块设备驱动的引入：

1. 总结“字符设备驱动程序”：

1．引入“字符设备驱动程序”：

APP: open, read, write. ----> 对应提供驱动程序的读写等函数。

-------------------------------------

驱动：drv\_open, drv\_read, drv\_write

-------------------------------------

硬件

当应用程序的 open,read 等函数要操作“硬件”时，自然引入了“驱动程序”的概念，最简单的方式是APP 调用open 时，驱动程序的“drv\_open”函数被调用等等。

2．这种最简单的驱动程序的写法：

代码步骤:

1．确定主设备号：可以自已确定，也可让内核分配。

2． 要构造驱动中的“open,read,write 等”是将它们放在一个“file\_operations”结构体中.

File\_operations==> .open, .read, .write, .poll 等。

这里"open"函数会去配置硬件的相关引脚等，还有注册中断。

3．register\_chrdev 注册字符设备构造的“file\_operations”结构:使用这个file\_operations 结构体。是把这个结构放到内核的某个以此设备的“主设备号”为下标的数组中去。

Register\_chrdev(主设备号，设备号，file\_operations 结构）.

4．入口函数：调用这个“register\_chrdev()”函数。内核装载某个模块时，会自动调用这个“入口函数”。

5．出口函数：

3．如何知道设备有数据过来：如按键。

①，查询方式：

驱动程序中提供一个 读函数，直接返回某引脚状态，应用程序只能连续不断的读，比较前后两次的引脚状态是否有变化。这种查询方式“太累”。CPU 会占用很高。这样引入“休眠唤醒”机制。

②．休眠唤醒：

APP 用 读函数 进入驱动程序的读函数。在驱动程序里，若没有数据就休眠，被唤醒时，就copy\_to\_user 把数据拷贝到用户空间后返回。

其中“唤醒”由“中断服务”程序来唤醒。在驱动程序中用“request\_irq”注册“中断服务”程序。

但可能没有“唤醒”的时候。就想着加个“闹钟--poll”。

③，poll 机制：定时的时长内，若“中断服务”程序有来唤醒就好，若没有来，则定时超时由“闹钟”来唤醒。

但“休眠唤醒”和“poll”都得“休眠”。

④，“异步通知”：发信号

以上4 种方法是写“字符驱动”的基本形式。但这4 种情况有个缺点是这种驱动只有自已方便使用。写成通用的驱动程序，则是看懂内核代码，将自已的驱动代码融合进去。引出“输入子系统”。对“按键”是“输入子系统”，对LCD 是“framebuffer”。

