Linux内核基础——Linux驱动开发(实训二,linux内核等待队列、自旋锁、信号量等机制)

# Linux驱动开发实训二

## 任务一、使用等待队列实现用户进程对驱动的堵塞与非堵塞读写

**内核模块：hello\_char.c**

#include<linux/module.h>  
#include<linux/kernel.h>  
#include<linux/init.h>  
#include<linux/cdev.h>  
#include<linux/fs.h>  
#include<linux/sched.h>  
#include<linux/string.h>  
#include<linux/uaccess.h>  
  
#define HELLO\_MAJOR 252  
#define CONTENT\_SIZE 1000  
static dev\_t hello\_devno;  
static struct cdev \*hello\_cdev;  
static struct device \*dev;  
static struct class \*cls;  
static char content[CONTENT\_SIZE]; //模拟设备文件容量为100字节  
  
static wait\_queue\_head\_t my\_quehead;  
  
static int hello\_open(struct inode \*pnode, struct file \*filp)  
{  
 printk(KERN\_ERR "This inode's Id is %lu \n", pnode->i\_ino);  
 return 0;  
}  
static ssize\_t hello\_write(struct file \*filp, const char \*buf, size\_t size, loff\_t \*p)  
{  
 int ret;  
 //是否发生写溢出，loff是否值合理 ; loff + 写入字节不应大于设备文件容量的数组分界  
 if(\*p < 0) {  
 return -1;  
 printk(KERN\_ERR "1write error!\n");  
 } else if(\*p + size > CONTENT\_SIZE ) {  
 printk(KERN\_ERR "2write error!\n");  
 return -1;  
 }  
 //此函数可以交换内核与userspace的数据  
 ret = copy\_from\_user((void\*)(content + \*p), (void\*) buf, size);   
 if(ret) {  
 printk(KERN\_ERR "3write error!\n");  
 return ret;  
 }  
 //移动文件指针  
 \*p += size;  
 printk("success!\n");  
 //如果设备文件中有内容，唤醒等待队列  
 if(strlen(content) > 0)  
 wake\_up\_interruptible(&my\_quehead);  
 //成功返回写入字节数，错误返回-1  
 return ret;  
}  
static ssize\_t hello\_read(struct file \*filp, char \*buf, size\_t size, loff\_t \*p)  
{  
 int ret;  
 //判断设备文件打开方式是阻塞还是非阻塞  
 if (filp->f\_flags & O\_NONBLOCK) {  
 printk(KERN\_INFO "non-block read!\n");  
 //如果无内容直接返回错误码，有内容则向下执行copy操作  
 if (strlen(content) == 0)  
 return -EAGAIN;  
 } else {  
 printk(KERN\_INFO "block read!\n");  
 //会先判断参数2：condition是否为真，为真向下执行，为假则进程进入睡眠态，由等待队列维护此进程  
 wait\_event\_interruptible(my\_quehead, strlen(content) > 0);  
 }  
  
  
 //是否发生读溢出，loff是否值合理 ; loff + 读取字节不应大于设备文件容量的数组分界  
 if (\*p < 0) {  
 printk(KERN\_ERR "1read error!\n");  
 return -1;  
 } else if(\*p + size > CONTENT\_SIZE ) {  
 printk(KERN\_ERR "2read error!\n");  
 return -1;  
 }  
 ret = copy\_to\_user((void\*)buf, (void\*)(content + \*p), size);  
 if (ret) {  
 printk(KERN\_ERR "3read error!\n");  
 return ret;  
 }  
 printk("sucess!\n");  
 //成功返回读出字节数，错误返回-1  
 return size;  
}  
//注册驱动文件操作集  
struct file\_operations hello\_ops = {  
 .owner = THIS\_MODULE,  
 .open = hello\_open,  
 .read = hello\_read,  
 .write = hello\_write,  
};  
//初始化内核模块  
static int init\_hello(void)  
{  
 int ret;  
 //初始化等待队列  
 init\_waitqueue\_head(&my\_quehead);  
 //宏转换设备号为系统可识别的32位二进制  
 hello\_devno = MKDEV(HELLO\_MAJOR, 0);  
  
 //自定义向内核申请驱动设备号  
 ret = register\_chrdev\_region(hello\_devno, 1, "zjcchar");  
 if (ret) {  
 //如申请失败，则动态由内核分配  
 ret = alloc\_chrdev\_region(&hello\_devno, 0, 1, "zjcchar");  
 if (ret) {  
 printk(KERN\_ERR "allocchr error!\n");  
 goto error0;  
 }  
 }  
 //动态创建一个驱动设备结构体  
 hello\_cdev = cdev\_alloc();  
 if (hello\_cdev == NULL) {  
 printk(KERN\_ERR "alloc error!\n");  
 ret = -1;  
 goto error1;   
 }  
 //初始化驱动设备结构体，与驱动文件操作集捆绑  
 cdev\_init(hello\_cdev, &hello\_ops);  
 //驱动设备结构体与设备号关联，并交由内核相关链表中维护  
 ret = cdev\_add(hello\_cdev, hello\_devno, 1);  
 if (ret) {  
 printk(KERN\_ERR "add error!\n");  
 goto error2;  
 }  
 //创建一个class类与当前模块绑定,供创建设备节点的API使用  
 cls = class\_create(THIS\_MODULE, "testclass");  
 if (IS\_ERR(cls)) {  
 ret = PTR\_ERR(dev);  
 printk(KERN\_ERR "class\_create error!\n");  
 goto error2;  
 }  
 //创建一个设备节点（文件）,指明设备节点与hello\_cdev中的设备号相关联  
 //并取名为testdev(会创建在/dev/testdev)  
 dev = device\_create(cls, NULL, hello\_cdev->dev, NULL, "testdev");  
 if (IS\_ERR(dev)) {  
 ret = PTR\_ERR(dev);  
 printk(KERN\_ERR "device\_create error!\n");  
 goto error3;  
 }  
 printk(KERN\_ERR "Hello, World!\n");  
 return 0;  
//一个简洁高效的错误处理机制  
error3:  
 class\_destroy(cls);  
error2:  
 cdev\_del(hello\_cdev);  
error1:   
 unregister\_chrdev\_region(hello\_devno, 1);  
error0:  
 return ret;  
}  
//清理并卸载内核模块,注意清理顺序是倒序的，因为各结构体之间有依赖关系  
static void cleanup\_hello(void)  
{  
 device\_destroy(cls, hello\_devno);  
 class\_destroy(cls);  
 cdev\_del(hello\_cdev);  
 unregister\_chrdev\_region(hello\_devno, 1);  
 printk(KERN\_ERR "Goodbye, world!\n");  
}  
module\_init(init\_hello);  
module\_exit(cleanup\_hello);  
MODULE\_LICENSE("GPL");  
MODULE\_AUTHOR("zjc");  
MODULE\_DESCRIPTION("A test project");

