**注册块设备驱动程序**

编写驱动程序的第一步当然是注册驱动程序

// 注册块设备驱动程序

int register\_blkdev(unsigned int major, const char \*name);

// 注销块设备驱动程序

int unregister\_blkdev(unsigned int major, const char \*name);

Major：主设备号，如果为0将由内核自动分配

Name：设备名称

**注册磁盘**

**Gendisk结构**

内核使用struct gendisk（定义于 <linux/genhd.h>）表示一个磁盘

int major：主设备号

int first\_minor：第一个次设备号（一个分区一个次设备号）

int minors：次设备号的数量

char disk\_name[32]：磁盘设备的名称，该名称显示在 /proc/partitions 中

struct block\_device\_operations \*fops;

设备的操作集合

struct request\_queue \*queue;

设备IO请求队列

sector\_t capacity;

以 512字节为一个扇区来计算，该磁盘包含的扇区数，驱动程序不能直接设置该成员，使用set\_capacity来设置.

void \*private\_data;

块驱动程序可使用这个成员作为一个指向它们自己内部数据的指针.

**block\_device\_operations结构**

block\_device\_operations是磁盘设备的操作集合

int (\*open)(struct inode \*inode, struct file \*filp);

打开设备操作

int (\*release)(struct inode \*inode, struct file \*filp);

关闭设备操作

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

实现 ioctl 系统调用的方法.

int (\*media\_changed) (struct gendisk \*gd);

内核调用该方法检查驱动器的介质，如果是，返回非0，该方法用于可移动介质

int (\*revalidate\_disk) (struct gendisk \*gd);

当介质被更换时，该方法被调用

struct module \*owner;

一个指向拥有这个结构的模块的指针; 它应当常常被初始化为 THIS\_MODULE.

**分配磁盘（gendisk）结构**

struct gendisk \*alloc\_disk(int minors);

Minors：次设备号的数目

**向内核添加磁盘结构**

void add\_disk(struct gendisk \*gd);

**删除磁盘**

void del\_gendisk(struct gendisk \*gd);

**块设备操作（block\_device\_operations）**

**open和release函数**

示例：

static int sbull\_open(struct inode \*inode, struct file \*filp)

{

    // 当inode指向的时磁盘设备时，inode->i\_bdev->bd\_disk 指向 gendisk 结构

    struct sbull\_dev \*dev = inode->i\_bdev->bd\_disk->private\_data;

filp->private\_data = dev;

    spin\_lock(&dev->lock);

    // 检查引用计数

    if (!dev->users)

        // 告诉内核检查介质是否发送改变

        check\_disk\_change(inode->i\_bdev);

    // 增加引用计数

dev->users++;

spin\_unlock(&dev->lock);

    return 0;

}

static int sbull\_release(struct inode \*inode, struct file \*filp)

{

struct sbull\_dev \*dev = inode->i\_bdev->bd\_disk->private\_data;

    spin\_lock(&dev->lock);

    // 减少引用计数

    dev->users--;

    spin\_unlock(&dev->lock);

    return 0;

}

**media\_changed和revalidate\_disk操作，可移动介质的支持**

// 告诉内核是否改变了介质，我们前面的open方法的check\_disk\_change调用会触发该函数

int sbull\_media\_changed(struct gendisk \*gd)

{

    // 这里只是简单的示例

    struct sbull\_dev \*dev = gd->private\_data;

    return dev->media\_change;

}

// 如果介质发生改变，内核则会调用该方法通知驱动程序

int sbull\_revalidate(struct gendisk \*gd)

{

    struct sbull\_dev \*dev = gd->private\_data;

    if (dev->media\_change)

    {

        dev->media\_change = 0;

        memset(dev->data, 0, dev->size);

    }

    return 0;

}

**Ioctl函数**

与字符设备函数相同，不做介绍

**请求处理**

gendisk结构包含一个请求队列，这个请求队列绑定一个请求处理函数

**请求队列（request\_queue）**

创建请求队列

request\_queue\_t \*blk\_init\_queue(request\_fn\_proc \*request, spinlock\_t \*lock);