⑤，“输入子系统”：我们自己写的驱动只有我们自己知道想让他成为更通用的版本如何写？就是引入输入子系统。融入别人写的代码（也是上面4 种方式写成）。

二，块设备驱动程序：

若块设备驱动程序也按以下字符设备驱动程序的简单思想来写：

APP : open, read, write. ----> 对应提供驱动程序的读写等函数。

-------------------------------------

块设备驱动：drv\_open, drv\_read, drv\_write

-------------------------------------

硬件 ： 块设备。

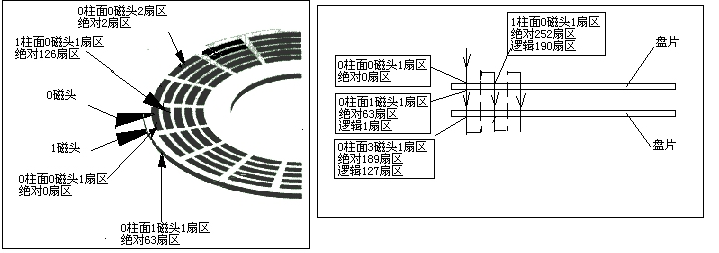
举例：

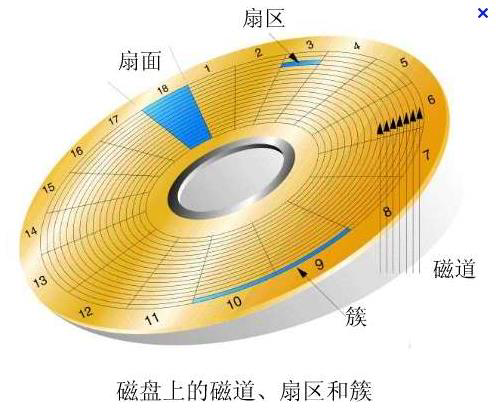
1. 硬盘：

磁头的概念：

就是右图的盘片：上下表面分别为磁头0 ，1 ，2， 3。

左下图像跑道一样东西：称为柱面做外边位0柱面。





磁盘的读写其实非常快，慢在机械结构读写装置的定位上面，从一个“磁头”的某“柱面”某“扇区”读到数据后（步骤R0），跳到另一个“磁头”的某“柱面”的某“扇区”去写（步骤W），接着再跳回原“磁头”相同柱面的下一个“扇区”去读（步骤R1）。慢就慢在读写扇区的跳转过程中。若按“字符设备”中的“opne”,"read","write" 方式，则总体效率在硬盘的读写上会非常低。

上面过程是“R0”->"W"->"R1"，这个步骤跳转2 次。

若优化这个步骤为：R0->R1->W。这个步骤跳转1 次。这样效率会高些。

总结：先不执行而是放入队列，优化后再执行（对硬盘有这种要求）。用“字符设备驱动”程序那样读写时就会在硬盘上跳

来跳去，整体效率会非常低。所以有必要引入“优化过程”。就是读写先不执行，先放到某个“队列”中去。（调整顺序）

2，flash：

是“块”里有很多一个一个的扇区。

假若现在要先写“扇区0”和“扇区1”。FLASH 要先擦除再写，现在用字符设备驱动的读写方式来读写：

对FLASH 上的擦除是整块整块的进行的。故：

写扇区0 的过程：

①，要写时，先把这整块读到一个 buf 中。

②，然后修改 buf 中扇区0 的数据。

③，这时再擦除整块。

④，再把修改过扇区0 的数据的 buf 烧写到整块。

写扇区1 的过程：

①，要写时，先把这整块读到一个 buf 中。

②，然后修改 buf 中扇区1 的数据。

③，这时再擦除整块。

④，再把修改过扇区1 的数据的 buf 烧写到整块。

则那么要修改多个扇区时，会擦除烧写多次。总体效率也会低。

优化：

①，先不执行。

②，优化 - 合并后执行。

合并：合并后只需要一次。

a,读出整块到buf 中。

b,在buf 中修改扇区0 和扇区1。

c,擦除。

d,烧写。

故，块设备不能像字符设备那样提供读写函数。

1. 先把读写放入队列，先不执行。

②，优化后再执行。

块设备驱动框架

**一.层次框架：**

APP: open,read,write "1.txt"

------------------------------------------ 文件读写

文件系统：vfat,ext2,ext3,yaffs （把文件的读写转换成对扇区的读写）//文件的读写操作肯定会转换位 对扇区的读写

------------------ll\_rw\_block-------------- 扇区读写会调用ll\_rw\_block（）函数

块设备驱动程序

硬件：硬盘、FLASH

对普通文件1.txt 的读写会转成对块设备的读写，要读写哪个扇区。从文件的读写转成对扇区的读写，中间会涉及到“文件系统”。

//我们对文件的读写怎末转换位对扇区的读写？通过文件系统这个桥梁。

应用程序读写一个变通的文件，最终会转换成操作硬件，由“块设备驱动程序”来操作硬件。

普通的文件转换成对扇区的读写，是由“文件系统”转换。

ll\_rw\_block 是通用的入口，读写请求会放入队列，优化后再来执行。LL\_rw\_block 会把“读写”放入队列，调用队列的处理函数去优化（调顺序、合并）执行。