动态创建设备号、设备文件、绑定驱动程序文件集。read中先判断文件是阻塞式打开方式还是非阻塞打开方式。再判断设备文件数组缓冲区中是否有内容。  
如果文件是阻塞式打开，有内容则返回内容，无内容时进程由等待队列维护进入睡眠态，写入内容后被唤醒。  
如果文件是非阻塞式打开，有内容则返回内容，无内容则返回一个错误码。

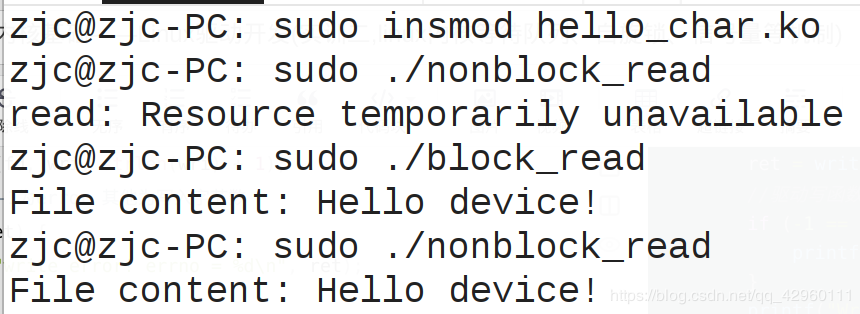
**测试文件：read.c**

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include<unistd.h>  
#include<string.h>  
#include<stropts.h>  
int main()  
{  
 int fd;  
 int ret = 0;  
 char a[500];  
   
 //阻塞式文件操作  
 fd = open("/dev/testdev", O\_RDWR);  
 //非阻塞式文件操作：  
 //fd = open("/dev/testdev", O\_RDWR | O\_NONBLOCK);  
 if (fd < 0) {  
 perror("open");  
 exit(EXIT\_FAILURE);  
 } else {  
 ret = read(fd, a, 13 + 1);   
 if (-1 == ret) {  
 printf("read error! errno = %d \n", ret);  
 }  
 }  
 close(fd);  
 printf("File content: %s\n", a);  
 return 0;  
}

**测试文件：write.c**

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include<unistd.h>  
#include<string.h>  
#include<stropts.h>  
int main()  
{  
 int fd;  
 int ret = 0;  
 char \*wri = "Hello device!";  
  
 fd = open("/dev/testdev", O\_RDWR);  
 if (fd < 0) {  
 perror("open");  
 exit(EXIT\_FAILURE);  
 } else {  
 //调用驱动写函数  
 ret = write(fd, wri, strlen(wri) + 1);  
 //驱动写函数 -1为error 其他为写入字节数  
 if (-1 == ret) {  
 printf("write error! errno = %d\n", ret);  
 }  
 printf("Write success!\n");  
 }  
 close(fd);  
 return 0;  
}

**Makefile见往期实训文档**

**结果截图**  
  
加载模块后，先是非阻塞读，由于设备驱动数组中无内容，所以内核直接返回错误码，在测试程序中用perror捕捉打印错误信息。

然后再执行阻塞读，会发现进程阻塞进入睡眠态，此时开启另外的中断执行写程序，唤醒了睡眠态，读取并打印出内容。

之后再次使用非阻塞读，直接返回数组内容并打印。

## 任务二、使用自旋锁与信号量对驱动中的公共资源进行保护

此任务主要是学习当对公共资源并行操作时如何避免出现异常，下面会在任务一的代码基础上使用自旋锁和信号量对写操作进行保护。避免并行写入出现数据异常。

自旋锁与信号量在开发人员的使用角度上来看，其实是比较相似的，当信号量的资源值val设置为1时用法和自旋锁相同。自旋锁抢占锁就相当于信号量down来拿下一个资源。自旋锁释放锁就相当于信号量up来释放一个资源。但其实信号量可以在初始化时定义多个资源，也就是设置val值时可以设置比1更大的数。对于信号量的down来说就是对资源val原子减1，up就是对资源val的原子加1。当资源数val为0时，表示没有资源可用，会睡眠等待资源等待被唤醒。