Request：请求处理函数

Lock：自旋锁

移除请求队列

void blk\_cleanup\_queue(request\_queue\_t \*);

返回下一个要处理的请求

struct request \*elv\_next\_request(request\_queue\_t \*queue);

如果无请求，返回NULL

**请求（request）**

Request\_queue\_t是request的队列，request结构如下：

sector\_t sector：

请求的开始扇区，以固定512字节为一扇区

unsigned long nr\_sectors：

请求操作的扇区数

char \*buffer;

一个指向缓冲区的指针，这个指针是一个内核虚拟地址并且可被驱动直接解引用

struct bio \*bio;

bio 是给这个请求的 bio 结构的链表. 你不应当直接访问这个成员，使用 rq\_for\_each\_bio(后面描述) 来访问.

请求方向

rq\_data\_dir(struct request \*req);

这个宏从请求中获取传送的方向，返回 0 表示从设备中读, 否则为写入设备.

将请求从队列中移除

void blkdev\_dequeue\_request(struct request \*req);

**请求处理函数（request\_fn\_proc）**

当内核需要读写设备时会调用该函数

void request(request\_queue\_t \*queue);

Queue：请求处理的队列

一个简单的处理函数

static void sbull\_request(request\_queue\_t \*q)

{

    struct request \*req;

    // 获取下一个请求

    while ((req = elv\_next\_request(q)) != NULL)

    {

        // req->rq\_disk 为 gendisk 结构

        struct sbull\_dev \*dev = req->rq\_disk->private\_data;

        // 如果请求不是文件系统的请求

        if (!blk\_fs\_request(req))

        {

            printk(KERN\_NOTICE "Skip non-fs request\n");

            // 无法完成

            end\_request(req, 0);

            continue;

        }

        // sbull自定义的函数，向磁盘读取或写入

        sbull\_transfer(dev,

                       req->sector,

                       req->current\_nr\_sectors,

                       req->buffer,

                       rq\_data\_dir(req));

        // 完成请求

        end\_request(req, 1);

    }

}

static void sbull\_transfer(struct sbull\_dev \*dev, unsigned long sector, unsigned long nsect, char \*buffer, int write)

{

    unsigned long offset = sector \* KERNEL\_SECTOR\_SIZE;

unsigned long nbytes = nsect \* KERNEL\_SECTOR\_SIZE;

    if ((offset + nbytes) > dev->size)

    {

        printk(KERN\_NOTICE "Beyond-end write (%ld %ld)\n",

               offset, nbytes);

        return;

}

    // 写入或读取

    if (write)

        memcpy(dev->data + offset, buffer, nbytes);

    else

        memcpy(buffer, dev->data + offset, nbytes);

}

**请求过程剖析**

从本质上讲，request是一个bio结构的链表

**Bio结构**

bio 结构在 <linux/bio.h> 中定义

sector\_t bi\_sector：

这个 bio 要被传送的第一个（512 字节计算）扇区.

unsigned int bi\_size;

被传送的数据大小, 以字节计

unsigned long bi\_flags;

一组描述 bio 的标志，如果这是一个写请求则最低位被置位（使用bio\_data\_dir(bio)宏获取传输方向）

Bio的合适是一个名为bi\_io\_vec的数组，其成员结构如下

struct bio\_vec

{

    struct page \*bv\_page;

    unsigned int bv\_len;

    unsigned int bv\_offset;

};

我们不推荐直接访问该数组，而是使用如下宏访问该数组

int segno;

struct bio\_vec \*bvec;

// segno 是当前的段号

bio\_for\_each\_segment(bvec, bio, segno)

{

    /\* Do something with this segment

}

为追踪bio状态，内核提供了如下宏

struct page \*bio\_page(struct bio \*bio);

返回一个指向页结构的指针, 表示下一个被传送的页.

