Linux 内核实验报告

实验题目：字符设备驱动程序实验

学号： 200900301236 （辅修号： ）
日期：2012.5.11 班级： 09 软 1 姓名：王添枝
Email：tzwang2012@163.com

实验目的：学习设备驱动程序的组织，学会编写字符设备驱动程序。

硬件环境：
软件环境：ubuntu 10.10
linux 内核:2.6.35.13
gcc:4.4.5

实验步骤：

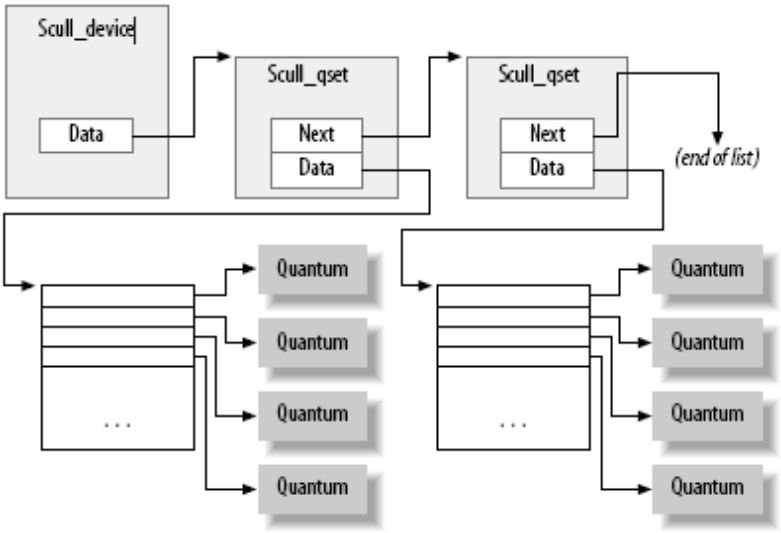
一、实验设计

1.问题 A

原来的程序有限定设备读写长度，要使其不限定长度，可以采用动态分配的方法。数据结构如下：

```
struct scull_qset {
   void **data;
   struct scull_qset *next;
};
struct scull_dev {
   struct scull_qset *data;   /* Pointer to first quantum
set */
   int quantum;              /* the current quantum size */
   int qset;                 /* the current array size */
   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    */
};
```

在 scull,每个设备是一个指针链表，每个都指向一个 scull_dev 结构，scull_dev 结构中，quantum 是量子的大小，qset 是量子集大小。scull_qset 结构中，每个这样的结构默认的量子集大小为 1000，量子大小为 4000，结构如下图所示：



2.问题B

数据结构如下：

```
struct scull_pipe {
      wait_queue_head_t inq,outq;/*read and write queues */
      char *buffer, *end;  /* begin of buf, end of buf */
      int buffersize;     /* used in pointer arithmetic */
      char *rp, *wp;  /* where to read, where to write */
      int nreaders,nwriters;/*number of openings for r/w */
         struct  fasync_struct  *async_queue;/*asynchronous
readers */
     struct semaphore sem; /* mutual exclusion semaphore */
    struct cdev cdev;          /* Char device structure */
};
```

这个设备驱动使用一个设备结构，它包含2个等待队列和一个缓冲。缓冲大小是以常用的方法可配置的（在编译时间，加载时间，或运行时间）。在open时清空缓冲区，在写时如果缓冲区满则等待，直到读进程唤醒它，再写进去；在读时如果缓冲区没有数据则等待，直到写进程唤醒它。

二、调试记录

1.问题A

由于用cp /dev/zero /dev/scull0命令时可能会使系统陷入崩溃状态，因此可以使用dd工具，限制复制的大小。

（1）执行shell脚本:sudo bash ./scull_load,安装模块和相应的字符设备文件.

（2）执行：sudo dd if=/dev/zero of=/dev/scull0 bs=1024 count=786432，其中bs表示一块的大小，count表示复制的块数，此处表示复制768M。

（3）用系统监视器观察内存使用情况，会发现内存使用率迅速上升到很大的值。

2.问题B

（1）执行读测试程序：sudo ./test 1,此处1表示读测试程序。
（2）执行写测试程序：sudo ./test 0,此处0（非1）表示写测试程序。
（3）在写测试程序中输入字符后可以看到读测试程序读出了相应的字符。

三、结论分析与体会

本次实验主要参照《linux设备驱动程序》的第三章字符设备驱动和第六章6.2阻塞IO来实现，要编写设备驱动程序是一件比较麻烦的事情，很多问题需要考虑，如并发和竞争等情况。