如何知道是“ll\_rw\_block”可以看《LINUX 内核源代码情景分析》。文件系统不是我们关心的重点，分析块设备驱动程序，就是“ll\_rw\_block”这个函数开始分析。

**二.流程：**

分析“ll\_rw\_block”：1，把“读写”放入队列。2，调用队列的处理函数。

文件系统把一个普通文件的读写转换成块设备扇区的读写，最终就会调用这个底层的函数“ll\_rw\_block”。

这个函数在内核“fs”目录下，这个“fs”目录下有各种各样的文件系统（文件的组织格式-纯软件的概念。文件如何排，目录等如何表示），“fs”目录下还有很多通用的文件， ll\_rw\_block 这个函数在“buffer.c”是这个"fs"目录下的通用文件。

void ll\_rw\_block(int rw, int nr, struct buffer\_head \*bhs[])

参1，表示是读或是写。

参3，数据传输三要素（源，目的，长度）放到参3 buffer\_head 结构的数组中。这个数组有参2nr 个数组项。

参2，有nr 个参3 buffer\_head 结构的数组项。

for (i = 0; i < nr; i++) //开始就for 循环。

-->struct buffer\_head \*bh = bhs[i];// bh 等于这个参3 数组的某一项。

-->submit\_bh(WRITE, bh); //提交buffer\_head 结构的bh。

-->struct bio \*bio;//使用buffer\_head 来构造bio（block input/output）.

下面就是用 buffer\_head 结构体bh 来构造bio 的情况：

bio->bi\_sector = bh->b\_blocknr \* (bh->b\_size >> 9);

bio->bi\_bdev = bh->b\_bdev;

bio->bi\_io\_vec[0].bv\_page = bh->b\_page;

bio->bi\_io\_vec[0].bv\_len = bh->b\_size;

bio->bi\_io\_vec[0].bv\_offset = bh\_offset(bh);

bio->bi\_vcnt = 1;

bio->bi\_idx = 0;

bio->bi\_size = bh->b\_size;

bio->bi\_end\_io = end\_bio\_bh\_io\_sync;

bio->bi\_private = bh;

-->submit\_bio(rw, bio); //提交 bio 。

-->generic\_make\_request(bio); //使用bio(“通用的构造请求”)构造请求。把请求放入队列。

-->\_\_generic\_make\_request(bio);

-->q = bdev\_get\_queue(bio->bi\_bdev); //找到队列。request\_queue\_t \*q;q 是请求队列。

-->ret = q->make\_request\_fn(q, bio); //调用队列中一个函数“make 构造request 请求的fn 函数”。在如下地址设置“q->make\_request\_fn = mfn;”mfn 在下面这个函数的形参里。

void blk\_queue\_make\_request(request\_queue\_t \* q, make\_request\_fn \* mfn)

{

q->make\_request\_fn = mfn;

}

blk\_queue\_make\_request最终会被request\_queue\_t \*blk\_init\_queue\_node如下的这个调用

request\_queue\_t \*blk\_init\_queue\_node(request\_fn\_proc \*rfn, spinlock\_t \*lock, int node\_id)

-->blk\_queue\_make\_request(q, \_\_make\_request);

最终“make\_request\_fn()”的默认函数是“\_\_make\_request()”//提供的默认的构造请求函数把请求放入队列

int \_\_make\_request(request\_queue\_t \*q, struct bio \*bio)

-->el\_ret = elv\_merge(q, &req, bio);

merge 合并。这里是先尝试合并。elv 是电梯调试算法。以elv 算法把bio 合并到请求队列。假若1 楼A 要上，2 楼B 要下，三楼C 要上，则不会让A 上了再让B 下了最后再跑到3 楼让C 上，而是先让电梯从1 楼上到头，再下来。一次传送中输运同样方向的人。（在一个方向上合并起来运输）。

