是权限 O_CREAT表示如果文件不存在就会创建，O_RDWR表示文件支持读写

还有一种是有第三个参数，也是表示权限的这里一般是表示了属主、同组和其他人对文件的文件操作权限。

read和write的参数很像

第一个参数都是文件名或者设备名（linux中设备也是文件）

第二个参数buff，对于read来说是用户程序定义的缓冲区，read会把文件中的内容读到这个用户的buff中来

对于write来说是，用户空间提供一个buff字符数组，会把内容写到文件中去。

第三个参数是最大的字节数，写入或者读得的，超过会截断

返回值都是实际写入或者读出的字节数，如果为-1，说明读写失败。

```
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>
#include<string.h>
#include<stdio.h>
int main(int argc,char* argv[]){
    char* filepath=argv[1];//获取命令行输入的第二个参数为文件名
    char* operate=argv[2];//第二个参数为执行的操作 write/read
    int fd=open(filepath,O_CREAT|O_RDWR);//这里打开输入的文件，如果没有就会自动创建，这里
    会获得描述该文件的描述符fd
    char buff[50];
    if(strcmp("write",operate)==0){
        printf("Input what you want to input:\n");
        read(0,buff,30);//0文件描述符对应的是stdin标准输入，这里就是从键盘获得输入
        size_t res=write(fd,buff,30);//将获得的输入写到fd关联的文件中去，返回值是实际写入
        的字节数
        if(res<0){
            printf("fail to write!");
        }
    }
    else if(strcmp("read",operate)==0){
        size_t res=read(fd,buff,30);//这里就是从fd关联的文件中读最多30个字节到buff中去
        if(res<0){
```

```

        printf("fail to read");
    }
    printf("the content of the file is ");
    printf(" %s",buff);

}

}

```

首先编译second.c文件获得可执行文件second

然后执行second文件使用./

发现分别试验write和read功能都成功了！

```

root@papa-virtual-machine:/home/papa/桌面/OS_CS/low_level/second_1# gcc second.c -o second
root@papa-virtual-machine:/home/papa/桌面/OS_CS/low_level/second_1# ./second content.txt write
Input what you want to input:
768MYFILE
root@papa-virtual-machine:/home/papa/桌面/OS_CS/low_level/second_1# ./second content.txt read
the content of the file is 768MYFILE

```

这里回答一下fopen, fwrite, fread与open, write, read系统调用的区别

fopen返回的是文件结构体指针, open返回的是文件句柄（就是对应的文件索引表中的索引值）

并且fread, fwrite是**带有缓冲区的**读写, 只有当缓冲区满了, 在向磁盘或者内存上读写, 可以减少read, write来回进行内核态和用户态的切换

(2)

要求多进程互斥的写同一个文件, 这里使用**信号量**机制。

```

void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t
offset);

```

这里主要使用mmap来进行共享内存,

第一个参数为NULL表示由系统自动去分配一块进程的虚拟空间来进行共享,

第二个参数是分配的空间大小size这里设置就正好是一个信号量结构体 (sem_t) 的大小

第三个参数表示对这块空间的权限, 这里PROT_WRITE和PROT_READ表示可以读写

第四个参数也是权限, MAP_SHARED表示共享, MAP_ANON表示匿名文件, 由于它的第五个参数没有指定某个文件句柄（填充的-1）所以是匿名文件

第六个参数是映射文件的offset偏移量。

sem_init()函数第一个参数是信号量结构的指针, 第二个参数是一个标志（当为1时表示进程间都可以共享, 0表示进程的线程间共享）, 第三个参数value就是初始化的值, 这里就是对p信号量设置初始值为1。