通过本次字符设备驱动程序实验，对编写字符设备驱动的步骤有了一定的了解，从中学到了不少的知识。

程序完整源代码：

1.scull.h

```
#ifndef _SCULL_H_
#define _SCULL_H_

#include <linux/ioctl.h> /* needed for the _IOW etc stuff used later */

/*
 * Macros to help debugging
 */

#undef PDEBUG             /* undef it, just in case */
#ifdef SCULL_DEBUG
#  ifdef __KERNEL__
     /* This one if debugging is on, and kernel space */
#    define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt, ## args)
#  else
     /* This one for user space */
#    define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#  endif
#else
#  define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif

#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* nothing: it's a placeholder */

#ifndef SCULL_MAJOR
#define SCULL_MAJOR 0   /* dynamic major by default */
#endif

#ifndef SCULL_NR_DEVS
#define SCULL_NR_DEVS 4    /* scull0 through scull3 */
#endif
```

```
#ifndef SCULL_P_NR_DEVS
#define SCULL_P_NR_DEVS 4  /* scullpipe0 through scullpipe3 */
#endif

/*
 * The bare device is a variable-length region of memory.
 * Use a linked list of indirect blocks.
 *
 * "scull_dev->data" points to an array of pointers, each
 * pointer refers to a memory area of SCULL_QUANTUM bytes.
 *
 * The array (quantum-set) is SCULL_QSET long.
 */
#ifndef SCULL_QUANTUM
#define SCULL_QUANTUM 4000
#endif

#ifndef SCULL_QSET
#define SCULL_QSET    1000
#endif

/*
 * The pipe device is a simple circular buffer. Here its default size
 */
#ifndef SCULL_P_BUFFER
#define SCULL_P_BUFFER 4000
#endif

/*
 * Representation of scull quantum sets.
 */
struct scull_qset {
    void **data;
    struct scull_qset *next;
};
struct scull_dev {
    struct scull_qset *data;  /* Pointer to first quantum set */
    int quantum;              /* the current quantum size */
    int qset;                 /* the current array size */
    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       */
```

```
};
/*
 * Split minors in two parts
 */
#define TYPE(minor)  (((minor) >> 4) & 0xf)   /* high nibble */
#define NUM(minor)   ((minor) & 0xf)      /* low  nibble */

/*
 * The different configurable parameters
 */
extern int scull_major;     /* main.c */
extern int scull_nr_devs;
extern int scull_quantum;
extern int scull_qset;
extern int scull_p_buffer;   /* pipe.c */
/*
 * Prototypes for shared functions
 */

int     scull_p_init(dev_t dev);
void    scull_p_cleanup(void);
int     scull_access_init(dev_t dev);
void    scull_access_cleanup(void);
int     scull_trim(struct scull_dev *dev);
ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
                loff_t *f_pos);
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count,
                 loff_t *f_pos);
loff_t  scull_llseek(struct file *filp, loff_t off, int whence);
int     scull_ioctl(struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg);
/*
 * Ioctl definitions
 */

/* 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)
/*
 * The other entities only have "Tell" and "Query", because they're
 * not printed in the book, and there's no need to have all six.
 * (The previous stuff was only there to show different ways to do it.
 */
#define SCULL_P_IOCTSIZE _IO(SCULL_IOC_MAGIC,   13)
#define SCULL_P_IOCQSIZE _IO(SCULL_IOC_MAGIC,   14)
/* ... more to come */
#define SCULL_IOC_MAXNR 14
#endif /* _SCULL_H_ */
```