-->init\_request\_from\_bio(req, bio); //若合并不成功，就用bio 构造请求。

-->add\_request(q, req); //把请求放到队列中去。

//下面接着就有一个执行队列的函数：\_generic\_unplug\_device(q)

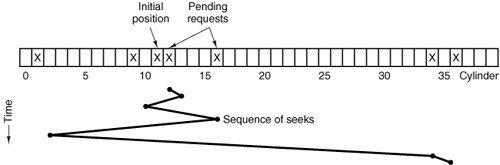
-->\_generic\_unplug\_device(q); 执行队列，但并不一定是在这个函数里执行。是if (sync)若同步时才执行。

-->q->request\_fn(q);//其实执行队列的方式就是调用队列的处理函数（request\_fn）。

电梯调度算法：

电梯算法主要用于磁盘寻道的优化：、

第一种是我们最为原始的先到先服务(first come first served)的算法，这个对于我们去下馆子撮一顿比较合适，先来就先吃，不然顾客有意见。不过对于磁盘寻道就不太合适了。如下图:



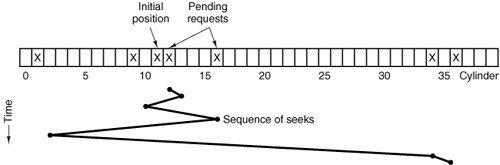
注意这张图并不是解释的先到先服务算法，我们只是借用下而已 :)

假设此时我们正在第11 道读取数据，然后陆陆续续有其他进程来要求我们提供磁盘内容给他们。这里我们把要读取的柱面 (如果你并不是研究磁盘寻道，那么这个词你可以理解为数据块，就是上面的小方块)按照进程提出要求的顺序记录下来的是1, 36, 16, 34, 9, 12,那么严格按照先到先服务原则，我们下一个要去的柱面是1 号，中间要经历10 个柱面，然后是36 号....... 等全部读下来，我们统计下，一共要"跑过"111 个柱面。

很明显的，这个算法效率太低，我们要来改善下算法。

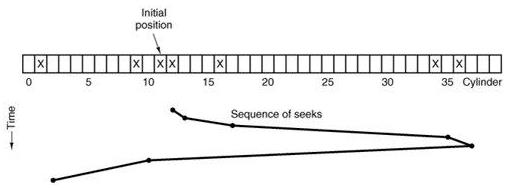
第二种是最短寻道算法(shortest seek first)

这种算法有点类似贪心，即是每次我们选择距离我们现在所处的点最近的一个点(柱 面)。如下图，若当前我们正好执行完对于11 号块的读取，下一个最近的是12 号块，那么我们读取12 号块的数据，接着读取16 号块...... 我们看到如果用这种算法的话，我们经过的方块号码 12, 9, 16, 1, 34, 36 这样我们总共的经历的柱面数为 61 块，这样我们大大节省了寻道时间。



这个算法本来已经很好了，不过我们不得不面临这样一个问题： 现在我们正在读取16 号块，马上要读取1 号块了，这是一个进程闯进来要求我们为他提供20 号块的信息，20 号距离16 号比较近，那我们就去二十号吧，然后 我们又接到通知要23 号数据.....这样一直做下去，呃，1 号信息呢？天晓得要等到什么时候去读取它内容！

所以这里我们需要一种算法来平衡效率和公平性(我们也不希望歧视了1 号小方块)。所以我们引进了电梯算法 。我们需要做一个标记，标记现在是向数字大的方向读，还是方向小的。如果现在是向前(数字大)读，那么我们就需要一直读下去，一直到最尾一个。同理向后读。这个算法如下图所示:



三，如何写块设备驱动程序：

LINUX 驱动程序中老种驱动都构造了一个结构体。

1，以面向对象的思想分配 gendisk 结构体。用alloc\_disk 函数。

2，设置 gendisk 结构体。

①，分配/设置一个队列：request\_queue\_t. （提供读写能力）用 blk\_init\_queue 函数。

1. ，设置 gendisk 其他信息。（提供磁盘属性：磁盘容量，扇区大小等）

3.注册 gendisk 结构体。用 add\_disk 函数。

块设备驱动编写

分配一段内存，用内存来模拟硬盘：

搜索“blk\_init\_queut()”参考“drivers\block\xd.c”和“drivers\block\z2ram.c”两个文件。

大致分析：

看一个驱动程序从“入口函数”开始看。

Xd.c ：

1. register\_blkdev(XT\_DISK\_MAJOR, "xd")

“file\_operations”结构体。其实这个注册块设备已经退化了。

只是在Cat /proc/devices 可以让你看到一些信息，若者这个“XT\_DISK\_MAJOR”主设备号写为0 时，register\_blkdev（0，“xd”）可以返回一个主设备号给你。

2．xd\_queue = blk\_init\_queue(do\_xd\_request, &xd\_lock);

初始化一个队列。

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

形参1 是在队列里传入一个处理函数“request\_fn\_proc”函数。

3．如何使用队列xd\_queue：

-->struct gendisk \*disk = alloc\_disk(64);分配一个 gendisk 结构体变量 disk

-->disk->queue = xd\_queue; 将此分配的gendisk 结构disk 的成员“queue”等于之前初始化的队列“xd\_queue”.

过程中还有主设备号，容量等等属性的设置。

4．add\_disk(xd\_gendisk[i]);

下面直接自已写代码：

一，入口函数：

//入口函数

static int ramblock\_init(void)

{

//1,分配一个 gendisk 结构体

//2,设置

//2.1,分配/设置队列:提供读写能力

//2.2,设置其他属性:比如容量等等.

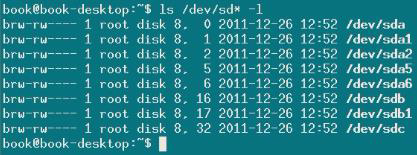
//3,注册

}

1．分配一个gendisk 结构体：allock\_disk()

alloc\_disk (int minors)需要参数“minor+s”是指次设备号个数。即“分区个数+0”。0 是指整个磁盘。为1 时，就是把整个磁盘当成一个分区，则在上面不能再创建分区了。如写成16，则最多可以创建15 个分区。

若是说，这个主设备号之下，从哪个次设备号开始都是对应这个块设备。



对于一个块设备，次设备号为“0”时，表示整个磁盘。如“/dev/sda”。次设备号“1， 2，、”表示是磁盘的第几个 主 分区。次设备号从5 开始是“扩展分区”。

2,分配/设置队列：blk\_init\_queue()

2,设置

//2.1,分配/设置队列:提供读写能力

ramblock\_queue = blk\_init\_queue (do\_rambloc\_request, &ramblock\_lock);

blk\_init\_queue (request\_fn\_proc \* rfn, spinlock\_t \* lock) 分配/设置一个队列。

参1，执行处理队列的函数。

参2，一个“自旋锁”。 DEFINE\_SPINLOCK (beep\_lock)

之前分析把“文件读写”转成“扇区读写”，对“扇区的读写”会放入个队列里面：

把“buffer\_head”构造为“bio”，把“bio”放入到队列里面，调用队列里面的“q->

make\_request\_fn”函数，这个“make\_request\_fn”构造请求的函数有默认的函数“\_\_make\_request”。

当我们初始化队列时，给我们提供了一个默认构造请求的函数“\_\_make\_request”。当它把这个“请求”放入到“队列”之后，以后会用这个队列里的“q->request\_fn()”来处理。这个“request\_fn()”就等于“blk\_init\_queue\_node（）”函数传进来的参数“形参1：request\_fn\_proc \*rfn”。这个形参1 会赋给“request\_fn”，最后就是我们要定义的一个处理请求的函数。

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

-->blk\_init\_queue\_node(rfn, lock, -1);