从进程与cpu的底层角度上来看，自旋锁与信号量是不同的。因为自旋锁在抢占锁（或者说等待进程释放这把锁）时，是不断的“旋转等待”的，也就是cpu会不断地去判断这把锁是否已经被释放。优点在于速度很快，减少了cpu调度切换的开销；缺点在于如果等待时间太长，则浪费了cpu的资源。信号量在等待资源的释放时，是进程挂起的睡眠态，也就是cpu将被调度去做其他事情。优点是当等待时间较长时节省了cpu的资源；缺点是cpu调度切换是需要开销的，不适合等待时间短的情况。  
一般来说，自旋锁用在中断的上下文中，因为中断对时间的要求特别高，中断的上半部是绝对不能睡眠的，而且处理时间很短，所以非常适合自旋锁使用。而信号量则适合等待时间较长的场景，在内核驱动中，使用信号量的场景还是非常多的。

### 1、使用自旋锁机制

**驱动模块：spinlock\_dev.c**

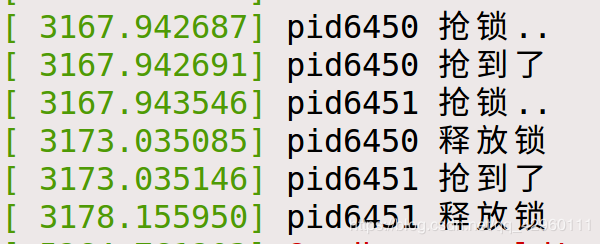
#include<linux/module.h>  
#include<linux/kernel.h>  
#include<linux/init.h>  
#include<linux/cdev.h>  
#include<linux/fs.h>  
#include<linux/sched.h>  
#include<linux/string.h>  
#include<linux/uaccess.h>  
#include<linux/delay.h>  
  
#define HELLO\_MAJOR 252  
#define CONTENT\_SIZE 1000  
static dev\_t hello\_devno;  
static struct cdev \*hello\_cdev;  
static struct device \*dev;  
static struct class \*cls;  
static char content[CONTENT\_SIZE]; //模拟设备文件容量为100字节  
static spinlock\_t spin;  
  
static wait\_queue\_head\_t my\_quehead;  
  
static int hello\_open(struct inode \*pnode, struct file \*filp)  
{  
 printk(KERN\_ERR "This inode's Id is %lu \n", pnode->i\_ino);  
 return 0;  
}  
static ssize\_t hello\_write(struct file \*filp, const char \*buf, size\_t size, loff\_t \*p)  
{  
 int ret;  
 //是否发生写溢出，loff是否值合理 ; loff + 写入字节不应大于设备文件容量的数组分界  
 if(\*p < 0) {  
 return -1;  
 printk(KERN\_ERR "1write error!\n");  
 } else if(\*p + size > CONTENT\_SIZE ) {  
 printk(KERN\_ERR "2write error!\n");  
 return -1;  
 }  
 printk(KERN\_INFO "pid%d 抢锁..\n", current->pid);  
 spin\_lock(&spin);  
 printk(KERN\_INFO "pid%d 抢到了\n", current->pid);  
 //睡眠5秒保证可观测到预期结果  
 msleep(5000);  
 //此函数可以交换内核与userspace的数据  
 ret = copy\_from\_user((void\*)(content + \*p), (void\*) buf, size);   
 printk(KERN\_INFO "pid%d 释放锁\n", current->pid);  
 spin\_unlock(&spin);  
 if(ret) {  
 printk(KERN\_ERR "3write error!\n");  
 return ret;  
 }  
 //移动文件指针  
 \*p += size;  
 //如果设备文件中有内容，唤醒等待队列  
 if(strlen(content) > 0)  
 wake\_up\_interruptible(&my\_quehead);  
 //成功返回写入字节数，错误返回-1  
 return ret;  
}  
static ssize\_t hello\_read(struct file \*filp, char \*buf, size\_t size, loff\_t \*p)  
{  
 int ret;  
 //判断设备文件打开方式是阻塞还是非阻塞  
 if (filp->f\_flags & O\_NONBLOCK) {  
 printk(KERN\_INFO "non-block read!\n");  
 //如果无内容直接返回错误码，有内容则向下执行copy操作  
 if (strlen(content) == 0)  
 return -EAGAIN;  
 } else {  
 printk(KERN\_INFO "block read!\n");  
 //会先判断参数2：condition是否为真，为真向下执行，为假则进程进入睡眠态，由等待队列维护此进程  
 wait\_event\_interruptible(my\_quehead, strlen(content) > 0);  
 }  
  
  
 //是否发生读溢出，loff是否值合理 ; loff + 读取字节不应大于设备文件容量的数组分界  
 if (\*p < 0) {  
 printk(KERN\_ERR "1read error!\n");  
 return -1;  
 } else if(\*p + size > CONTENT\_SIZE ) {  
 printk(KERN\_ERR "2read error!\n");  
 return -1;  
 }  
 ret = copy\_to\_user((void\*)buf, (void\*)(content + \*p), size);  
 if (ret) {  
 printk(KERN\_ERR "3read error!\n");  
 return ret;  
 }  
 //成功返回读出字节数，错误返回-1  
 return size;  
}  
//注册驱动文件操作集  
struct file\_operations hello\_ops = {  
 .owner = THIS\_MODULE,  
 .open = hello\_open,  
 .read = hello\_read,  
 .write = hello\_write,  
};  
//初始化内核模块  
static int init\_hello(void)  
{  
 int ret;  
 //初始化等待队列  
 init\_waitqueue\_head(&my\_quehead);  
 //初始化自旋锁  
 spin\_lock\_init(&spin);  
 //宏转换设备号为系统可识别的32位二进制  
 hello\_devno = MKDEV(HELLO\_MAJOR, 0);  
  