2.main.c

```
//#include <linux/config.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
#include <linux/kernel.h>   /* printk() */
#include <linux/slab.h>     /* kmalloc() */
#include <linux/fs.h>       /* everything... */
#include <linux/errno.h>    /* error codes */
#include <linux/types.h>    /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h>    /* O_ACCMODE */
#include <linux/seq_file.h>
#include <linux/cdev.h>
#include <asm/system.h>     /* cli(), *_flags */
#include <asm/uaccess.h>    /* copy_*_user */
#include "scull.h"      /* local definitions */
```

```
/*
 * Our parameters which can be set at load time.
 */
int scull_major =   SCULL_MAJOR;
int scull_minor =   0;
int scull_nr_devs = SCULL_NR_DEVS;    /* number of bare scull devices */
int scull_quantum = SCULL_QUANTUM;
int scull_qset =    SCULL_QSET;
module_param(scull_major, int, S_IRUGO);
module_param(scull_minor, int, S_IRUGO);
module_param(scull_nr_devs, int, S_IRUGO);
module_param(scull_quantum, int, S_IRUGO);
module_param(scull_qset, int, S_IRUGO);

MODULE_AUTHOR("Alessandro Rubini, Jonathan Corbet");
MODULE_LICENSE("Dual BSD/GPL");

struct scull_dev *scull_devices; /* allocated in scull_init_module */
/*
 * Empty out the scull device; must be called with the device
 * semaphore held.
 */
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;
}
```

```
#ifdef SCULL_DEBUG /* use proc only if debugging */
/*
 * The proc filesystem: function to read and entry
 */
int scull_read_procmem(char *buf, char **start, off_t offset,
                   int count, int *eof, void *data)
{
     int i, j, len = 0;
     int limit = count - 80; /* Don't print more than this */

     for (i = 0; i < scull_nr_devs && len <= limit; i++) {
          struct scull_dev *d = &scull_devices[i];
          struct scull_qset *qs = d->data;
          if (down_interruptible(&d->sem))
               return -ERESTARTSYS;
          len += sprintf(buf+len,"\nDevice %i: qset %i, q %i, sz %li\n",
                    i, d->qset, d->quantum, d->size);
          for (; qs && len <= limit; qs = qs->next) { /* scan the list */
               len += sprintf(buf + len, "  item at %p, qset at %p\n",
                         qs, qs->data);
               if (qs->data && !qs->next) /* dump only the last item */
                    for (j = 0; j < d->qset; j++) {
                         if (qs->data[j])
                              len += sprintf(buf + len,
                                        "    % 4i: %8p\n",
                                        j, qs->data[j]);
                    }
          }
          up(&scull_devices[i].sem);
     }
     *eof = 1;
     return len;
}
/*
 * For now, the seq_file implementation will exist in parallel.  The
 * older read_procmem function should maybe go away, though.
 */

/*
 * Here are our sequence iteration methods.  Our "position" is
 * simply the device number.
 */
static void *scull_seq_start(struct seq_file *s, loff_t *pos)
{
```

```
    if (*pos >= scull_nr_devs)
        return NULL;   /* No more to read */
    return scull_devices + *pos;
}

static void *scull_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
    (*pos)++;
    if (*pos >= scull_nr_devs)
        return NULL;
    return scull_devices + *pos;
}
static void scull_seq_stop(struct seq_file *s, void *v)
{
    /* Actually, there's nothing to do here */
}
static int scull_seq_show(struct seq_file *s, void *v)
{
    struct scull_dev *dev = (struct scull_dev *) v;
    struct scull_qset *d;
    int i;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    seq_printf(s, "\nDevice %i: qset %i, q %i, sz %li\n",
              (int) (dev - scull_devices), dev->qset,
              dev->quantum, dev->size);
    for (d = dev->data; d; d = d->next) { /* scan the list */
        seq_printf(s, "  item at %p, qset at %p\n", d, d->data);
        if (d->data && !d->next) /* dump only the last item */
            for (i = 0; i < dev->qset; i++) {
                if (d->data[i])
                    seq_printf(s, "    % 4i: %8p\n",
                              i, d->data[i]);
            }
    }
    up(&dev->sem);
    return 0;
}

/*
 * Tie the sequence operators up.
 */
static struct seq_operations scull_seq_ops = {
```

```
    .start = scull_seq_start,
    .next  = scull_seq_next,
    .stop  = scull_seq_stop,
    .show  = scull_seq_show
};

/*
 * Now to implement the /proc file we need only make an open
 * method which sets up the sequence operators.
 */
static int scull_proc_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &scull_seq_ops);
}

/*
 * Create a set of file operations for our proc file.
 */
static struct file_operations scull_proc_ops = {
    .owner   = THIS_MODULE,
    .open    = scull_proc_open,
    .read    = seq_read,
    .llseek  = seq_lseek,
    .release = seq_release
};



/*
 * Actually create (and remove) the /proc file(s).
 */

static void scull_create_proc(void)
{
    struct proc_dir_entry *entry;
    create_proc_read_entry("scullmem", 0 /* default mode */,
              NULL /* parent dir */, scull_read_procmem,