-->blk\_queue\_make\_request(q, \_\_make\_request);提供了默认构造请求的函数。

对于我们这里这个把内存模拟成硬盘的驱动来说，最终会是我们这里定义的“do\_rambloc\_request”来处理请求。

3，设置其他属性：

①，alloc\_disk（）：

②，主设备号：

static int major;

major = register\_blkdev(0, "ramblock"); //自动分配主设备号

ramblock\_disk->major = major; //属性:主设备号

主设备号可以自已定义，也可以让系统自动分配。

Register\_blkdev()函数退化了，用此函数当参1 为“0”时可以让系统自动分配一个主设备号：

Static int major;

Major = register\_blkdev(0,"ramblock");

Register\_blkdev()相比"register\_chrdev()"少了一个形参3“file\_operations”结构体。只不过当形参1 为“0”时，register\_blkdev()可以自动为块设备分配一个主设备号，且在“cat /proc/device”时可以看到有哪些块设备：



③，第一个次设备号是什么 和 块设备的名字。

ramblock\_disk->first\_minor = 0; //第一个次设备号写为0，则从0~16都对应这个块设备。

printf(ramblock\_disk->disk\_name, "ramblock"); //块设备的名字

④，fops ：操作函数。即使是空的操作函数，这个fops 也要提供。

static struct block\_device\_operations ramblock\_fops = {

.owner = THIS\_MODULE,

.ioctl = ramblock\_ioctl,

.getgeo = ramblock\_getgeo,

};

经过实验，即使这个函数里什么也没有（如没有".ioctl"和".getgeo"），也要提供这个结构体。不然会出错。

ramblock\_disk->fops = &ramblock\_fops; //必要的操作函数

（ramblock\_disk->private\_data = p;私有数据，这里我们不需要。）

⑤，设置队列：

把设置队列放在“分配队列”那一步骤代码紧跟的之下。就是把队列使用起来，即放到request\_queue\_t 结构变量ramblock\_queue 里中去。

//2.1,分配/设置队列:提供读写能力

ramblock\_queue = blk\_init\_queue (do\_rambloc\_request, &ramblock\_lock);

ramblock\_disk->queue = ramblock\_queue; //设置队列

⑥，容量：设置容量时，是以扇区为单位。

块设备容量 1M 字节

#define RAMBLOCK\_SIZE (1024\*1024)

set\_capacity(ramblock\_disk, RAMBLOCK\_SIZE / 512);

单位是“扇区”，在内核里，对于文件系统那一层，永远认为扇区是512 字节。

4．注册：add\_disk().

//3,注册

add\_disk(ramblock\_disk);

5，处理请求：

在“分配/设置队列 blk\_init\_queue()”函数中，形参1“request\_fn\_proc \*rfn”是用来“处理”队列中的请求的函数。

//定义"处理队列请求"的函数

static void do\_rambloc\_request (request\_queue\_t \* q)

{

static int cnt = 0;

printk("do\_rambloc\_request", ++cnt);

}

这里什么也没做，以后慢慢完善。以上一个驱动程序基本上写完了。

二，出口函数：

//出口函数

static void ramblock\_exit(void)

{

unregister\_blkdev(ramblock\_disk); //卸载块设备

del\_gendisk(ramblock\_disk); //清除gendisk结构

put\_disk(ramblock\_disk); //释放块设备结构空间

blk\_cleanup\_queue (ramblock\_queue); //清除队列.

}

这里是想用内存分模拟磁盘，但还未分配内存。这里先做实验。

三．测试：

1．编译



加载驱动后，马上用到了“do\_ramblock\_request”队列请求处理函数。但到这里后就再也不能返回到shell 了。因为这个函数对“队列 equest\_queue\_t \* q”没有任何处理。所以没有再返回。

看内核示例“drivers/block/xd.c”中的“do\_xd\_request(request\_queue\_t \* q)”：

以“电梯调度算法”从“request\_queue\_t \* q”队列里面取出下一个请求“req = elv\_next\_request(q)”，然后执行这个请求“req”。最后“end\_request(req, res); /\* wrap up, 0 = fail, 1 = success \*/”res 即0 或1.

