·四川大学计算机学院学院

实 验 报 告

学号：2022141460176 姓名：杨一舟 专业：计算机科学与技术 第 14 周

课程名称	操作系统课程设计	实验课时	2
实验项目	虚拟字符设备驱动	实验时间	2024.6.3
实验目的	理解字符设备驱动的开发步骤，这包括编写驱动模块代码、通过编译工具链进行编译、将编译后的模块加载到 Linux 内核中、对模块进行功能测试，以及在测试完成后安全地卸载该模块。		
实验环境	VS code 的 ssh 远程连接插件 云服务器 Connect.westc.gpuhub.com		
实验内容	**实验设计** 1. 编写驱动程序 (globalvar.c)： 2. 编写测试程序 (read.c 和 write.c)： 3. 编写 Makefile： 4. 编译驱动程序： 5. 加载驱动程序： 6. 检查设备文件： 7. 测试读写功能： 8. 卸载驱动程序： **实验步骤** **1、编写驱动与测试程序（在 VS code 中展示）** 		

Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := globalvar.o#obj-m 指编译成外部模块
else
    KERNELDIR := /lib/modules/$(shell uname -r)/build
    #定义一个变量,指向内核目录
    PWD := $(shell pwd)
modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
clean:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

Read.c

```
#include<sys/types.h>
#include<unistd.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
int main()
{
    int fd,i;
    char msg[101];
    fd= open("/dev/chardev0",O_RDWR,S_IRUSR|S_IWUSR);
    if(fd!=-1)
    {
        while(1)
        {
            for(i=0;i<101;i++)
                msg[i]='\0';
            read(fd,msg,100);
            printf("%s\n",msg);
            if(strcmp(msg,"quit")==0)
            {
                close(fd);
                break;
            }
```

```
        }
    }
    else
    {
        printf("device open failure,%d\n",fd);
    }
    return 0;
}
```

Write.c

```
#include<sys/types.h>
#include<unistd.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
int main()
{
    int fd;
    char msg[100];
    fd= open("/dev/chardev0",O_RDWR,S_IRUSR|S_IWUSR);
    if(fd!=-1)
    {
        while(1)
        {
            printf("Please input the globar:\n");
            scanf("%s",msg);
            write(fd,msg,strlen(msg));
            if(strcmp(msg,"quit")==0)
            {
                close(fd);
                break;
            }
        }
    }
    else
    {
        printf("device open failure\n");
    }
    return 0;
```

```
}
```

Globalvar.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/wait.h>
#include <linux/semaphore.h>
#include <linux/sched.h>
#include <linux/cdev.h>
#include <linux/types.h>
#include <linux/kdev_t.h>
#include <linux/device.h>

#define MAXNUM 100
#define MAJOR_NUM 456 //主设备号 ,没有被使用

struct dev{
    struct cdev devm; //字符设备
    struct semaphore sem;
    wait_queue_head_t outq;//等待队列,实现阻塞操作
    int flag; //阻塞唤醒标志
    char buffer[MAXNUM+1]; //字符缓冲区
    char *rd,*wr,*end; //读,写,尾指针
}globalvar;
static struct class *my_class;
int major=MAJOR_NUM;

static ssize_t globalvar_read(struct file *,char *,size_t ,loff_t *);
static ssize_t globalvar_write(struct file *,const char *,size_t ,loff_t *);
static int globalvar_open(struct inode *inode,struct file *filp);
static int globalvar_release(struct inode *inode,struct file *filp);
/*
结构体 file_operations 在头文件 linux/fs.h 中定义，用来存储驱动内核模块提供的对设备进行各种操作的
函数的指针。
*/
struct file_operations globalvar_fops =
{
    //用来从设备中获取数据
    .read=globalvar_read,
```

```
    //发送数据给设备
    .write=globalvar_write,
     .open=globalvar_open,
    //当最后一个打开设备的用户进程执行 close ()系统调用时，内核将调用驱动程序的 release () 函数：
release 函数的主要任务是清理未结束的输入/输出操作、释放资源、用户自定义排他标志的复位等。
    .release=globalvar_release,
};
//内核模块的初始化
static int globalvar_init(void)
{


    int result = 0;
    int err = 0;


    dev_t dev = MKDEV(major, 0);//
    if(major)
    {
        //静态申请设备编号
        result = register_chrdev_region(dev, 1, "charmem");
    }
    else
    {
        //动态分配设备号
        result = alloc_chrdev_region(&dev, 0, 1, "charmem");
        major = MAJOR(dev);
    }
    if(result < 0)
        return result;
      //注册字符设备驱动，设备号和 file_operations 结构体进行绑定
    cdev_init(&globalvar.devm, &globalvar_fops);


    globalvar.devm.owner = THIS_MODULE;
    err = cdev_add(&globalvar.devm, dev, 1);
    if(err)
        printk(KERN_INFO "Error %d adding char_mem device", err);
    else
    {
        printk("globalvar register success\n");
```