              NULL /* client data */);
    entry = create_proc_entry("scullseq", 0, NULL);
    if (entry)
         entry->proc_fops = &scull_proc_ops;
}

static void scull_remove_proc(void)
{
```

```
    /* no problem if it was not registered */
    remove_proc_entry("scullmem", NULL /* parent dir */);
    remove_proc_entry("scullseq", NULL);
}


#endif /* SCULL_DEBUG */
/*
 * Open and close
 */

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 = dev; /* 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 */
}

int scull_release(struct inode *inode, struct file *filp)
{
    return 0;
}
/*
 * Follow the list
 */
struct scull_qset *scull_follow(struct scull_dev *dev, int n)
{
    struct scull_qset *qs = dev->data;

        /* Allocate first qset explicitly if need be */
    if (! qs) {
        qs = dev->data = kmalloc(sizeof(struct scull_qset), GFP_KERNEL);
        if (qs == NULL)
            return NULL;  /* Never mind */
```

```
        memset(qs, 0, sizeof(struct scull_qset));
    }

    /* Then follow the list */
    while (n--) {
        if (!qs->next) {
            qs->next = kmalloc(sizeof(struct scull_qset), GFP_KERNEL);
            if (qs->next == NULL)
                return NULL;  /* Never mind */
            memset(qs->next, 0, sizeof(struct scull_qset));
        }
        qs = qs->next;
        continue;
    }
    return qs;
}

/*
 * Data management: read and write
 */

ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
                loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;    /* the first listitem */
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset; /* how many bytes in the listitem */
    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;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;

    /* 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)
        count = quantum - q_pos;

    if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;

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

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) {
```

```
            dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
            if (!dptr->data)
                goto out;
            memset(dptr->data, 0, qset * sizeof(char *));
        }
        if (!dptr->data[s_pos]) {
            dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
            if (!dptr->data[s_pos])
                goto out;
        }
        /* write only up to the end of this quantum */
        if (count > quantum - q_pos)
            count = quantum - q_pos;

        if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
            retval = -EFAULT;
            goto out;
        }
        *f_pos += count;
        retval = count;

           /* update the size */
        if (dev->size < *f_pos)
            dev->size = *f_pos;

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

/*
 * The ioctl() implementation
 */

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

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

        /*
         * extract the type and number bitfields, and don't decode
         * wrong cmds: return ENOTTY (inappropriate ioctl) before access_ok()
```

```
 */
if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

/*
 * the direction is a bitmask, and VERIFY_WRITE catches R/W
 * transfers. `Type' is user-oriented, while
 * access_ok is kernel-oriented, so the concept of "read" and
 * "write" is reversed
 */
if (_IOC_DIR(cmd) & _IOC_READ)
     err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
     err =  !access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd));
if (err) return -EFAULT;

switch(cmd) {

  case SCULL_IOCRESET:
     scull_quantum = SCULL_QUANTUM;
     scull_qset = SCULL_QSET;
     break;

  case SCULL_IOCSQUANTUM: /* Set: arg points to the value */
     if (! capable (CAP_SYS_ADMIN))
          return -EPERM;
     retval = __get_user(scull_quantum, (int __user *)arg);
     break;

  case SCULL_IOCTQUANTUM: /* Tell: arg is the value */
     if (! capable (CAP_SYS_ADMIN))
          return -EPERM;
     scull_quantum = arg;
     break;

  case SCULL_IOCGQUANTUM: /* Get: arg is pointer to result */
     retval = __put_user(scull_quantum, (int __user *)arg);
     break;

  case SCULL_IOCQQUANTUM: /* Query: return it (it's positive) */
     return scull_quantum;

  case SCULL_IOCXQUANTUM: /* eXchange: use arg as pointer */
     if (! capable (CAP_SYS_ADMIN))
```

```
        return -EPERM;
    tmp = scull_quantum;
    retval = __get_user(scull_quantum, (int __user *)arg);
    if (retval == 0)
        retval = __put_user(tmp, (int __user *)arg);
    break;

case SCULL_IOCHQUANTUM: /* sHift: like Tell + Query */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    tmp = scull_quantum;
    scull_quantum = arg;
    return tmp;

case SCULL_IOCSQSET:
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    retval = __get_user(scull_qset, (int __user *)arg);
    break;

case SCULL_IOCTQSET:
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    scull_qset = arg;