将“队列请求处理函数”修改为下面：

//定义"处理队列请求"的函数

static void do\_rambloc\_request (request\_queue\_t \* q)

{

static int cnt = 0;

struct request \*req;

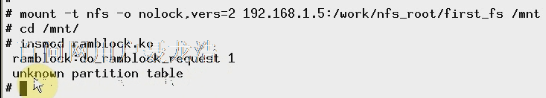
printk("do\_rambloc\_request", ++cnt);

while( (req = elv\_next\_request(q)) != NULL ){ //以电梯调度算法取出队列下一个请求,但没处理直接返回.

end\_request(req, 1);

}

}



可以返回到SHELL，只是说了不识别的分区表。

这样便可以装载块驱动后返回。在“while”中一般是在这里做真正的操作硬件。若是真正的磁盘，则从队列里取请求

开始读写操作即可。这里实验目的是“内存模拟磁盘”则一会用 memcpy 操作内存即可完成读写。



cat /proc/device //查看块设备



四．最终完善的内存模拟磁盘

1，“硬件相关操作”：分配内存。

//声明一个块内存

static unsigned char \*ramblock\_buf;

//3，硬件相关操作:

ramblock\_buf = kzalloc (RAMBLOCK\_SIZE, GFP\_KERNEL); //分配内存

2，在“队列请求”处理函数中操作：实现内存模拟磁盘。

①，数据传输三要素：

涉及到数据的传输，则离不开“三要素”：块设备数据传输的三要素在结构体“request” 中定义：

struct request {

sector\_t sector; /\* next sector to submit 下一个要提交的扇区。这就源或是目的\*/

…

unsigned int current\_nr\_sectors; //当前要处理的扇区个数。这是长度。

…

char \*buffer; //要写时这时目的(从扇区读数据到此)，要读时这是源。

｝

块设备操作时是以扇区为单位，即使是写一个字节，也是先读出一个扇区，再修改这个扇区里的那个字节后，再将整个扇区写进去

②，读写的方向：

while( (req = elv\_next\_request(q)) != NULL ){ //以电梯调度算法取出队列下一个请求,但没处理直接返回.

//数据传输三要素:源，目的，长度 结构request中会有这三要素.

//3.3，源 或者 目的

unsigned long offset = req->sector <<9; //偏移值。左移9位相当于乘以512-1个扇区大小. 故相当于 req->sector\*512.

//3.4, 目的 或者 源 : req->buffer

//3.5,长度

unsigned long len = req->current\_nr\_sectors <<9; //长度。当前要传输多少个扇区.

//方向:读写。

if(rq\_data\_dir(req) == READ) //若是读

{//memcpy(目的，源，长度);

memcpy(req->buffer, ramblock\_buf+offset, len); //从磁盘上的源读长度len到目的内存的buffer。

}

else //写则倒过来

{

memcpy(ramblock\_buf+offset, req->buffer,len); //把内存buffer中的数据写到磁盘上.

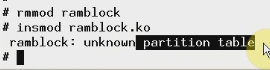
}

end\_request(req, 1);

}

这样这后，一会就可以格式化，挂接内容了。

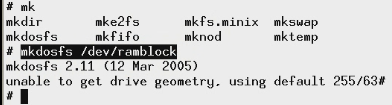
3，再编译后测试：

1. . insmod ramblock.ko
2. 