 //自定义向内核申请驱动设备号  
 ret = register\_chrdev\_region(hello\_devno, 1, "zjcchar");  
 if (ret) {  
 //如申请失败，则动态由内核分配  
 ret = alloc\_chrdev\_region(&hello\_devno, 0, 1, "zjcchar");  
 if (ret) {  
 printk(KERN\_ERR "allocchr error!\n");  
 goto error0;  
 }  
 }  
 //动态创建一个驱动设备结构体  
 hello\_cdev = cdev\_alloc();  
 if (hello\_cdev == NULL) {  
 printk(KERN\_ERR "alloc error!\n");  
 ret = -1;  
 goto error1;   
 }  
 //初始化驱动设备结构体，与驱动文件操作集捆绑  
 cdev\_init(hello\_cdev, &hello\_ops);  
 //驱动设备结构体与设备号关联，并交由内核相关链表中维护  
 ret = cdev\_add(hello\_cdev, hello\_devno, 1);  
 if (ret) {  
 printk(KERN\_ERR "add error!\n");  
 goto error2;  
 }  
 //创建一个class类与当前模块绑定,供创建设备节点的API使用  
 cls = class\_create(THIS\_MODULE, "testclass");  
 if (IS\_ERR(cls)) {  
 ret = PTR\_ERR(dev);  
 printk(KERN\_ERR "class\_create error!\n");  
 goto error2;  
 }  
 //创建一个设备节点（文件）,指明设备节点与hello\_cdev中的设备号相关联  
 //并取名为testdev(会创建在/dev/testdev)  
 dev = device\_create(cls, NULL, hello\_cdev->dev, NULL, "testdev");  
 if (IS\_ERR(dev)) {  
 ret = PTR\_ERR(dev);  
 printk(KERN\_ERR "device\_create error!\n");  
 goto error3;  
 }  
 printk(KERN\_ERR "Hello, World!\n");  
 return 0;  
 //一个简洁高效的错误处理机制  
error3:  
 class\_destroy(cls);  
error2:  
 cdev\_del(hello\_cdev);  
error1:   
 unregister\_chrdev\_region(hello\_devno, 1);  
error0:  
 return ret;  
}  
//清理并卸载内核模块,注意清理顺序是倒序的，因为各结构体之间有依赖关系  
static void cleanup\_hello(void)  
{  
 device\_destroy(cls, hello\_devno);  
 class\_destroy(cls);  
 cdev\_del(hello\_cdev);  
 unregister\_chrdev\_region(hello\_devno, 1);  
 printk(KERN\_ERR "Goodbye, world!\n");  
}  
module\_init(init\_hello);  
module\_exit(cleanup\_hello);  
MODULE\_LICENSE("GPL");  
MODULE\_AUTHOR("zjc");  
MODULE\_DESCRIPTION("A test project");

自旋锁用法很简单，在初始化模块时对自旋锁进行初始化，在write中执行copy\_from\_user函数时上锁保护，防止并行写入数据冲突。写好后释放锁。这里注意对于copy\_from\_user返回值的判断一定放在释放锁后，防止出现异常导致锁无法释放，出现死锁。在打debug时，使用current宏获得了执行操作的进程id号，方便清晰看出锁关系。

在测试程序中，使用fork同时创建了两个进程对设备节点进行写操作。

**测试程序 write.h**

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include<unistd.h>  
#include<string.h>  
#include<stropts.h>  
int main()  
{  
 int fd;  
 pid\_t fork\_fd;  
 int ret = 0;  
 char \*wri = "Hello device!";  
  
 fd = open("/dev/testdev", O\_RDWR);  
 if (fd < 0) {  
 perror("open");  
 exit(EXIT\_FAILURE);  
 }  
 fork\_fd = fork();  
 if(fork\_fd < 0) {  
 printf("fork error!\n");  
 return -1;  
 }  
  
 //调用驱动写函数  
 ret = write(fd, wri, strlen(wri) + 1);  
 //驱动写函数 -1为error 其他为写入字节数  
 if (-1 == ret) {  
 printf("write error! errno = %d\n", ret);  
 }  
 printf("Write success!\n");  
  
 return 0;  
}

**结果截图，Makefile见往期实训文档**  
  
结果很清晰，两个几乎同时创建的进程，一个拿到锁后，另外一个需要等待释放。释放后才可以抢到锁。

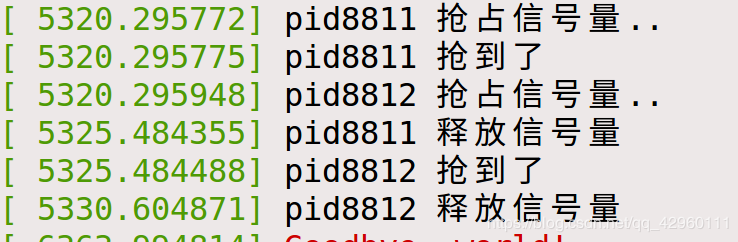
### 2、使用信号量机制

在前面已经讲过，当信号量的资源值val设置为1时，从程序开发人员的角度来说，两个机制使用方式相同。都是先初始化，然后在copy\_form\_user前进行抢占锁/资源，然后释放锁/资源。

