scull

字符驱动程序

- 编写字符驱动设备的框架

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/export.h>
#include <linux/fs.h>
#include <linux/cdev.h>

dev_t devts;//利用自动申请设备号的方法申请下来的设备号
int devmin = 0;
int devCount = 1;
char * devname ="demoDev";
int ret;
struct cdev *m_cdev;//申请到的cdev结构体空间



int demo_open (struct inode *inode, struct file *filp)
{
   printk(KERN_INFO"open=============\n");
   return 0;
}

int demo_release (struct inode *inode, struct file *filp)
{
   printk(KERN_INFO"release=============\n");

   return 0;
}
//定义一个字符设备文件操作函数集
struct file_operations fops = {
            .owner = THIS_MODULE,
            .open = demo_open,
            .release = demo_release,
};
int __init demo_init(void)
```

```
{
    //0.申请设备号
    ret = alloc_chrdev_region(&devts,devmin,devCount,devname);//自动申请设备号
    if(ret == 0 || ret < 0){
            printk("alloc_chrdev_region fail\n");
            goto err0;
    }
    else
            printk(KERN_INFO "REGION dev_t is %d \n",devts);//打印设备号
    //1.申请cdev结构体分配空间
    m_cdev = cdev_alloc();
    if(m_cdev == NULL){
            printk("cdev_alloc fail\n");
            goto err1;
    }
    //2.初始化cdev结构体
    //2.1 定义一个字符设备文件操作函数集
    cdev_init(m_cdev, &fops);
    //3.添加cdev结构体到内核，因为要把这个设备交给内核进行管理
    ret = cdev_add(m_cdev, devts, devCount);
    if(ret < 0){
            printk(KERN_INFO "CDEV_ADD ERR");
            goto err2;
    }
    return 0;
err2:
    unregister_chrdev_region(devts,devCount);
    cdev_del(m_cdev);
    return ret;

err1:
    unregister_chrdev_region(devts,devCount);
err0:
    return ret;
}


void  __exit demo_exit(void)
{
    cdev_del(m_cdev);
```

```
75     unregister_chrdev_region(devts,devCount);
76 
77 }
78 
79 
80 
81 
82 
83 
84 module_init(demo_init);
85 module_exit(demo_exit);
86 MODULE_LICENSE("GPL");
87 MODULE_AUTHOR("ZHAOBO");
88 MODULE_DESCRIPTION("This is demo....");
```

一些重要的数据结构

驱动程序涉及到三个重要的数据结构：file_operation ,file,node;

- 文件操作：

  file_operation

  按照惯例file_operation结构或者指向这种结构的指针被称为f_ops。这个数据结构是将**设备编号跟具体的驱动程序连接起来的**

  每个打开的文件file跟一个file_operation相连

- file结构（struct file）

  file结构跟用户空间的FILE没有任何关系，struct file是一个内核结构

  file 表示一个打开的文件（系统中每个打开的文件都有一个在内核空间都有一个对应的file结构**），在open**时候创建，并且将这个文件传递给在该文件上操作的所有函数

  在内核源码中指向struct file 结构的指针通常被称为**filp,**

  - **inode结构**

  内核用inode结构在内部表示文件，file表示打开的文件，对于一个文件，会有打开的多个文件file结构，但是他们都指向单个的inode结构

  inode结构中只有下面两个字段对编写驱动有用

```
1 dev_t                 i_rdev;//包含了真正的设备编号
2 struct cdev           *i_cdev;//struct cdev 是内核中表示字符设备的结构，当inode指向一个
```

下面以scull设备的驱动源码做具体分析

- 字符设备的框架参考上面写的demo
- 接下来看file_operation结构

open方法

open方法提供驱动程序初始化的能力，open应该完成以下任务

1. 检查设备特定的错误（诸如设备未就绪的问题或类似的硬件问题）
2. 如果设备时首次打开，则对其进行初始化
3. 如有必要，对f_op指针进行赋值
4. 分配并填写置于filp-private_data的数据

下面是scull的open方法的源码

```
int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* device information */

    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = 666; /* for other methods */