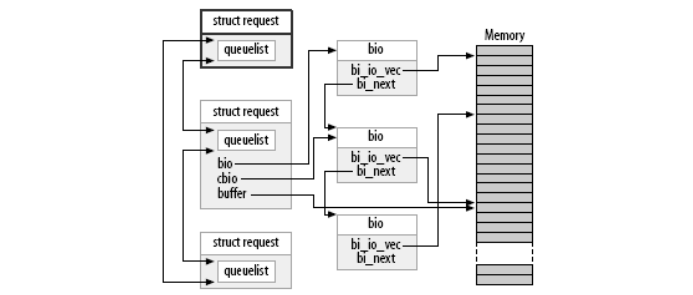
int bio\_offset(struct bio \*bio);

返回页内的被传送的数据的偏移.

int bio\_cur\_sectors(struct bio \*bio);

返回要被传送出当前页的扇区数.

request, bio, bi\_io\_vec与内存的关系



**请求完成函数**

当设备完成IO请求的部分或全部时，必须调用如下函数通知块设备子系统

int end\_that\_request\_first(struct request \*req, int success, int count);

Req：请求

Success：是否成功，1为成功，0为失败

Count：本次传输的扇区数

返回值：0表示该请求（req）的扇区数以全部传输完成

当完成IO请求的全部时，必须调用如下函数通知块设备子系统

void end\_that\_request\_last(struct request \*req);

其函数会将请求从队列中删除

关于end\_request

我们在上面的一个例子中使用的是内核的另一个函数end\_request完成IO请求，其源码如下：

void end\_request(struct request \*req, int uptodate)

{

    if (!end\_that\_request\_first(req, uptodate, req->hard\_cur\_sectors))

    {

        add\_disk\_randomness(req->rq\_disk);

        blkdev\_dequeue\_request(req);

        end\_that\_request\_last(req);

    }

}

**使用bio的例子**

// 请求处理

static void sbull\_full\_request(request\_queue\_t \*q)

{

    struct request \*req;

    int sectors\_xferred;

    struct sbull\_dev \*dev = q->queuedata;

    // 获取下一个请求

    while ((req = elv\_next\_request(q)) != NULL)

    {

        // 如果请求不是文件系统发出的

        if (!blk\_fs\_request(req))

        {

            printk(KERN\_NOTICE "Skip non-fs request\n");

            end\_request(req, 0);

            continue;

        }

        // sbull处理请求函数

        sectors\_xferred = sbull\_xfer\_request(dev, req);

        // 完成请求的一部分

        if (!end\_that\_request\_first(req, 1, sectors\_xferred))

        {

            blkdev\_dequeue\_request(req);

            // 完成全部请求的全部

            end\_that\_request\_last(req);

        }

    }

}

static int sbull\_xfer\_request(struct sbull\_dev \*dev, struct request \*req)

{

    struct bio \*bio;

int nsect = 0;

    // 循环bio数组

    rq\_for\_each\_bio(bio, req)

    {

        // sbull的bio处理函数

        sbull\_xfer\_bio(dev, bio);

        nsect += bio->bi\_size / KERNEL\_SECTOR\_SIZE;

    }

    return nsect;

}

static int sbull\_xfer\_bio(struct sbull\_dev \*dev, struct bio \*bio)

{

    int i;

    struct bio\_vec \*bvec;

sector\_t sector = bio->bi\_sector;

    /\* 循环bi\_io\_vec数组 \*/

    bio\_for\_each\_segment(bvec, bio, i)

    {

        // 映射bi\_io\_vec到内核逻辑地址

        char \*buffer = \_\_bio\_kmap\_atomic(bio, i, KM\_USER0);

        // sbull模拟数据到缓存或缓存到设备的函数

        sbull\_transfer(dev, sector, bio\_cur\_sectors(bio), buffer, bio\_data\_dir(bio) == WRITE);

        sector += bio\_cur\_sectors(bio);

        // 解除映射

        \_\_bio\_kunmap\_atomic(bio, KM\_USER0);

    }

    return 0; /\* Always "succeed" \*/

}