    break;

case SCULL_IOCGQSET:
    retval = __put_user(scull_qset, (int __user *)arg);
    break;

case SCULL_IOCQQSET:
    return scull_qset;

case SCULL_IOCXQSET:
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    tmp = scull_qset;
    retval = __get_user(scull_qset, (int __user *)arg);
    if (retval == 0)
        retval = put_user(tmp, (int __user *)arg);
    break;

case SCULL_IOCHQSET:
    if (! capable (CAP_SYS_ADMIN))
```

```
            return -EPERM;
          tmp = scull_qset;
          scull_qset = arg;
          return tmp;

         /*
          * The following two change the buffer size for scullpipe.
          * The scullpipe device uses this same ioctl method, just to
          * write less code. Actually, it's the same driver, isn't it?
          */

        case SCULL_P_IOCTSIZE:
          scull_p_buffer = arg;
          break;

        case SCULL_P_IOCQSIZE:
          return scull_p_buffer;


        default:  /* redundant, as cmd was checked against MAXNR */
          return -ENOTTY;
      }
      return retval;

}



/*
 * The "extended" operations -- only seek
 */

loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
      struct scull_dev *dev = filp->private_data;
      loff_t newpos;

      switch(whence) {
        case 0: /* SEEK_SET */
          newpos = off;
          break;

        case 1: /* SEEK_CUR */
          newpos = filp->f_pos + off;
```

```
            break;

        case 2: /* SEEK_END */
            newpos = dev->size + off;
            break;

        default: /* can't happen */
            return -EINVAL;
    }
    if (newpos < 0) return -EINVAL;
    filp->f_pos = newpos;
    return newpos;
}



struct file_operations scull_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .ioctl =    scull_ioctl,
    .open =     scull_open,
    .release =  scull_release,
};

/*
 * Finally, the module stuff
 */

/*
 * The cleanup function is used to handle initialization failures as well.
 * Thefore, it must be careful to work correctly even if some of the items
 * have not been initialized
 */
void scull_cleanup_module(void)
{
    int i;
    dev_t devno = MKDEV(scull_major, scull_minor);

    /* Get rid of our char dev entries */
    if (scull_devices) {
        for (i = 0; i < scull_nr_devs; i++) {
            scull_trim(scull_devices + i);
```

```
                cdev_del(&scull_devices[i].cdev);
            }
            kfree(scull_devices);
        }

#ifdef SCULL_DEBUG /* use proc only if debugging */
        scull_remove_proc();
#endif

        /* cleanup_module is never called if registering failed */
        unregister_chrdev_region(devno, scull_nr_devs);

        /* and call the cleanup functions for friend devices */
        scull_p_cleanup();
        scull_access_cleanup();

}



/*
 * Set up the char_dev structure for this device.
 */
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
        int err, devno = MKDEV(scull_major, scull_minor + index);

        cdev_init(&dev->cdev, &scull_fops);
        dev->cdev.owner = THIS_MODULE;
        dev->cdev.ops = &scull_fops;
        err = cdev_add (&dev->cdev, devno, 1);
        /* Fail gracefully if need be */
        if (err)
                printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}



int scull_init_module(void)
{
        int result, i;
        dev_t dev = 0;

/*
 * Get a range of minor numbers to work with, asking for a dynamic
 * major unless directed otherwise at load time.
```

```
 */
    if (scull_major) {
         dev = MKDEV(scull_major, scull_minor);
         result = register_chrdev_region(dev, scull_nr_devs, "scull");
    } else {
         result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs,
                   "scull");
         scull_major = MAJOR(dev);
    }
    if (result < 0) {
         printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
         return result;
    }

       /*
     * allocate the devices -- we can't have them static, as the number
     * can be specified at load time
     */
    scull_devices = kmalloc(scull_nr_devs * sizeof(struct scull_dev), GFP_KERNEL);
    if (!scull_devices) {
         result = -ENOMEM;
         goto fail;  /* Make this more graceful */
    }
    memset(scull_devices, 0, scull_nr_devs * sizeof(struct scull_dev));

       /* Initialize each device. */
    for (i = 0; i < scull_nr_devs; i++) {
         scull_devices[i].quantum = scull_quantum;
         scull_devices[i].qset = scull_qset;
         init_MUTEX(&scull_devices[i].sem);
         scull_setup_cdev(&scull_devices[i], i);
    }

       /* At this point call the init function for any friend device */
    dev = MKDEV(scull_major, scull_minor + scull_nr_devs);
    dev += scull_p_init(dev);
    dev += scull_access_init(dev);

#ifdef SCULL_DEBUG /* only when debugging */
    scull_create_proc();
#endif

    return 0; /* succeed */
```

```
  fail:
     scull_cleanup_module();
     return result;
}

module_init(scull_init_module);
module_exit(scull_cleanup_module);
```