    /* now trim to 0 the length of the device if open was write-only */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {
            if (down_interruptible(&dev->sem))
                    return -ERESTARTSYS;
            scull_trim(dev); /* ignore errors */
            up(&dev->sem);
    }
    return 0;          /* success */
}

```

对于open函数，是为了打开具体的设备，对于字符设备来说我们需要得到代表字符设备的cdev结构体的指针。inode字段的i_cdev中保存了这个结构体指针

所以需要提供一个方法利用inode->i_cdev来得到表示scull设备的结构体

```
container_of(inode->i_cdev, struct scull_dev, cdev);
```

然后就会返回表示scull设备的结构体的指针，将这个地址保存到file结构的private_data中，方便对该指针的访问

以写方式打开时：将文件长度截取为0

```
struct scull_dev {
    struct scull_qset *data;  /* Pointer to first quantum set */
    int quantum;              /* the current quantum size */一个量子所占空间的大小，总共的内
```

```
    int qset;                 /* the current array size *//中间数组的大小 也可理解量子集中量
```

```
    unsigned long size;       /* amount of data stored here */
    unsigned int access_key;  /* used by sculluid and scullpriv */
    struct semaphore sem;     /* mutual exclusion semaphore     */
    struct cdev cdev;         /* Char device structure              */
};
```

```
scull_trim(dev); /* ignore errors */
int scull_trim(struct scull_dev *dev)
{
    struct scull_qset *next, *dptr;
    int qset = dev->qset;   /* "dev" is not-null */
    int i;

    for (dptr = dev->data; dptr; dptr = next) { /* all the list items */
            if (dptr->data) {
                    for (i = 0; i < qset; i++)
                            kfree(dptr->data[i]);
                    kfree(dptr->data);
                    dptr->data = NULL;
            }
            next = dptr->next;
            kfree(dptr);
    }
    dev->size = 0;
    dev->quantum = scull_quantum;
    dev->qset = scull_qset;
    dev->data = NULL;
    return 0;
}
```

scull的内存使用

在scull中，设备的数据存放在一个单向链表中，链表类型为scull_qset

```
struct scull_qset {
    void **data;
    struct scull_qset *next;
};
```

下面的代码说明如何利用

read和write方法

scull_read

```
ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
                   loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data; //这里是从file结构体中将open操作时放入privat
    struct scull_qset *dptr;        /* the first listitem */
    int quantum = dev->quantum, qset = dev->qset;//得到量子以及量子集的大小
    int itemsize = quantum * qset; /* how many bytes in the listitem */链表的一个节点中共有
    int item, s_pos, q_pos, rest;
    ssize_t retval = 0;

    if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    if (*f_pos >= dev->size)
            goto out;
    if (*f_pos + count > dev->size)
            count = dev->size - *f_pos;

    /* find listitem, qset index, and offset in the quantum */
    item = (long)*f_pos / itemsize; //寻找链表项，也就是链表中的节点的第几个量子
    rest = (long)*f_pos % itemsize;//确定在量子集中的偏移
    s_pos = rest / quantum; q_pos = rest % quantum;// 确定在量子集中的偏移

    /* follow the list up to the right position (defined elsewhere) */
    dptr = scull_follow(dev, item);

    if (dptr == NULL || !dptr->data || ! dptr->data[s_pos])
            goto out; /* don't fill holes */

    /* read only up to the end of this quantum */
    if (count > quantum - q_pos)//如果要读的字节数大于这个链表中的字节数
            count = quantum - q_pos;//就读到文件尾部

    if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {//从内核空间将数据读到用户空间
            retval = -EFAULT;
            goto out;
```

```
    }
    *f_pos += count;//更新文件偏移
    retval = count;

  out:
    up(&dev->sem);
    return retval;
}
```

总结来说

1. 如果返回值等于传递给read系统调用的count参数，则说明所请求的字节数传输成功完成
2. 如果返回值是正的但是比count小，则说明部分数据成功
3. 如果返回值为0，则表示已经倒了文件尾部

Write方法

read和write方法类似（每次只处理一个量子）