sem_wait()就是将信号量的值-1, 然后判断如果<0, 这个进程就会block阻塞。

sem_post()就是将信号量值+1, 然后判断如果>=0, 说明有进程在等待就会唤醒它。

下面是代码：

```
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>
#include<string.h>
#include<stdio.h>
#include<semaphore.h>
#include<pthread.h>
#include<sys/mman.h>
int main(int argc,char* argv[]){
    char* filepath=argv[1];
    char* operate=argv[2];
    int fd=open(filepath,O_CREAT|O_RDWR,0666);
    char buff[50];
    if(strcmp("write",operate)==0){
        pid_t id=fork();
        sem_t* p =
(sem_t*)mmap(NULL,sizeof(sem_t),PROT_WRITE|PROT_READ,MAP_SHARED|MAP_ANON,-1,0);
        sem_init(p,1,1);
        if(id>0){
            //father
            sem_wait(p);
            char buff[]="768PROC1 MYFILE1\n";
            pid_t id=getpid();
            printf("[father] id=%d\n",id);
            write(fd,buff,strlen(buff));
            sem_post(p);
        }else{
            //son
            sem_wait(p);
            char buff[]="768PROC2 MYFILE2\n";
            pid_t id=getpid();
            printf("[son] id=%d\n",id);
            write(fd,buff,strlen(buff));
            sem_post(p);
        }
        sem_destroy(p);//消除信号量p
        munmap(p,sizeof(sem_t));//解除共享区的映射，释放这块内存
    }else if(strcmp("read",operate)==0){
        size_t res=read(fd,buff,100);
        printf("the numbers of read=%d ",res);
        if(res<0){
            printf("fail to read");
        }
        printf("\n%s",buff);
    }
    close(fd);
}
```

```
}
```

```
gcc second_improved.c -o second2
```

先编译链接生成可执行文件second2

然后运行并输入参数

先写入文件

```
./second2 content.txt write
```

输出结果，这个用于展示父子进程的pid以及先后顺序

```
[father] id=4101  
[son] id=4102
```

咱们再读一下文件，看是否成功写入 `./second2 content.txt read`

```
the numbers of read=34
```

结果

```
768PROC1 MYFILE1  
768PROC2 MYFILE2
```

没问题，第一个是写入的字节数我读到了，然后写入的内容也是符合上面先输出father，再输出son的。

3.内核编程

我们这里是使用添加模块的方式来添加系统调用

首先要查找系统调用表的位置 `sys_call_table`

输入指令

```
cat /proc/kallsyms | grep sys_call
```

找到了表的位置（这里注意每次开机这个位置是可能会变化的，所以每次都要重新更改宏中的地址，当然别忘了加上前缀0x）

```
ffffffff86e6f3e0 t proc_sys_call_handler  
ffffffff87e00320 D sys_call_table  
ffffffff87e013a0 D ia32_sys_call_table
```

然后查找一下空闲的系统调用号以供咱们使用

输入命令

```
cat /usr/include/asm/unistd_64.h
```

然后发现这中间都是空的，我们就可以使用了。

```

#define __NR_clock_adjtime 305
#define __NR_syncfs 306
#define __NR_sendmsg 307
#define __NR_setns 308
#define __NR_getcpu 309
#define __NR_process_vm_readv 310
#define __NR_process_vm_writev 311
#define __NR_kcmp 312
#define __NR_finit_module 313
#define __NR_sched_getattr 314
#define __NR_sched_getattr 315
#define __NR_renameat2 316
#define __NR_seccomp 317
#define __NR_getrandom 318
#define __NR_mknod_create 319
#define __NR_kexec_file_load 320
#define __NR_bpf 321
#define __NR_execveat 322
#define __NR_userfaultfd 323
#define __NR_membarrier 324
#define __NR_mlock2 325
#define __NR_copy_file_range 326
#define __NR_preadv2 327
#define __NR_pwritev2 328
#define __NR_pkey_mprotect 329
#define __NR_pkey_alloc 330
#define __NR_pkey_free 331
#define __NR_statx 332
#define __NR_io_pgetevents 333
#define __NR_rseq 334
#define __NR_pidfd_send_signal 424
#define __NR_to_uring_setup 425
#define __NR_to_uring_enter 426
#define __NR_to_uring_register 427
#define __NR_open_tree 428
#define __NR_move_mount 429
#define __NR_fsopen 430
#define __NR_fsconfig 431
#define __NR_fsmount 432
#define __NR_fspick 433
#define __NR_pidfd_open 434
#define __NR_clone3 435
#define __NR_close_range 436
#define __NR_openat2 437
#define __NR_pidfd_getfd 438
#define __NR_faccessat2 439
#define __NR_process_madvise 440
#define __NR_epoll_pwait2 441