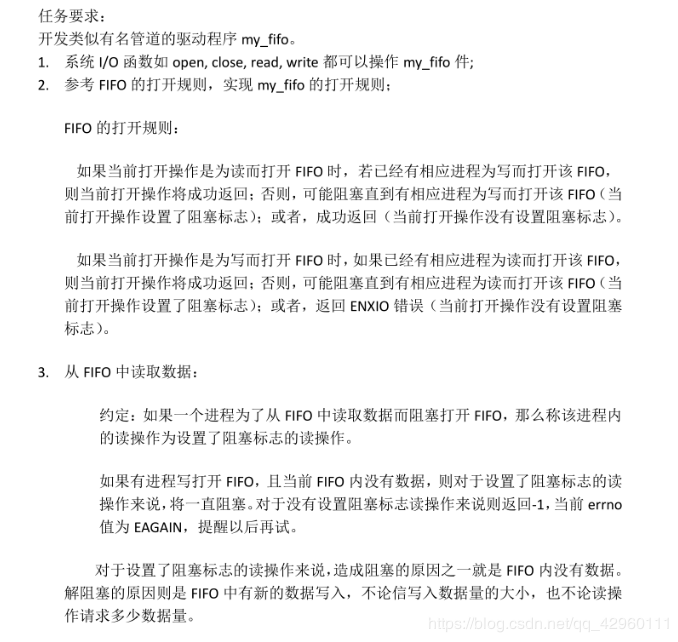
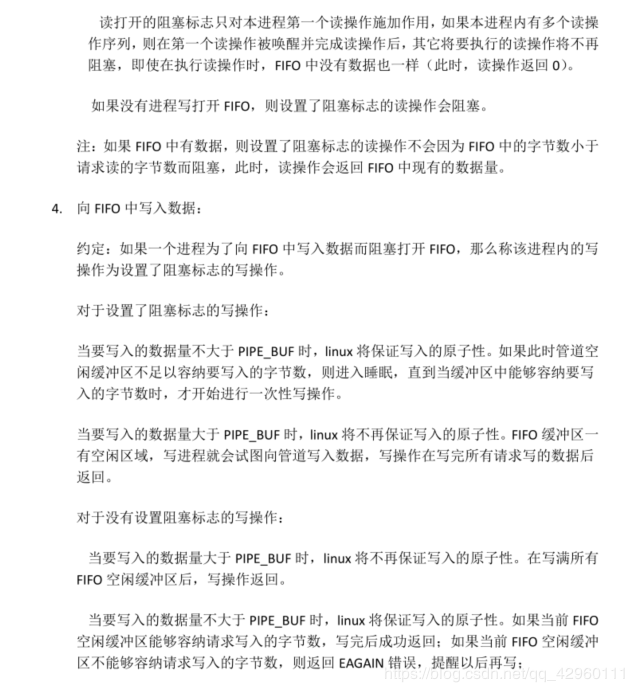
**驱动模块:semaphore\_dev.c**

#include<linux/module.h>  
#include<linux/kernel.h>  
#include<linux/init.h>  
#include<linux/cdev.h>  
#include<linux/fs.h>  
#include<linux/sched.h>  
#include<linux/string.h>  
#include<linux/uaccess.h>  
#include<linux/delay.h>  
  
#define HELLO\_MAJOR 252  
#define CONTENT\_SIZE 1000  
static dev\_t hello\_devno;  
static struct cdev \*hello\_cdev;  
static struct device \*dev;  
static struct class \*cls;  
static char content[CONTENT\_SIZE]; //模拟设备文件容量为100字节  
static struct semaphore sem;  
  
static wait\_queue\_head\_t my\_quehead;  
  
static int hello\_open(struct inode \*pnode, struct file \*filp)  
{  
 printk(KERN\_ERR "This inode's Id is %lu \n", pnode->i\_ino);  
 return 0;  
}  
static ssize\_t hello\_write(struct file \*filp, const char \*buf, size\_t size, loff\_t \*p)  
{  
 int ret;  
 //是否发生写溢出，loff是否值合理 ; loff + 写入字节不应大于设备文件容量的数组分界  
 if(\*p < 0) {  
 return -1;  
 printk(KERN\_ERR "1write error!\n");  
 } else if(\*p + size > CONTENT\_SIZE ) {  
 printk(KERN\_ERR "2write error!\n");  
 return -1;  
 }  
 printk(KERN\_INFO "pid%d 抢信号量..\n", current->pid);  
 down(&sem);  
 printk(KERN\_INFO "pid%d 抢信号量\n", current->pid);  
 //睡眠5秒保证可观测到预期结果  
 msleep(5000);  
 //此函数可以交换内核与userspace的数据  
 ret = copy\_from\_user((void\*)(content + \*p), (void\*) buf, size);   
 printk(KERN\_INFO "pid%d 释放信号量\n", current->pid);  
 up(&sem);  
 if(ret) {  
 printk(KERN\_ERR "3write error!\n");  
 return ret;  
 }  
 //移动文件指针  
 \*p += size;  
 //如果设备文件中有内容，唤醒等待队列  
 if(strlen(content) > 0)  
 wake\_up\_interruptible(&my\_quehead);  
 //成功返回写入字节数，错误返回-1  
 return ret;  
}  
static ssize\_t hello\_read(struct file \*filp, char \*buf, size\_t size, loff\_t \*p)  
{  
 int ret;  
 //判断设备文件打开方式是阻塞还是非阻塞  
 if (filp->f\_flags & O\_NONBLOCK) {  
 printk(KERN\_INFO "non-block read!\n");  
 //如果无内容直接返回错误码，有内容则向下执行copy操作  
 if (strlen(content) == 0)  
 return -EAGAIN;  
 } else {  
 printk(KERN\_INFO "block read!\n");  
 //会先判断参数2：condition是否为真，为真向下执行，为假则进程进入睡眠态，由等待队列维护此进程  
 wait\_event\_interruptible(my\_quehead, strlen(content) > 0);  
 }  
  