```
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count,
                loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset;
    int item, s_pos, q_pos, rest;
    ssize_t retval = -ENOMEM; /* value used in "goto out" statements */

    if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;

    /* find listitem, qset index and offset in the quantum */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;

    /* follow the list up to the right position */
    dptr = scull_follow(dev, item);
    if (dptr == NULL)
            goto out;
    if (!dptr->data) {
```

```
24              dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
25              if (!dptr->data)
26                      goto out;
27              memset(dptr->data, 0, qset * sizeof(char *));
28      }
29      if (!dptr->data[s_pos]) {
30              dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
31              if (!dptr->data[s_pos])
32                      goto out;
33      }
34      /* write only up to the end of this quantum */
35      if (count > quantum - q_pos)
36              count = quantum - q_pos;
37
38      if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
39              retval = -EFAULT;
40              goto out;
41      }
42      *f_pos += count;
43      retval = count;
44
45           /* update the size */
46      if (dev->size < *f_pos)
47              dev->size = *f_pos;
48
49    out:
50      up(&dev->sem);
51      return retval;
52  }
```

readv和writev

Unix系统早就已经支持两个可选的系统调用readv和writev

现在先复习一下系统调用：根据APUE的解释，内核的接口称为系统调用，公用函数库构建在系统调用接口之上，应用程序可以使用公用函数库，也可使用系统调用

read和write系统调用，读取不连续的内存要多次调用read，将数据存入不连续的内存中也需要多次调用write。为了避免避免多次系统调用的开销所以引进了writev和readv

ioctl系统调用

ioctl原型

```
int  ioctl(int fd,unsigned long cmd,...)
第三个参数要依靠第二个参数
```

驱动程序的ioctl方法

```
int ioctl(struct inode *inode,struct file *filp,unsignded int cmd,unsigned long arg)
inode和filp两个指针的值对应用胡空间的描述符fd
arg为附加参数
```

在编写代码之前，需要选择对应不同命令的编号。但是创建命令号的方法内核提供了几个函数。定义号码的新方法使用了4个位字段，如下

1. type,幻数：选择一个号码（已经使用了的在ioctl-number.txt）这个字段有八位宽（_IOC_TYPEBITS）
2. number,序数：顺序编号，8个bit
3. direction：如果cmd涉及到数据的传输方向，则该字段定义数据的传输方向。可以使用的值

- _IOC_NONE没有数据传输
- _ION_READ 读
- _IOC_WRITE 写
- _IOC_READ|WRITE双向传输

4. size ： 锁涉及的用户数据大小

构建命令号的宏<linux/ioctl.h>

- _IO(type,nr):构建无参数的命令编号
- _IOR（type,nr,datatype）；用于构建从驱动程序中读取数据的命令编号
- _IOW（type,nr,datatype）:用于写入数据的命令
- _IOWR（type,nr,fatatype）:用于双向传输

type和number位字段通过参数传入，二size位字段通过对datatype参数取sizeof获得

下面是调用方法

```
/* Use 'k' as magic number */
#define SCULL_IOC_MAGIC  'k'
/* Please use a different 8-bit number in your code */

#define SCULL_IOCRESET    _IO(SCULL_IOC_MAGIC, 0)

/*
 * S means "Set" through a ptr,
 * T means "Tell" directly with the argument value
 * G means "Get": reply by setting through a pointer
 * Q means "Query": response is on the return value
 * X means "eXchange": switch G and S atomically
 * H means "sHift": switch T and Q atomically
```

```
 */
#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC,  1, int)
#define SCULL_IOCSQSET    _IOW(SCULL_IOC_MAGIC,  2, int)
#define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC,   3)
#define SCULL_IOCTQSET    _IO(SCULL_IOC_MAGIC,   4)
#define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC,  5, int)
#define SCULL_IOCGQSET    _IOR(SCULL_IOC_MAGIC,  6, int)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC,   7)
#define SCULL_IOCQQSET    _IO(SCULL_IOC_MAGIC,   8)
#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, int)
#define SCULL_IOCXQSET    _IOWR(SCULL_IOC_MAGIC,10, int)
#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC,  11)
#define SCULL_IOCHQSET    _IO(SCULL_IOC_MAGIC,  12)
```

当调用ioctl的cmd是不合法参数时按照Posix标准返回-ENOTTY

使用ioctl参数

使用附加参数需要注意

1. 如果是整数，直接使用就可以了
2. 如果是指针，必须确保指向的用户空间是合法的

使用access_ok验证地址