```

下面看我们的程序：

咱们是通过修改Cr0这个控制寄存器的第十七位，这位表示能否进行写入

所以clear_cr0和setback_cr0这两函数一个是给将它第十七位置0，另一个是重置回1。

当向系统调用传参数的时候，不能向传统的传入几个参数，要以wrapper封装的方式传递，所以这里使用

寄存器集合struct pt_regs

它的di，si分别表示传递的第一个和第二个参数

```

#include<linux/kernel.h>
#include<linux/init.h>
#include<linux/unistd.h>
#include<linux/module.h>
#include<linux/sched.h>

MODULE_LICENSE("Dual BSD/GPL");
#define SYS_CALL_TABLE_ADDRESS 0xffffffff87e00320//这里是每次都需要根据系统调用表的位置更改的
#define NUM 335//这是我使用的系统调用号
unsigned long *my_sys_call_table;//指向系统调用表的指针

static int (*funcPtr)(void);//保存我们要添加的系统调用号那块地址的原有数据的

int old_cr0;//保存原来cr0寄存器中的值

static int clear_cr0(void){
    unsigned int cr0;
    unsigned int ret;
    asm volatile("movq %%cr0,%%rax":"=a"(cr0));//这里的意思是把cr0的原始值通过ax寄存器
    保存到cr0变量中
    ret=cr0;//保存旧值要返回的

```

```

cr0&=0xffffffffffffffffff; //将第十七位置0
asm volatile("movq %%rax,%%cr0::\"a\"(cr0)); //将cr0的新值通过ax寄存器传输到cr0寄存器上
return ret; //返回cr0的旧值，最后是要重置的
}
static void setback_cr0(int v){
    asm volatile("movq %%rax,%%cr0::\"a\"(v)); //将旧值重置回cr0寄存器中
}
asm linkage int my_sys_call(struct pt_regs* regs){
    printk("the syscall of the module....\n");
    int sid;
    sid=regs->di; //接受我们传入的学号后三位
    int flag;
    flag=regs->si; //接受我们传入的标志位，为1得到学号的十位，为0得到学号的个位
    int res=0;
    printk("the argument sid= %d,flag=%d\n",sid,flag);
    if(flag){
        res=sid/10%10;
    }else{
        res=sid%10;
    }
    printk("result=%d\n",res);
    return res;
}

static int __init call_init(void){
    printk("call init....\n");
    my_sys_call_table=(unsigned long*)SYS_CALL_TABLE_ADDRESS;
    funcPtr=( int(*) (void))my_sys_call_table[NUM]; //保存原有的这里的数据
    old_cr0=clear_cr0(); //修改cr0寄存器，使之可以写入
    my_sys_call_table[NUM]=(unsigned long)&my_sys_call; //填入我们的系统调用函数
    setback_cr0(old_cr0); //再重置
    return 0;
}
static void __exit call_exit(void){
    printk("call exit....\n");
    old_cr0=clear_cr0();
    my_sys_call_table[NUM]=(unsigned long)funcPtr; //恢复原状
    setback_cr0(old_cr0);
}

module_init(call_init); //在模块加载时会运行
module_exit(call_exit); //在模块卸载时运行

```

Makefile文件


```

obj-m:=my_system.o
CURRENT_PATH:=$(shell pwd)
CONFIG_MODULE_SIG=n
LINUX_KERNEL_PATH:=/home/papa/桌面/Share/linux-5.19.10
all:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) modules
clean:
    make -C $(LINUX_KERNEL_PATH) M=$(CURRENT_PATH) clean

```


首先make生成我们的模块（以ko为后缀的文件）

```
make
```

 my_system.ko

然后装载模块并查看日志

```
sudo insmod my_system.ko[sudo] papa
sudo dmesg -c
```

发现call_init函数已经成功调用，模块装载成功

```
[ 4662.728702] call init....
```

接下来对我们的装载的模块进行测试

这是测试文件

```
#include<stdio.h>
#include<stdlib.h>
#include<linux/kernel.h>
#include<sys/syscall.h>
#include<unistd.h>

int main(int argc,char *argv[]){
    int flag=1;
    int x=syscall(335,768,flag);
    printf("flag : %d,the syscall result is %d\n",flag,x);
    x=syscall(335,768,(flag=0));
    printf("flag : %d,the syscall result is %d\n",flag,x);

    return 0;
}
```

