1. **ioctl接口**

应用空间执行ioctl设置文件属性

驱动程序除了提供读写操作外，通常也提供ioctl接口用于应用程序控制设备行为

用户空间 ioctl 系统调用有下面的原型:

int ioctl(int fd, unsigned long cmd, ...);

Cmd：执行的命令编号

驱动方法ioctl原型:

int (\*ioctl) (struct inode \*inode, struct file \*filp, unsigned int cmd,

unsigned long arg);

inode 和 filp 指针是对应应用程序传递的文件描述符 fd 的值

cmd 执行的命令编号

arg 执行的命令需要的参数

**选择命令编号**

每个设备都有ioctl，这样每个设备都会定义自己的命令编号，即使这样，我们还是希望命令编号在系统中是唯一的

ioctl命令号为32位，分为4个段

type

类型，在整个驱动中. 这个成员是 8 位宽(\_IOC\_TYPEBITS).

number

序号. 它是 8 位(\_IOC\_NRBITS)宽.

direction

数据传送的方向,如果这个特殊的命令涉及数据传送. 可能的值是

\_IOC\_NONE(没有数据传输),

\_IOC\_READ,

\_IOC\_WRITE

\_IOC\_READ|\_IOC\_WRITE (数据在 2 个方向被传送).

size

涉及到的用户数据的大小.

命令编号有关的宏，头文件头文件 <asm/ioctl.h>, 它包含在 <linux/ioctl.h> 中

创建命令编号：

\_IO(type,nr)：定义给没有参数的命令

\_IOR(type, nre, datatype)：定义从驱动中读数据的命令，size为datatype的大小

\_IOW(type,nr,datatype)：定义向驱动中写数据的命令，size为datatype的大小

\_IOWR(type,nr,datatype)：定义双向传输数据的命令，size为datatype的大小

type 和 number 成员作为参数被传递, size 成员通过应用 sizeof 到 datatype 参数而得到.

解码编号:

\_IOC\_DIR(nr),

\_IOC\_TYPE(nr),

\_IOC\_NR(nr),

\_IOC\_SIZE(nr).

Nr为命令编号

示例：

#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)

#define SCULL\_IOC\_MAXNR 14

**返回值**

如果试图执行一个不存在的命令，则应返回-ENOTTY

**预定义的命令**

内核为所有文件预定义了以下几个命令，在选择命令编号时，不要使用如下编号

FIOCLEX

使文件描述符被关闭, 当调用进程执行一个新程序时.

FIONCLEX

FIOQSIZE

这个命令返回一个文件或者目录的大小;

FIONBIO

**权限检查**

在执行某些命令时，可能我们并不想让每一个用户都能执行，这是可以做以下权限的检查

权限定义在<linux/capability.h> 中

如下几个示例权限：

CAP\_DAC\_OVERRIDE

这个能力来推翻在文件和目录上的存取的限制(数据存取控制, 或者 DAC).

CAP\_NET\_ADMIN

进行网络管理任务的能力, 包括那些能够影响网络接口的.

CAP\_SYS\_MODULE

加载或去除内核模块的能力.

CAP\_SYS\_RAWIO

进行 "raw" I/O 操作的能力. 例子包括存取设备端口或者直接和 USB 设备通讯.

CAP\_SYS\_ADMIN

一个捕获-全部的能力, 提供对许多系统管理操作的存取.

CAP\_SYS\_TTY\_CONFIG

进行 tty 配置任务的能力.