  
 //是否发生读溢出，loff是否值合理 ; loff + 读取字节不应大于设备文件容量的数组分界  
 if (\*p < 0) {  
 printk(KERN\_ERR "1read error!\n");  
 return -1;  
 } else if(\*p + size > CONTENT\_SIZE ) {  
 printk(KERN\_ERR "2read error!\n");  
 return -1;  
 }  
 ret = copy\_to\_user((void\*)buf, (void\*)(content + \*p), size);  
 if (ret) {  
 printk(KERN\_ERR "3read error!\n");  
 return ret;  
 }  
 //成功返回读出字节数，错误返回-1  
 return size;  
}  
//注册驱动文件操作集  
struct file\_operations hello\_ops = {  
 .owner = THIS\_MODULE,  
 .open = hello\_open,  
 .read = hello\_read,  
 .write = hello\_write,  
};  
//初始化内核模块  
static int init\_hello(void)  
{  
 int ret;  
 //初始化等待队列  
 init\_waitqueue\_head(&my\_quehead);  
 //初始化信号量，并且设置初始值为1（资源可用）  
 sema\_init(&sem, 1);  
 //宏转换设备号为系统可识别的32位二进制  
 hello\_devno = MKDEV(HELLO\_MAJOR, 0);  
  
 //自定义向内核申请驱动设备号  
 ret = register\_chrdev\_region(hello\_devno, 1, "zjcchar");  
 if (ret) {  
 //如申请失败，则动态由内核分配  
 ret = alloc\_chrdev\_region(&hello\_devno, 0, 1, "zjcchar");  
 if (ret) {  
 printk(KERN\_ERR "allocchr error!\n");  
 goto error0;  
 }  
 }  
 //动态创建一个驱动设备结构体  
 hello\_cdev = cdev\_alloc();  
 if (hello\_cdev == NULL) {  
 printk(KERN\_ERR "alloc error!\n");  
 ret = -1;  
 goto error1;   
 }  
 //初始化驱动设备结构体，与驱动文件操作集捆绑  
 cdev\_init(hello\_cdev, &hello\_ops);  
 //驱动设备结构体与设备号关联，并交由内核相关链表中维护  
 ret = cdev\_add(hello\_cdev, hello\_devno, 1);  
 if (ret) {  
 printk(KERN\_ERR "add error!\n");  
 goto error2;  
 }  
 //创建一个class类与当前模块绑定,供创建设备节点的API使用  
 cls = class\_create(THIS\_MODULE, "testclass");  
 if (IS\_ERR(cls)) {  
 ret = PTR\_ERR(dev);  
 printk(KERN\_ERR "class\_create error!\n");  
 goto error2;  
 }  
 //创建一个设备节点（文件）,指明设备节点与hello\_cdev中的设备号相关联  
 //并取名为testdev(会创建在/dev/testdev)  
 dev = device\_create(cls, NULL, hello\_cdev->dev, NULL, "testdev");  
 if (IS\_ERR(dev)) {  
 ret = PTR\_ERR(dev);  
 printk(KERN\_ERR "device\_create error!\n");  
 goto error3;  
 }  
 printk(KERN\_ERR "Hello, World!\n");  
 return 0;  
//一个简洁高效的错误处理机制  
error3:  
 class\_destroy(cls);  
error2:  
 cdev\_del(hello\_cdev);  
error1:   
 unregister\_chrdev\_region(hello\_devno, 1);  
error0:  
 return ret;  
}  
//清理并卸载内核模块,注意清理顺序是倒序的，因为各结构体之间有依赖关系  
static void cleanup\_hello(void)  
{  
 device\_destroy(cls, hello\_devno);  
 class\_destroy(cls);  
 cdev\_del(hello\_cdev);  
 unregister\_chrdev\_region(hello\_devno, 1);  
 printk(KERN\_ERR "Goodbye, world!\n");  
}  
module\_init(init\_hello);  
module\_exit(cleanup\_hello);  
MODULE\_LICENSE("GPL");  
MODULE\_AUTHOR("zjc");  
MODULE\_DESCRIPTION("A test project");

**测试程序相同，结果截图，Makefile见往期实训文档**

  
解释与上相同。

## 任务三、编写驱动程序，实现与linux中有名管道一样的功能

此题使用循环队列来实现管道输入输出数据，队列队首队尾指针为全局变量。此题功能实现如下：  
  
  
**驱动程序:my\_pipe.c**

#include<linux/module.h>  
#include<linux/kernel.h>  
#include<linux/init.h>  
#include<linux/cdev.h>  
#include<linux/fs.h>  
#include<linux/sched.h>  
#include<linux/string.h>  
#include<linux/uaccess.h>  
#include<linux/delay.h>  
#include<linux/string.h>  
  