3.pipe.c

```
#include <linux/module.h>
#include <linux/moduleparam.h>

#include <linux/kernel.h>   /* printk(), min() */
#include <linux/slab.h>     /* kmalloc() */
#include <linux/fs.h>       /* everything... */
#include <linux/proc_fs.h>
#include <linux/errno.h>    /* error codes */
#include <linux/types.h>    /* size_t */
#include <linux/fcntl.h>
#include <linux/poll.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
#include <linux/sched.h>

#include "scull.h"     /* local definitions */

struct scull_pipe {
        wait_queue_head_t inq, outq;       /* read and write queues */
        char *buffer, *end;                /* begin of buf, end of buf */
        int buffersize;                    /* used in pointer arithmetic */
        char *rp, *wp;                     /* where to read, where to write */
        int nreaders, nwriters;            /* number of openings for r/w */
        struct fasync_struct *async_queue; /* asynchronous readers */
        struct semaphore sem;              /* mutual exclusion semaphore */
        struct cdev cdev;                  /* Char device structure */
};

/* parameters */
static int scull_p_nr_devs = SCULL_P_NR_DEVS;  /* number of pipe devices */
int scull_p_buffer =  SCULL_P_BUFFER;/* buffer size */
dev_t scull_p_devno;             /* Our first device number */

module_param(scull_p_nr_devs, int, 0);    /* FIXME check perms */
module_param(scull_p_buffer, int, 0);
```

```
static struct scull_pipe *scull_p_devices;

static int scull_p_fasync(int fd, struct file *filp, int mode);
static int spacefree(struct scull_pipe *dev);
/*
 * Open and close
 */
static int scull_p_open(struct inode *inode, struct file *filp)
{
     struct scull_pipe *dev;

     dev = container_of(inode->i_cdev, struct scull_pipe, cdev);
     filp->private_data = dev;

     if (down_interruptible(&dev->sem))
          return -ERESTARTSYS;
     if (!dev->buffer) {
          /* allocate the buffer */
          dev->buffer = kmalloc(scull_p_buffer, GFP_KERNEL);
          if (!dev->buffer) {
               up(&dev->sem);
               return -ENOMEM;
          }
     }
     dev->buffersize = scull_p_buffer;
     dev->end = dev->buffer + dev->buffersize;
     dev->rp = dev->wp = dev->buffer; /* rd and wr from the beginning */

     /* use f_mode,not  f_flags: it's cleaner (fs/open.c tells why) */
     if (filp->f_mode & FMODE_READ)
          dev->nreaders++;
     if (filp->f_mode & FMODE_WRITE)
          dev->nwriters++;
     up(&dev->sem);

     return nonseekable_open(inode, filp);
}
static int scull_p_release(struct inode *inode, struct file *filp)
{
     struct scull_pipe *dev = filp->private_data;

     /* remove this filp from the asynchronously notified filp's */
     scull_p_fasync(-1, filp, 0);
```

```
    down(&dev->sem);
    if (filp->f_mode & FMODE_READ)
        dev->nreaders--;
    if (filp->f_mode & FMODE_WRITE)
        dev->nwriters--;
    if (dev->nreaders + dev->nwriters == 0) {
        kfree(dev->buffer);
        dev->buffer = NULL; /* the other fields are not checked on open */
    }
    up(&dev->sem);
    return 0;
}


/*
 * Data management: read and write
 */

static ssize_t scull_p_read (struct file *filp, char __user *buf, size_t count,
                loff_t *f_pos)
{
    struct scull_pipe *dev = filp->private_data;

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

    while (dev->rp == dev->wp) { /* nothing to read */
        up(&dev->sem); /* release the lock */
        if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
        PDEBUG("\"%s\" reading: going to sleep\n", current->comm);
        if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
            return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
        /* otherwise loop, but first reacquire the lock */
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    }
    /* ok, data is there, return something */
    if (dev->wp > dev->rp)
        count = min(count, (size_t)(dev->wp - dev->rp));
    else /* the write pointer has wrapped, return data up to dev->end */
        count = min(count, (size_t)(dev->end - dev->rp));
    if (copy_to_user(buf, dev->rp, count)) {
        up (&dev->sem);
```

```
          return -EFAULT;
     }
     dev->rp += count;
     if (dev->rp == dev->end)
          dev->rp = dev->buffer; /* wrapped */
     up (&dev->sem);

     /* finally, awake any writers and return */
     wake_up_interruptible(&dev->outq);
     PDEBUG("\"%s\" did read %li bytes\n",current->comm, (long)count);
     return count;
}

/* Wait for space for writing; caller must hold device semaphore.  On
 * error the semaphore will be released before returning. */
static int scull_getwritespace(struct scull_pipe *dev, struct file *filp)
{
     while (spacefree(dev) == 0) { /* full */
          DEFINE_WAIT(wait);

          up(&dev->sem);
          if (filp->f_flags & O_NONBLOCK)
               return -EAGAIN;
          PDEBUG("\"%s\" writing: going to sleep\n",current->comm);
          prepare_to_wait(&dev->outq, &wait, TASK_INTERRUPTIBLE);
          if (spacefree(dev) == 0)
               schedule();
          finish_wait(&dev->outq, &wait);
          if (signal_pending(current))
               return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
          if (down_interruptible(&dev->sem))
               return -ERESTARTSYS;
     }
     return 0;
}

/* How much space is free? */
static int spacefree(struct scull_pipe *dev)
{
     if (dev->rp == dev->wp)
          return dev->buffersize - 1;
     return ((dev->rp + dev->buffersize - dev->wp) % dev->buffersize) - 1;
}
```

```
static ssize_t scull_p_write(struct file *filp, const char __user *buf, size_t count,
                loff_t *f_pos)
{
     struct scull_pipe *dev = filp->private_data;
     int result;

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

     /* Make sure there's space to write */