我们继续编译链接再执行

```
gcc test.c -o test
./test
```

结果正确!

```
flag : 1,the syscall result is 6
flag : 0,the syscall result is 8
```

我们再查看一下日志也没问题！

```
the syscall of the module....
the argument sid= 768,flag=1
result=6
the syscall of the module....
the argument sid= 768,flag=0
result=8
```

最后记得卸载模块rmmod

4.驱动编程

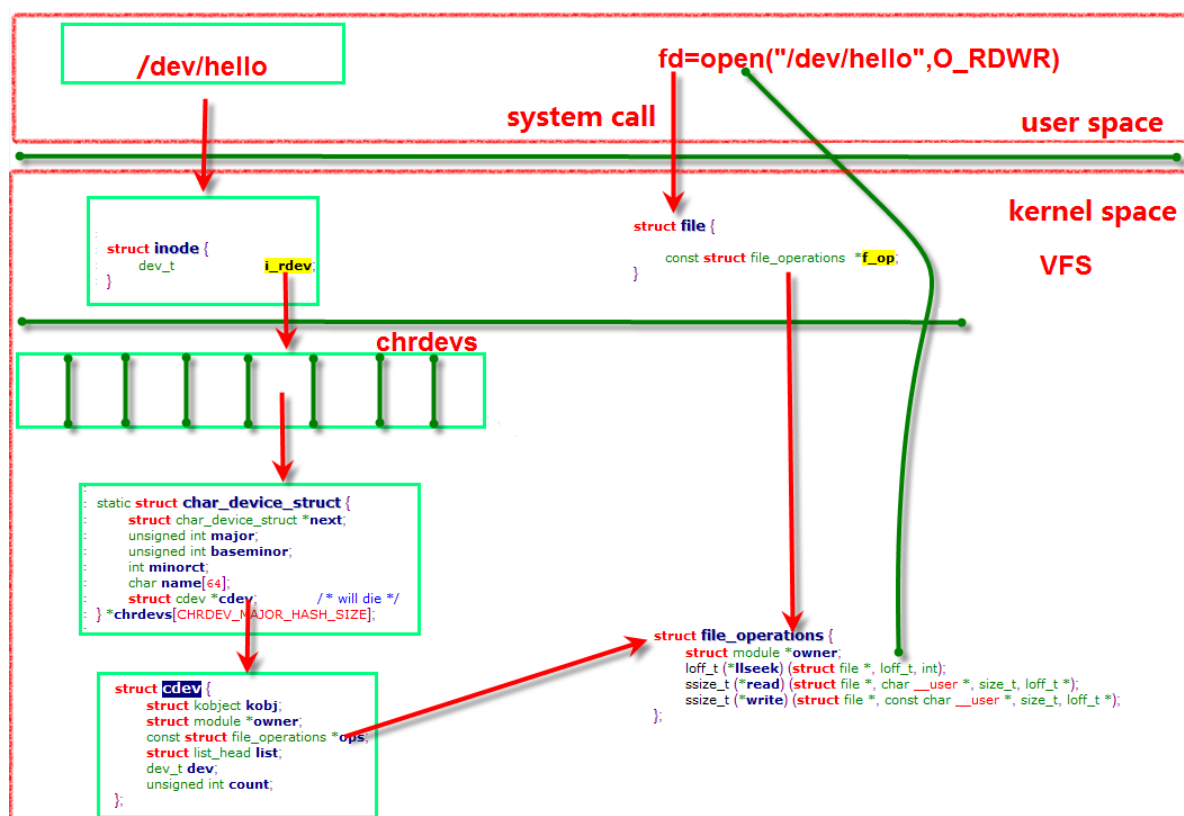
linux中设备也是一种文件，都在/dev目录下。

对文件的操作对设备都可以，只不过需要通过file_operation结构体去进行绑定

就比如当对设备文件open时就会去调用咱们自己自定义的cdev_open函数

所以驱动编程其实就是咱们去重新实现这些系统调用

linux中，struct cdev表示一个字符设备，绑定文件操作函数其实也是通过cdev去进行绑定



这里在注意我们提供的设备号，也要保证是空闲的未被使用的，

可以使用命令 `cat /proc/devices`

查看哪些设备号是空闲可用的，我这里使用的是60

这里注意一下ioctl函数中的几个不同的cmd命令，这是我在另一个头文件中所定义的

TEST_MAGIC是幻数，表示是同一类的命令，然后TEST_MAX_NR定义了最多的命令数目

正好下面就是从1到3，三个命令

```
#ifndef TEST_CMD_H
#define TEST_CMD_H

#define TEST_MAGIC 'x'
#define TEST_MAX_NR 3

#define RW_CLEAR _IO(TEST_MAGIC,1)
#define READ_OLD _IO(TEST_MAGIC,2)
#define READ_NEW _IO(TEST_MAGIC,3)

#endif
```

```
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/ctype.h>
#include<linux/device.h>
#include<linux/cdev.h>
#include<linux/fs.h>
#include<linux/init.h>
#include<linux/uaccess.h>
#include"test_cmd.h"
#define BUFF_SIZE 1024
#define id 768
#define ERROR 1
#define MY_DEVICE_NAME "papa_cdev"
//这里就是内核中的字符缓冲区rwbuff，初始值为学号后三位
static char rwbuff[BUFF_SIZE+1]="768";//注意最大长度为什么是1024+1
//因为虽然最大存储长度是1024，但是要考虑到字符串的结束符"\0"
static int rwbufflen=4;//这里也是考虑到了结束符所以是3+1=4

static char ori_rwbuff[BUFF_SIZE+1];//这个是来存储旧数据的，如果要对当前的缓冲区进行write
操作时，原数据就会被保存到ori_rwbuff中
static int oribufflen=0;
static int inuse=0;//用于实现互斥访问该设备的
//对应open系统调用
static int cdev_open(struct inode* i,struct file* f){
    printk("cdev_open is called!\n");
    printk("rwbuff : %s\n",rwbuff);
    printk("ori_rwbuff : %s\n",ori_rwbuff);
    //当没人使用时，我就可以打开这个设备