#define HELLO\_MAJOR 252  
#define CONTENT\_SIZE 100  
static dev\_t hello\_devno;  
static struct cdev \*hello\_cdev;  
static struct device \*dev;  
static struct class \*cls;  
static char content[CONTENT\_SIZE]; //模拟设备文件容量为100字节  
static struct semaphore sem;  
static spinlock\_t spin;  
//0 read / 1 write  
static int pipe[2] = {0};  
static int rp, wp = 0;  
  
static wait\_queue\_head\_t my\_quehead;  
  
static int hello\_open(struct inode \*pnode, struct file \*filp)  
{  
 if (!(filp->f\_flags & O\_NONBLOCK) && !(0xf & filp->f\_flags)) {  
 printk(KERN\_INFO "以阻塞方式只读打开\n");  
 if(pipe[1] > 0) {  
 pipe[0] ++;  
 } else {  
 pipe[0] ++;  
 wait\_event\_interruptible(my\_quehead, pipe[1] > 0);  
 }  
 } else if ((filp->f\_flags & O\_NONBLOCK) && !(0xf & filp->f\_flags)) {  
 printk(KERN\_INFO "以非阻塞只读方式打开\n");  
 pipe[0] ++;  
 } else if (!(filp->f\_flags & O\_NONBLOCK) && (O\_WRONLY & filp->f\_flags)) {  
 printk(KERN\_INFO "以阻塞只写方式打开\n");  
 if (pipe[0] > 0) {  
 pipe[1] ++;  
 } else {  
 pipe[1] ++;  
 wait\_event\_interruptible(my\_quehead, pipe[0] > 0);  
 }  
 } else if(filp->f\_flags & (O\_WRONLY | O\_NONBLOCK)) {   
 printk(KERN\_INFO "以非阻塞只写方式打开\n");  
 if (pipe[0] > 0) {  
 pipe[1] ++;  
 } else {  
 return -ENXIO;  
 }  
 }  
 printk(KERN\_ERR "This pipe[0][1] is %d,%d \n", pipe[0], pipe[1]);  
 wake\_up\_interruptible(&my\_quehead);  
 return 0;  
}  
static ssize\_t hello\_write(struct file \*filp, const char \*buf, size\_t size, loff\_t \*p)  
{  
 int ret = 0;  
 int i = 0;  
 //写入字节数大于最大的容量  
 if (size > CONTENT\_SIZE - 1) {  
 if (filp->f\_flags & O\_NONBLOCK) {  
 //写满为止  
 while ((wp + 1) % CONTENT\_SIZE != rp) {  
 down(&sem);  
 ret = copy\_from\_user((void\*)(content + wp), (void\*) (buf + i), 1);   
 if(ret) {  
 printk(KERN\_ERR "3write error!\n");  
 ret = -1;  
 break;  
 }  
 spin\_lock(&spin);  
 wp = (wp + 1) % CONTENT\_SIZE;  
 spin\_unlock(&spin);  
 i++;  
 up(&sem);  
 ret = i;  
 }  
 } else {  
 printk("right! size = %d\n", size);  
 //阻塞写则一直写，直到写完  
 while(i < size) {  
 if((wp + 1) % CONTENT\_SIZE != rp) {  
 down(&sem);  
 ret = copy\_from\_user((void\*)(content + wp), (void\*) (buf + i), 1);   
 if(ret) {  
 printk(KERN\_ERR "3write error!\n");  
 ret = -1;  
 break;  
 }  
 spin\_lock(&spin);  
 wp = (wp + 1) % CONTENT\_SIZE;  
 spin\_unlock(&spin);  
 i++;  
 ret = size;  
 up(&sem);  
 } else {  
 printk("满了！i = %d\n", i);  
 msleep(500);  
 }  
 }  
 }  
 } else {  
 //写入字节数不大于最大容量  
  
 //目前的空间不够写  
 if(size > CONTENT\_SIZE - 1 - (wp - rp + CONTENT\_SIZE) % CONTENT\_SIZE) {  
 if (filp->f\_flags & O\_NONBLOCK) {  
 return -EAGAIN;   
 } else {  
 wait\_event\_interruptible(my\_quehead, (CONTENT\_SIZE - 1 - (wp - rp + CONTENT\_SIZE) % CONTENT\_SIZE) >= size);  
 }  
 }  
 //资源够写  
 down(&sem);  
 for (i = 0; i < size; i++) {  
 ret = copy\_from\_user((void\*)(content + wp), (void\*) (buf + i), 1);   
 if(ret) {  
 printk(KERN\_ERR "3write error!\n");  
 ret = -1;  
 break;  
 }  
 spin\_lock(&spin);  
 wp = (wp + 1) % CONTENT\_SIZE;  
 spin\_unlock(&spin);  
 ret = size;  
 }  
 up(&sem);  
 }  
  
  
  
 //如果设备文件中有内容，唤醒等待队列  
 if((wp - rp + CONTENT\_SIZE) % CONTENT\_SIZE > 0)  
 wake\_up\_interruptible(&my\_quehead);  
 //成功返回写入字节数，错误返回-1  
 return ret;  
}  
  
static ssize\_t hello\_read(struct file \*filp, char \*buf, size\_t size, loff\_t \*p)  
{  
 int ret = 0;  
 int i = 0;  
 //判断设备文件打开方式是阻塞还是非阻塞  
 if (filp->f\_flags & O\_NONBLOCK) {  
 printk(KERN\_INFO "non-block read!\n");  
 //如果无内容直接返回错误码，有内容则向下执行copy操作  
 if ((wp - rp + CONTENT\_SIZE) % CONTENT\_SIZE == 0)  
 return -EAGAIN;  
 } else {  
 if(pipe[1] <=0 ) {  
 wait\_event\_interruptible(my\_quehead, pipe[1] > 0);  
 }  
 printk(KERN\_INFO "block read!\n");  
 //会先判断参数2：缓冲区有无内容，为真向下执行，为假则进程进入睡眠态，由等待队列维护此进程  
 wait\_event\_interruptible(my\_quehead, (wp - rp + CONTENT\_SIZE) % CONTENT\_SIZE > 0);  
 }  
  