通过 capable 函数来进行检查(定义在 <linux/sched.h>):

int capable(int capability);

示例：

if (!capable(CAP\_SYS\_ADMIN))

    return -EPERM;

**Ioctl实现示例**

int scull\_ioctl(struct inode \*inode, struct file \*filp, unsigned int cmd, unsigned long arg)

{

    switch (cmd)

    {

    // 执行 SCULL\_IOCRESET 命令，这里 SCULL\_IOCRESET 命令会重置 scull\_quantum、scull\_qset的值

    case SCULL\_IOCRESET:

        scull\_quantum = SCULL\_QUANTUM;

        scull\_qset = SCULL\_QSET;

        break;

    case SCULL\_IOCSQUANTUM:

        // 进行权限检查

        if (!capable(CAP\_SYS\_ADMIN))

            return -EPERM;

        retval = \_\_get\_user(scull\_quantum, (int \_\_user \*)arg);  // 从用户空间读取参数设置scull\_quantum

        break;

    case SCULL\_IOCTQUANTUM:

        if (!capable(CAP\_SYS\_ADMIN))

            return -EPERM;

        scull\_quantum = arg;

        break;

    case SCULL\_IOCGQUANTUM:

        retval = \_\_put\_user(scull\_quantum, (int \_\_user \*)arg);  // 向用户空间写入 scull\_quantum 的值

        break;

    case SCULL\_IOCQQUANTUM:

        return scull\_quantum;

    case SCULL\_IOCXQUANTUM:

        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:

        if (!capable(CAP\_SYS\_ADMIN))

            return -EPERM;

        tmp = scull\_quantum;

        scull\_quantum = arg;

        return tmp;

    default:

        return -ENOTTY;     // 无效命令返回-ENOTTY

}

    return retval;

}

1. **阻塞I/O**

**等待队列**

等待队列是一个进程链表，包含等他某个事件的所有进程

头文件在<linux/wait.h>，wait\_queue\_head\_t为等待队列头

宏定义和初始化等待队列：

DECLARE\_WAIT\_QUEUE\_HEAD(name);

或者动态定义和初始化等待队列：

wait\_queue\_head\_t my\_queue;

init\_waitqueue\_head(&my\_queue);

**简单的睡眠**

等待宏

wait\_event(queue, condition)

wait\_event\_interruptible(queue, condition)

wait\_event\_timeout(queue, condition, timeout)

wait\_event\_interruptible\_timeout(queue, condition, timeout)

唤醒函数

void wake\_up(wait\_queue\_head\_t \*queue);

void wake\_up\_interruptible(wait\_queue\_head\_t \*queue);

Queue：等待队列

Condition：条件

其中interruptible为可中断版本

示例：

static DECLARE\_WAIT\_QUEUE\_HEAD(wq);

static int flag = 0;

ssize\_t sleepy\_read(struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*pos)

{

    // 等待 flag != 0

    wait\_event\_interruptible(wq, flag != 0);

    flag = 0;

    printk(KERN\_DEBUG "awoken %i (%s)\n", current->pid,

           current->comm);

    return 0; /\* EOF \*/

}

ssize\_t sleepy\_write(struct file \*filp, const char \_\_user \*buf, size\_t count, loff\_t \*pos)

{

    // 设置 flag 并唤醒 wq 等待队列上的进程

    flag = 1;

    wake\_up\_interruptible(&wq);

    return count;

}

1. **高级休眠**

**进程如何进行休眠**

1. 分配并初始化一个wait\_queue\_t结构
2. 将wait\_queue\_t结构加入等待队列中
3. 设置进程状态

void set\_current\_state(int new\_state);

1. 检查唤醒条件，如果唤醒条件不满足，让出cpu，至此，代码会暂停在schedule后，直到进程状态被重新设为TASK\_RUNNING，进程得到cpu，代码接着执行

if (!condition)

schedule();

**手工休眠**

（1）定义wait\_queue\_t结构

DEFINE\_WAIT(my\_wait);

或者：

wait\_queue\_t my\_wait;

init\_wait(&my\_wait);

（2）将wait\_queue\_t结构加入等待队列中并设置进程状态

void prepare\_to\_wait(wait\_queue\_head\_t \*queue, wait\_queue\_t \*wait, int state);

1. 检查唤醒条件，进入休眠

if (!condition)

schedule();

1. 清理动作

void finish\_wait(wait\_queue\_head\_t \*queue, wait\_queue\_t \*wait);