     result = scull_getwritespace(dev, filp);
     if (result)
          return result; /* scull_getwritespace called up(&dev->sem) */

     /* ok, space is there, accept something */
     count = min(count, (size_t)spacefree(dev));
     if (dev->wp >= dev->rp)
          count = min(count, (size_t)(dev->end - dev->wp)); /* to end-of-buf */
     else /* the write pointer has wrapped, fill up to rp-1 */
          count = min(count, (size_t)(dev->rp - dev->wp - 1));
     PDEBUG("Going to accept %li bytes to %p from %p\n", (long)count, dev->wp, buf);
     if (copy_from_user(dev->wp, buf, count)) {
          up (&dev->sem);
          return -EFAULT;
     }
     dev->wp += count;
     if (dev->wp == dev->end)
          dev->wp = dev->buffer; /* wrapped */
     up(&dev->sem);

     /* finally, awake any reader */
     wake_up_interruptible(&dev->inq);  /* blocked in read() and select() */

     /* and signal asynchronous readers, explained late in chapter 5 */
     if (dev->async_queue)
          kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
     PDEBUG("\"%s\" did write %li bytes\n",current->comm, (long)count);
     return count;
}

static unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
     struct scull_pipe *dev = filp->private_data;
     unsigned int mask = 0;
```

```
    /*
     * The buffer is circular; it is considered full
     * if "wp" is right behind "rp" and empty if the
     * two are equal.
     */
    down(&dev->sem);
    poll_wait(filp, &dev->inq,  wait);
    poll_wait(filp, &dev->outq, wait);
    if (dev->rp != dev->wp)
        mask |= POLLIN | POLLRDNORM;    /* readable */
    if (spacefree(dev))
        mask |= POLLOUT | POLLWRNORM;   /* writable */
    up(&dev->sem);
    return mask;
}




static int scull_p_fasync(int fd, struct file *filp, int mode)
{
    struct scull_pipe *dev = filp->private_data;

    return fasync_helper(fd, filp, mode, &dev->async_queue);
}



/* FIXME this should use seq_file */
#ifdef SCULL_DEBUG
static void scullp_proc_offset(char *buf, char **start, off_t *offset, int *len)
{
    if (*offset == 0)
        return;
    if (*offset >= *len) { /* Not there yet */
        *offset -= *len;
        *len = 0;
    }
    else {            /* We're into the interesting stuff now */
        *start = buf + *offset;
        *offset = 0;
    }
```

```
}


static int scull_read_p_mem(char *buf, char **start, off_t offset, int count,
         int *eof, void *data)
{
     int i, len;
     struct scull_pipe *p;

#define LIMIT (PAGE_SIZE-200)    /* don't print any more after this size */
     *start = buf;
     len = sprintf(buf, "Default buffersize is %i\n", scull_p_buffer);
     for(i = 0; i<scull_p_nr_devs && len <= LIMIT; i++) {
          p = &scull_p_devices[i];
          if (down_interruptible(&p->sem))
               return -ERESTARTSYS;
          len += sprintf(buf+len, "\nDevice %i: %p\n", i, p);
/*        len += sprintf(buf+len, "   Queues: %p %p\n", p->inq, p->outq);*/
          len += sprintf(buf+len, "   Buffer: %p to %p (%i bytes)\n", p->buffer, p->end, p-
>buffersize);
          len += sprintf(buf+len, "   rp %p   wp %p\n", p->rp, p->wp);
          len += sprintf(buf+len, "   readers %i   writers %i\n", p->nreaders, p->nwriters);
          up(&p->sem);
          scullp_proc_offset(buf, start, &offset, &len);
     }
     *eof = (len <= LIMIT);
     return len;
}


#endif



/*
 * The file operations for the pipe device
 * (some are overlayed with bare scull)
 */
struct file_operations scull_pipe_fops = {
     .owner =  THIS_MODULE,
     .llseek =no_llseek,
     .read =       scull_p_read,
     .write =  scull_p_write,
     .poll =       scull_p_poll,
```

```
    .ioctl =  scull_ioctl,
    .open =       scull_p_open,
    .release =    scull_p_release,
    .fasync =scull_p_fasync,
};


/*
 * Set up a cdev entry.
 */
static void scull_p_setup_cdev(struct scull_pipe *dev, int index)
{
    int err, devno = scull_p_devno + index;

    cdev_init(&dev->cdev, &scull_pipe_fops);
    dev->cdev.owner = THIS_MODULE;
    err = cdev_add (&dev->cdev, devno, 1);
    /* Fail gracefully if need be */
    if (err)
        printk(KERN_NOTICE "Error %d adding scullpipe%d", err, index);
}



/*
 * Initialize the pipe devs; return how many we did.
 */
int scull_p_init(dev_t firstdev)
{
    int i, result;

    result = register_chrdev_region(firstdev, scull_p_nr_devs, "scullp");
    if (result < 0) {
        printk(KERN_NOTICE "Unable to get scullp region, error %d\n", result);
        return 0;
    }
    scull_p_devno = firstdev;
    scull_p_devices = kmalloc(scull_p_nr_devs * sizeof(struct scull_pipe), GFP_KERNEL);
    if (scull_p_devices == NULL) {
        unregister_chrdev_region(firstdev, scull_p_nr_devs);
        return 0;
    }
    memset(scull_p_devices, 0, scull_p_nr_devs * sizeof(struct scull_pipe));
    for (i = 0; i < scull_p_nr_devs; i++) {
```

```
            init_waitqueue_head(&(scull_p_devices[i].inq));
            init_waitqueue_head(&(scull_p_devices[i].outq));
            init_MUTEX(&scull_p_devices[i].sem);
            scull_p_setup_cdev(scull_p_devices + i, i);
        }
#ifdef SCULL_DEBUG
        create_proc_read_entry("scullpipe", 0, NULL, scull_read_p_mem, NULL);
#endif
        return scull_p_nr_devs;
}

/*
 * This is called by cleanup_module or on failure.
 * It is required to never fail, even if nothing was initialized first
 */
void scull_p_cleanup(void)
{
        int i;

#ifdef SCULL_DEBUG
        remove_proc_entry("scullpipe", NULL);
#endif

        if (!scull_p_devices)
            return; /* nothing else to release */

        for (i = 0; i < scull_p_nr_devs; i++) {
            cdev_del(&scull_p_devices[i].cdev);
            kfree(scull_p_devices[i].buffer);
        }
        kfree(scull_p_devices);
        unregister_chrdev_region(scull_p_devno, scull_p_nr_devs);
        scull_p_devices = NULL; /* pedantic */
}
```