    if(inuse==0){
        inuse=1;
        try_module_get(THIS_MODULE);//该模块计数+1
        return 0;
    }
}
```

```

    }
    return -1;
}
//对应close系统调用
static int cdev_release(struct inode*i,struct file* f){
    printk("[papa] device is released!\n");
    //释放该设备时，我这里是重回原样，关键是要注意inuse要变为0，要不然以后就不能打开了，用完要释放的
    memset(ori_rwbuff,0,BUFF_SIZE);
    oribufflen=0;
    strcpy(rwbuff,"768");
    rwbufflen=4;
    inuse=0;
    module_put(THIS_MODULE);
    return 0;
}
//对应write系统调用
static ssize_t cdev_write(struct file*f,const char* u_buff,size_t size,loff_t *l)
{
    ssize_t ret=0;
    printk("[papa]cdev_write is called!\n");
    if(size>BUFF_SIZE){//如果写入长度超过了1024，会截断的最多写1024个
        size=BUFF_SIZE;
        //u_buff[size]='\0';
    }
    //printk("before copy ori_rwbuff is %s\n",ori_rwbuff);
    //printk("before copy rwbuff is %s\n",rwbuff);
    strcpy(ori_rwbuff,rwbuff);//执行写入前，保存最近一次的缓冲区数据，拷贝一下到ori_rwbuff里
    printk("after copy ori_rwbuff is %s\n",ori_rwbuff);
    printk("after copy rwbuff is %s\n",rwbuff);
    oribufflen=rwbufflen;
    // printk("The content before the last modification is %s\n",ori_rwbuff);
    //下面这个函数就实现了从用户空间拷贝到内核空间，将用户空间提供的数据拷贝到我们的字符设备缓冲区中
    if(copy_from_user(rwbuff,u_buff,size)){
        printk(KERN_ERR"fail to copy from user_buff to rwbuff!\n");
        ret=0;
    }
    else{
        //这里特别处理，因为如果长度>=1024的话，结束标识符\0也会被截断，所以我们要给他在最后加上，否则输出字符串会乱码
        if(size==BUFF_SIZE){ rwbuff[size]='\0';}
        printk("write data is %s\n",rwbuff);
        printk("write byte is %d\n",strlen(rwbuff));
        ret=size;
        rwbufflen=size;
    }
    return ret;
}
//对应read系统调用
static ssize_t cdev_read(struct file*f,char* u_buff,size_t size,loff_t* l){
    int ret=0;
    printk("[papa]cdev_read is called!\n");