```
int access_ok(int type,const void *addr,unsigned long size);
第一个参数应该是VERIFY_READ或VERIFY_WRITE,取决于要执行的动作是读取还是写入用户空间内存区
addr是一个用户空间地址
size是字节数
如果在指定地址既要读取又要写入，则应该使用VERIFY_WRITE，因为它是VERIFY_READ的超集
返回值：1表示成功，0表示失败。如果返回失败，驱动程序通常要返回-EFAULT给调用者
```

在传递单个数据时，避免使用copy_from_user和copy_to_user（速度较慢）。使用put_user(datum,ptr)和__put_user().记得使用access_ok来验证内存的合法地址。从用户空间读:get_user(local,ptr),__get_user(local,ptr)

ioctl的实现

驱动层

```
int scull_ioctl(struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg)
{

    int err = 0, tmp;
    int retval = 0;
```

```
7
8     /*
9      * extract the type and number bitfields, and don't decode
10     * wrong cmds: return ENOTTY (inappropriate ioctl) before access_ok()
11     */
12    if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
13    if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;
14
15    /*
16     * the direction is a bitmask, and VERIFY_WRITE catches R/W
17     * transfers. `Type' is user-oriented, while
18     * access_ok is kernel-oriented, so the concept of "read" and
19     * "write" is reversed
20     */
21    if (_IOC_DIR(cmd) & _IOC_READ)
22            err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
23    else if (_IOC_DIR(cmd) & _IOC_WRITE)
24            err =  !access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd));
25    if (err) return -EFAULT;
26
27    switch(cmd) {
28
29      case SCULL_IOCRESET:
30            scull_quantum = SCULL_QUANTUM;
31            scull_qset = SCULL_QSET;
32            break;
33
34      case SCULL_IOCSQUANTUM: /* Set: arg points to the value */
35            if (! capable (CAP_SYS_ADMIN))
36                    return -EPERM;
37            retval = __get_user(scull_quantum, (int __user *)arg);
38            break;
39
40      case SCULL_IOCTQUANTUM: /* Tell: arg is the value */
41            if (! capable (CAP_SYS_ADMIN))
42                    return -EPERM;
43            scull_quantum = arg;
44            break;
45
```

```
46      case SCULL_IOCGQUANTUM: /* Get: arg is pointer to result */
47            retval = __put_user(scull_quantum, (int __user *)arg);
48            break;
49
50      case SCULL_IOCQQUANTUM: /* Query: return it (it's positive) */
51            return scull_quantum;
52
53      case SCULL_IOCXQUANTUM: /* eXchange: use arg as pointer */
54            if (! capable (CAP_SYS_ADMIN))
55                    return -EPERM;
56            tmp = scull_quantum;
57            retval = __get_user(scull_quantum, (int __user *)arg);
58            if (retval == 0)
59                    retval = __put_user(tmp, (int __user *)arg);
60            break;
61
62      case SCULL_IOCHQUANTUM: /* sHift: like Tell + Query */
63            if (! capable (CAP_SYS_ADMIN))
64                    return -EPERM;
65            tmp = scull_quantum;
66            scull_quantum = arg;
67            return tmp;
68
69      case SCULL_IOCSQSET:
70            if (! capable (CAP_SYS_ADMIN))
71                    return -EPERM;
72            retval = __get_user(scull_qset, (int __user *)arg);
73            break;
74
75      default:  /* redundant, as cmd was checked against MAXNR */
76            return -ENOTTY;
77    }
78    return retval;
79
80  }
81
```

应用层

```
1  int quantum
```

```
2 ioctl(fd,SCULL_IOCSQUANTUM,&quantum);
```