```
        sema_init(&globalvar.sem,1); //初始化信号量
        init_waitqueue_head(&globalvar.outq); //初始化等待队列
        globalvar.rd = globalvar.buffer; //读指针
        globalvar.wr = globalvar.buffer; //写指针
        globalvar.end = globalvar.buffer + MAXNUM;//缓冲区尾指针
        globalvar.flag = 0; // 阻塞唤醒标志置 0
    }
    /*
    定义在/include/linux/device.h
    创建 class 并将 class 注册到内核中，返回值为 class 结构指针
    在驱动初始化的代码里调用 class_create 为该设备创建一个 class，再为每个设备调用 device_create
创建对应的设备。
    省去了利用 mknod 命令手动创建设备节点
    */
    my_class = class_create(THIS_MODULE, "chardev0");
    device_create(my_class, NULL, dev, NULL, "chardev0");
    return 0;
}

static int globalvar_open(struct inode *inode,struct file *filp)
{
    try_module_get(THIS_MODULE);//模块计数加一
    printk("This chrdev is in open\n");
    return(0);
}

static int globalvar_release(struct inode *inode,struct file *filp)
{
    module_put(THIS_MODULE); //模块计数减一
    printk("This chrdev is in release\n");
    return(0);
}
static void globalvar_exit(void)
{
    device_destroy(my_class, MKDEV(major, 0));
    class_destroy(my_class);
    cdev_del(&globalvar.devm);
    unregister_chrdev_region(MKDEV(major, 0), 1);//注销设备
}
```

```
static ssize_t globalvar_read(struct file *filp,char *buf,size_t len,loff_t *off)
{
    if(wait_event_interruptible(globalvar.outq,globalvar.flag!=0)) //不可读时 阻塞读进程
    {
        return -ERESTARTSYS;
    }

    if(down_interruptible(&globalvar.sem)) //P 操作
    {
        return -ERESTARTSYS;
    }
    globalvar.flag = 0;
    printk("into the read function\n");
    printk("the rd is %c\n",*globalvar.rd); //读指针
    if(globalvar.rd < globalvar.wr)
        len = min(len,(size_t)(globalvar.wr - globalvar.rd)); //更新读写长度
    else
        len = min(len,(size_t)(globalvar.end - globalvar.rd));
    printk("the len is %d\n",len);
    /*
    read 和 write 代码要做的工作,就是在用户地址空间和内核地址空间之间进行整段数据的拷贝。
    */
    if(copy_to_user(buf,globalvar.rd,len))
    {
        printk(KERN_ALERT"copy failed\n");
        /*
        up 递增信号量的值,并唤醒所有正在等待信号量转为可用状态的进程。
         */
        up(&globalvar.sem);
        return -EFAULT;
    }
    printk("the read buffer is %s\n",globalvar.buffer);
    globalvar.rd = globalvar.rd + len;
    if(globalvar.rd == globalvar.end)
        globalvar.rd = globalvar.buffer; //字符缓冲区循环
    up(&globalvar.sem); //V 操作
    return len;
}
static ssize_t globalvar_write(struct file *filp,const char *buf,size_t len,loff_t *off)
{
```

```
    if(down_interruptible(&globalvar.sem)) //P 操作
    {
        return -ERESTARTSYS;
    }
    if(globalvar.rd <= globalvar.wr)
        len = min(len,(size_t)(globalvar.end - globalvar.wr));
    else
        len = min(len,(size_t)(globalvar.rd-globalvar.wr-1));
    printk("the write len is %d\n",len);
    if(copy_from_user(globalvar.wr,buf,len))
    {
        up(&globalvar.sem); //V 操作
        return -EFAULT;
    }
    printk("the write buffer is %s\n",globalvar.buffer);
    printk("the len of buffer is %d\n",strlen(globalvar.buffer));
    globalvar.wr = globalvar.wr + len;
    if(globalvar.wr == globalvar.end)
    globalvar.wr = globalvar.buffer; //循环
    up(&globalvar.sem);//V 操作
    globalvar.flag=1; //条件成立,可以唤醒读进程
    wake_up_interruptible(&globalvar.outq); //唤醒读进程
    return len;
}
module_init(globalvar_init);
module_exit(globalvar_exit);
MODULE_LICENSE("GPL");
```

2、创建设备文件并在 Dev 目录下检查

```
(base) root@autodl-container-1f864284bf-f38ce9f1:~/exp9# sudo insmod globalvar.ko
(base) root@autodl-container-1f864284bf-f38ce9f1:~/exp9# sudo mknod /dev/charput0 c 241 0
(base) root@autodl-container-1f864284bf-f38ce9f1:~/exp9# ls /dev
charput0  fd          null              nvidia9     pts     stderr  tty
core      full        nvidia-uvm        nvidiactl   random  stdin   urandom
dri       mqueue      nvidia-uvm-tools  ptmx        shm     stdout  zero
```

可以看到已经成功创建了 charput0 这一文件

3、运行用户程序 read 与 write

打开一个终端运行 `read`
./read

	打开另一个终端运行 `write` ./write (base) root@autodl-container-1f864284bf-f38ce9f1:~/exp9# ./write Please input the globar: HELLO WORLD **4、查看用户程序的输出** HELLO WORLD HELLO WORLD HELLO WORLD HELLO WORLD 输出结果与输入内容一致 **5、卸载内核模块并清理编译文件** (base) root@autodl-container-1f864284bf-f38ce9f1:~/exp9# sudo rmmod globalvar (base) root@autodl-container-1f864284bf-f38ce9f1:~/exp9# make clean make -C /lib/modules/3.10.0-957.21.3.el7.x86_64/build M=/root/os/experiment8 clean make[1]: Entering directory `/usr/src/kernels/3.10.0-957.21.3.el7.x86_64' CLEAN /root/os/experiment8/.tmp_versions CLEAN /root/os/experiment8/Module.symvers make[1]: Leaving directory `/usr/src/kernels/3.10.0-957.21.3.el7.x86_64'
实验结果	通过以上步骤，可以成功创建一个虚拟字符设备驱动，并通过用户空间程序进行读写测试。
小　结	本次虚拟字符设备驱动实验让我深入理解了 Linux 内核驱动的开发流程。通过编写和加载驱动模块，我理解了字符设备驱动的基本框架和原理，也熟悉了字符设备驱动的工作机制和内核编程的步骤。
指导老师评　议	成绩评定：　　指导教师签名：