4.test.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
main(int argc,char *argv[])
{
        int fd ;
        char num;
```

```
int jieshou;
jieshou=atoi(argv[1]);
//int ptr_write=0;
//int ptr_read;
if(jieshou==1)
{
        fd= open("/dev/scullpipe",O_RDWR);
        int jishu=0;
    while(1)//send data
    {
         //printf("1\n");
         if (fd != -1 )
         {
                 if(read(fd, &num, sizeof(char)))
                   printf("The scull is %c\n", num);
                 /**int i=0;
                 for(i=0;i<=10;i++)
                 {
                 //sleep(0);
                 //printf("%d\n",i);
                 }*/
         }
         else
         {
              printf("Device open failure\n");
         }
    }
    close(fd);
}
else
{
        fd = open("/dev/scullpipe",O_RDWR|O_APPEND);
        int jishu=0;
        while(1)//receive data
        {
         if (fd != -1 )
         {
                 //read(fd, &num, sizeof(char));
              //printf("The scull is %d\n", num);
              //printf("Please input the num written to scull\n");
              scanf("%c", &num);
              write(fd, &num, sizeof(char));
              //num=0;
              //read(fd, &num, sizeof(char));
```

```
                //printf("The scull is %d\n", num);
            }
            else
            {
                printf("Device open failure\n");
            }
        }
        close(fd);
    }
}
```

参考材料

linux 设备驱动程序
linux 内核设计与实现