示例：

static int scull\_getwritespace(struct scull\_pipe \*dev, struct file \*filp)

{

    while (spacefree(dev) == 0)

    {

        DEFINE\_WAIT(wait);

        if (filp->f\_flags & O\_NONBLOCK)

            return -EAGAIN;

        prepare\_to\_wait(&dev->outq, &wait, TASK\_INTERRUPTIBLE);

        if (spacefree(dev) == 0)

            schedule();

        finish\_wait(&dev->outq, &wait);

    }

    return 0;

}

1. **Poll和select**

应用空间poll和select和epoll都会维护一组文件，当某个文件可读或可写时，其会调用其绑定的函数

上述三个函数均需要驱动程序提供poll函数来支持，该函数原型如下：

unsigned int (\*poll) (struct file \*filp, poll\_table \*wait);

Poll\_table：是一个wait\_queue\_head\_t（等待队列）列表，我们需要将我们的等待队列加入到该列表中

返回值：返回设备的可读写状态

POLLIN

如果设备可被不阻塞地读, 这个位必须设置.

POLLRDNORM

这个位必须设置, 如果"正常"数据可用来读. 一个可读的设备返回 ( POLLIN | POLLRDNORM ).

POLLRDBAND

这个位指示带外数据可用来从设备中读取.

POLLPRI

高优先级数据(带外)可不阻塞地读取.

POLLHUP

当读这个设备的进程见到文件尾, 驱动必须设置 POLLUP(hang-up).

POLLERR

一个错误情况已在设备上发生.

POLLOUT

这个位在返回值中设置, 如果设备可被写入而不阻塞.

POLLWRNORM

一个可写的设备返回( POLLOUT | POLLWRNORM).

POLLWRBAND

如同 POLLRDBAND , 这个位意思是带有零优先级的数据可写入设备.

示例：

static unsigned int scull\_p\_poll(struct file \*filp, poll\_table \*wait)

{

    struct scull\_pipe \*dev = filp->private\_data;

    unsigned int mask = 0;

    down(&dev->sem);

    poll\_wait(filp, &dev->inq, wait);       // 将读队列加入 wait 中

poll\_wait(filp, &dev->outq, wait);      // 将写队列加入 wait 中

    if (dev->rp != dev->wp)

        mask |= POLLIN | POLLRDNORM;        // 返回当前设备可读

    if (spacefree(dev))

        mask |= POLLOUT | POLLWRNORM;       // 返回当前设备可写

    up(&dev->sem);

    return mask;

}

1. **异步通知**

用户程序中开启异步的代码如下：

signal(SIGIO, &input\_handler);                  // 绑定 SIGIO 信号处理函数

fcntl(STDIN\_FILENO, F\_SETOWN, getpid());        // 改变文件 STDIN\_FILENO 的属主

oflags = fcntl(STDIN\_FILENO, F\_GETFL);          // 获取文件标志

fcntl(STDIN\_FILENO, F\_SETFL, oflags | FASYNC);  // 设置文件标志位异步

fcntl函数用于改变文件属性

**从内核角度看待这一过程**

1. 当发出 F\_SETOWN, 修改文件结构的属主 filp->f\_owner.
2. 当 F\_SETFL 被执行 FASYNC 位被置位时, 驱动程序的 fasync 方法被调用.
3. 当数据到达, 所有的注册异步通知的进程必须被发出一个 SIGIO 信号.

**驱动程序的fasync**

static int scull\_p\_fasync(int fd, struct file \*filp, int mode)

{

    // 头文件 <linux/fs.h>

    struct scull\_pipe \*dev = filp->private\_data;

    return fasync\_helper(fd, filp, mode, &dev->async\_queue);    // 将文件与异步队列管理

}

**驱动程序的write**

当其他进程调用write向驱动程序写入数据时，我们应该通知那些等待异步读的进程

    // 头文件 <linux/fs.h>

    if (dev->async\_queue)

        kill\_fasync(&dev->async\_queue, SIGIO, POLL\_IN);     // 通知异步队列