  
 //是否发生读溢出，loff是否值合理 ; loff + 读取字节不应大于设备文件容量的数组分界  
 if(size <= 0) {  
 return -1;  
 } else if(size > ((wp - rp + CONTENT\_SIZE) % CONTENT\_SIZE)) {  
 printk(KERN\_ERR "2SIZE TO BIG!\n");  
 size = (wp - rp + CONTENT\_SIZE) % CONTENT\_SIZE;  
 }  
  
 for (i = 0; i < size; i++) {  
 ret = copy\_to\_user((void\*)(buf + i), (void\*)(content + rp), 1);  
 if (ret) {  
 printk(KERN\_ERR "3read error!\n");  
 ret = -1;  
 break;  
 }   
 spin\_lock(&spin);  
 rp = (rp + 1) % CONTENT\_SIZE;  
 spin\_unlock(&spin);  
 ret = size;  
 }  
 //有内容读出就唤醒队列  
 if(ret != -1) {  
 wake\_up\_interruptible(&my\_quehead);  
 }  
 //成功返回读出字节数，错误返回-1  
 return ret;  
}  
static int hello\_close(struct inode \*pnode, struct file \*filp)  
{  
 if (!(filp->f\_flags & 0xf)) {  
 printk(KERN\_INFO "只读关闭\n");  
 pipe[0] --;  
 } else if (filp->f\_flags & O\_WRONLY) {  
 printk(KERN\_INFO "只写关闭\n");  
 pipe[1] --;  
 }  
 return 0;  
}  
//注册驱动文件操作集  
struct file\_operations hello\_ops = {  
 .owner = THIS\_MODULE,  
 .open = hello\_open,  
 .read = hello\_read,  
 .write = hello\_write,  
 .release = hello\_close,  
};  
//初始化内核模块  
static int init\_hello(void)  
{  
 int ret;  
 //初始化文件数组  
 memset(content, 0, CONTENT\_SIZE);  
 //初始化自旋锁  
 spin\_lock\_init(&spin);  
 //初始化等待队列  
 init\_waitqueue\_head(&my\_quehead);  
 //初始化信号量，并且设置初始值为1（资源可用）  
 sema\_init(&sem, 1);  
 //宏转换设备号为系统可识别的32位二进制  
 hello\_devno = MKDEV(HELLO\_MAJOR, 0);  
  
 //自定义向内核申请驱动设备号  
 ret = register\_chrdev\_region(hello\_devno, 1, "zjcchar");  
 if (ret) {  
 //如申请失败，则动态由内核分配  
 ret = alloc\_chrdev\_region(&hello\_devno, 0, 1, "zjcchar");  
 if (ret) {  
 printk(KERN\_ERR "allocchr error!\n");  
 goto error0;  
 }  
 }  
 //动态创建一个驱动设备结构体  
 hello\_cdev = cdev\_alloc();  
 if (hello\_cdev == NULL) {  
 printk(KERN\_ERR "alloc error!\n");  
 ret = -1;  
 goto error1;   
 }  
 //初始化驱动设备结构体，与驱动文件操作集捆绑  
 cdev\_init(hello\_cdev, &hello\_ops);  
 //驱动设备结构体与设备号关联，并交由内核相关链表中维护  
 ret = cdev\_add(hello\_cdev, hello\_devno, 1);  
 if (ret) {  
 printk(KERN\_ERR "add error!\n");  
 goto error2;  
 }  
 //创建一个class类与当前模块绑定,供创建设备节点的API使用  
 cls = class\_create(THIS\_MODULE, "testclass");  
 if (IS\_ERR(cls)) {  
 ret = PTR\_ERR(dev);  
 printk(KERN\_ERR "class\_create error!\n");  
 goto error2;  
 }  
 //创建一个设备节点（文件）,指明设备节点与hello\_cdev中的设备号相关联  
 //并取名为testdev(会创建在/dev/testdev)  
 dev = device\_create(cls, NULL, hello\_cdev->dev, NULL, "testdev");  
 if (IS\_ERR(dev)) {  
 ret = PTR\_ERR(dev);  
 printk(KERN\_ERR "device\_create error!\n");  
 goto error3;  
 }  
 printk(KERN\_ERR "Hello, World!\n");  
 return 0;  
 //一个简洁高效的错误处理机制  
error3:  
 class\_destroy(cls);  
error2:  
 cdev\_del(hello\_cdev);  
error1:   
 unregister\_chrdev\_region(hello\_devno, 1);  
error0:  
 return ret;  
}  
//清理并卸载内核模块,注意清理顺序是倒序的，因为各结构体之间有依赖关系  
static void cleanup\_hello(void)  
{  
 device\_destroy(cls, hello\_devno);  
 class\_destroy(cls);  
 cdev\_del(hello\_cdev);  
 unregister\_chrdev\_region(hello\_devno, 1);  
 printk(KERN\_ERR "Goodbye, world!\n");  
}  
module\_init(init\_hello);  
module\_exit(cleanup\_hello);  
MODULE\_LICENSE("GPL");  
MODULE\_AUTHOR("zjc");  
MODULE\_DESCRIPTION("A test project");

**Makefile见往期实训文档，测试程序分别是阻塞、非阻塞读写程序，多且简单不展示。由于此题测试点过多，结果截图简单示意**

**结果截图:**