```

```

if(size>BUFF_SIZE) size=BUFF_SIZE;
//从内核空间拷贝到用户空间，将内核缓冲区中的数据写到用户空间提供的地址中去
if(copy_to_user(u_buff,rwbuff,size+1)){
    printk(KERN_ERR"error:fail to copy to user_buff from rwbuff");
    ret=0;
}else{
    printk("[cdev_read] rwbuff is %s\n",rwbuff);
    ret=size;
}

return ret;
}

//用于功能扩展的系统调用对应ioctl
static long cdev_ioctl(struct file* f,unsigned int cmd,unsigned long args){
    printk("[papa]cdev_ioctl is calling!\n");
    int ret;
    switch(cmd){
        case RW_CLEAR:{
            //对应清空操作
            //这里将字符设备的缓冲区清空（使用memset）
            memset(rwbuff,0,rwbufflen);
            rwbufflen=0;
            printk("successfully clear! now buff_len is %d\n",rwbufflen);
            ret=1;

        };break;
        case READ_OLD:{
            //读最近一次写入前的数据
            //将ori_rwbuff中的数据拷贝到用户空间
            if(copy_to_user((char*)args,ori_rwbuff,oribufflen)){
                printk("[READ_OLD] copy_to_user fail\n");
                ret = EFAULT;
            }else{
                printk("[READ_OLD] copy_to_user successfully!\n");
            }
        };break;
        case READ_NEW:{
            //这里就是read操作一样了，读最新的数据
            if(copy_to_user((char*)args,rwbuff,rwbufflen)){
                printk("[READ_NEW] copy_to_user fail\n");
                ret = EFAULT;
            }else{
                printk("[READ_NEW] copy_to_user successfully!\n");
            }

        };break;
        default:printk("error cmd!\n");
            ret = - EINVAL;
            break;
    }
}

```

```

    return ret;
}
//这里进行系统调用与设备文件操作的具体绑定
static struct file_operations cdev_fops={
    .open = cdev_open,
    .release = cdev_release,
    .read = cdev_read,
    .write = cdev_write,
    .unlocked_ioctl = cdev_ioctl,
    .owner = THIS_MODULE
};

//这里进行设备号的注册和文件操作的绑定
static int __init chrdev_init(void){
    printk("cdev_init is called!\n");
    printk("hello,my id is %d\n",id);
    int ret=1;
    ret=register_chrdev(60,MY_DEVICE_NAME,&cdev_fops);//这个函数一举完成了设备号的注册
    和操作的绑定，把我们前面定义的cdev_fops文件操作结构体与绑定到了设备上
    if(ret != -1){
        printk("[papa] device successfully initialized.\n");
    }else{
        printk("[papa] device failed when initializing.\n");
    }
    return ret;
}

static void __exit chrdev_exit(void){
    printk("dev_exit is called!\n");
    unregister_chrdev(60,MY_DEVICE_NAME);//注销该设备号
    printk("[papa] device successfully removed.\n");
}

MODULE_LICENSE("GPL");
module_init(chrdev_init);
module_exit(chrdev_exit);

```

此时咱们的设备驱动模块就已经编写完成了，进行make来得到模块文件。

装载模块

```
sudo insmod papadev.ko
```

dmesg查看日志

```
[ 6443.015198] cdev_init is called!
[ 6443.015203] hello,my id is 768
[ 6443.015206] [papa] device successfully initialized.
```

安装完之后再查看一下设备，\$ cat /proc/devices

结果是

```
3 /dev/pdmx
5 ttyprintk
6 lp
7 vcs
10 misc
13 input
14 sound/midi
14 sound/dmmidi
21 sg
29 fb
60 papa_cdev
89 i2c
99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
202 cpu/msr
```

发现了我们的设备

但是还差一步，就是要手动添加一个设备结点，

```
sudo mknod /dev/papa_cdev c 60 0
```

c表示是一个字符设备，60对应的你上面程序的主设备号，0是次设备号

接下来在用户测试程序中就可以通过 `/dev/papa_cdev` 这个路径来访问设备啦。

不过最后还要修改一下权限，对这个设备文件的访问

```
chmod 777 /dev/papa_cdev
```

输入这个查看我们手动添加的设备结点 `ll /dev/papa_cdev`

没问题，添加成功了。

```
crwxrwxrwx 1 root root 60, 0 5月 26 15:49 /dev/papa_cdev
```

接下来我们开始测试

test1:

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<string.h>
```

```

#include<sys/ioctl.h>
#include"test_cmd.h"
#define MAX_SIZE 1024
#define MY_DEVICE_NAME "/dev/papa_cdev"
int main(){
    int fd=open(MY_DEVICE_NAME,O_RDWR);
    if(fd!=-1){
        char write_buff[]="I am from user\n";
        char read_buff[1024];
        int ret=0;
        //test the first read is id
        read(fd,read_buff,1024);
        printf("read_buff : %s\n",read_buff);

        //test write and read
        printf("-----test write and read-----\n");
        if(ret=write(fd,write_buff,strlen(write_buff))){
            printf("write successfully!\n");
        }
        read(fd,read_buff,1024);
        printf("read_buff : %s\n",read_buff);

        //test
        // printf("-----test parts over 1024 are discarded-----\n");
        // char rubb[1026];
        // printf("write 1023 s into rubb\n");
        // for(int i=0;i<1023;i++){
        //     rubb[i]='s';
        // }
        // rubb[1023]='a';
        // rubb[1024]='b';
        // rubb[1025]='\0';
        // write(fd,rubb,1026);
        // read(fd,read_buff,1024);
        // printf("read_buff : %s\n",read_buff);

        //test read old by ioctl
        printf("-----test read old by ioctl-----\n");
        ioctl(fd,READ_OLD,read_buff);
        printf("get ori_rwbuff : %s\n",read_buff);

        //test read new by ioctl
        printf("-----test read new by ioctl-----\n");
        write(fd,"test read by ioctl",19);
        ioctl(fd,READ_NEW,read_buff);
        printf("get new_rwbuff :%s\n",read_buff);

        //clear
        printf("-----test clear by ioctl-----\n");
        ioctl(fd,RW_CLEAR);
        read(fd,read_buff,1024);
        printf("read_buff : %s\n",read_buff);

    }else{

```



```

        printf("fail to open %s",MY_DEVICE_NAME);
    }
    close(fd);

    return 0;
}

```

执行之后发现

先读到了我们的学号

然后也成功的写入，并通过ioctl测试了获得最近一次写入前的数据和当前的数据都没问题！

最后测试ioctl的clear功能也成功了！

```

read_buff : 768
-----test write and read-----
write successfully!
read_buff : I am from user

-----test read old by ioctl-----
get ori_rwbuff : 768
-----test read new by ioctl-----
get new_rwbuff :test read by ioctl
-----test clear by ioctl-----
read_buff :

```

再来测试不能重复打开的功能

```

#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include<string.h>
#define MY_DEVICE_NAME "/dev/papa_cdev"
int main(){
    int fd=open(MY_DEVICE_NAME,O_RDWR);
    if(fd!=-1){
        printf("open successfully!\n");
        fd=open(MY_DEVICE_NAME,O_RDWR);//这里重复打开
        if(fd==-1){
            printf("fail to reopen\n");
        }
    }
    close(fd);

    return 0;
}

```

```

open successfully!
fail to reopen

```

最后测试最多写入1024个字符，多的会被截断

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<string.h>
#define MY_DEVICE_NAME "/dev/papa_cdev"
#define MAX_SIZE 1030
int main(){
    int fd=open(MY_DEVICE_NAME,O_RDWR);
    if(fd!=-1){
        char rubb[MAX_SIZE];
        for(int i=0;i<MAX_SIZE;i++){
            rubb[i]='z';
        }
        rubb[MAX_SIZE-1]='\0';//in fact ,write bytes is 1029
        rubb[1023]='a';
        int size=strlen(rubb);
        // printf("%d\n",size);
        int ret=write(fd,rubb,size);
        if(ret!=-1){
            printf("write rubb successfully!\n");
            char read_buff[MAX_SIZE];
            read(fd,read_buff,MAX_SIZE);
            printf("read_buff is %s\n",read_buff);
            printf("bytes is %ld",strlen(read_buff));

        }else{
            printf("fail to write rubb!\n");
        }

    }else{
        printf("fail to open %s\n",MY_DEVICE_NAME);
    }
    close(fd);

    return 0;
}
```

[illegible]