还是提示了“未识别的分区表”，这是因为分配的内存全都是清0 的。

ramblock\_buf = kzalloc (RAMBLOCK\_SIZE, GFP\_KERNEL); //分配内存

里面还没有分区表。

1. . 格式化: mkdosfs /dev/ramblock
2. 
3. 此时，视频里面的系统里只有“mkdosfs”，就用这个。我自己的文件系统没有mkdosfs
4. 挂接: mount /dev/ramblock /tmp/ //将/dev/ramblock挂接到/temp下
5. 读写文件: cd /tmp, 在里面用vi新创建 文件等都可以。
6. cd /; umount /tmp/ 后，重新挂载时，里面的内容还存在。
7. 

⑥. cat /dev/ramblock > /mnt/ramblock.bin //把这个块设备存成一个bin文件

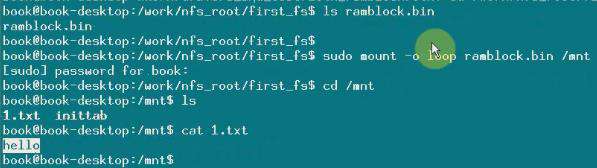
可以把整个磁盘cat 成一个bin 文件。相当于一个镜像。

1. 上面都是在开发板上操作的。

⑦. 在PC 上查看ramblock.bin

sudo mount -o loop ramblock.bin /mnt

（-o loop 回还设备：可将一个普通文件当成一个块设备文件挂接。



在ＰＣ机上也可以看到之前在开发板上创建的两个文件。证明块设备实验成功了。

用内存来模拟，忽略了硬件的复杂操作。这里只用了“ｍｅｍｃｐｙ（）”就实现了硬件操作。

进一步做实验，把“队列请求”处理函数中读写打印出来。

static void do\_rambloc\_request (request\_queue\_t \* q)

{

static int r\_cnt = 0;

static int w\_cnt = 0;

struct request \*req;

//printk("do\_rambloc\_request", ++cnt);

while( (req = elv\_next\_request(q)) != NULL ){ //以电梯调度算法取出队列下一个请求,但没处理直接返回.

//数据传输三要素:源，目的，长度 结构request中会有这三要素.

//3.3，源 或者 目的

unsigned long offset = req->sector <<9; //偏移值。左移9位相当于乘以512-1个扇区大小. 故相当于 req->sector\*512.

//3.4, 目的 或者 源 : req->buffer

//3.5,长度

unsigned long len = req->current\_nr\_sectors <<9; //长度。当前要传输多少个扇区.

//方向:读写。

if(rq\_data\_dir(req) == READ) //若是读

{//memcpy(目的，源，长度);

printk("do\_rambloc\_request read %d\n", ++r\_cnt);

memcpy(req->buffer, ramblock\_buf+offset, len); //从磁盘上的源读长度len到目的内存的buffer。

}

else //写则倒过来

{

printk("do\_rambloc\_request write %d\n", ++w\_cnt);

memcpy(ramblock\_buf+offset, req->buffer,len); //把内存buffer中的数据写到磁盘上.

}

end\_request(req, 1);

}

}

编译测试：

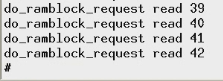


加载驱动时读了一次。



格式化时读写次数。



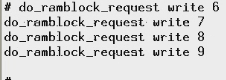


挂接时从，read４ 一直到了read４２



拷贝一个文件到磁盘上，发现也是“读”。

这是并没有立即写。块设备的读写操作会先放到队列里面，并不会立即执行。



到了“read 43”后，等了好些时候才出现上面的写。

有时候在WINDOWS 上可发现 拷贝文件到U 盘的“进度条”已完成 ，这时去卸载U 盘时， 会提示说“设备忙”，U 盘的灯也在闪。这表示在后台还要写。

但是要想让写到块设备上的文件立即写入块设备：sync（同步）



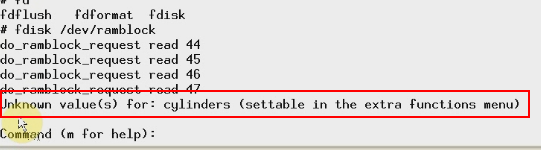
sync 是一个系统调用 ，同步的意思。



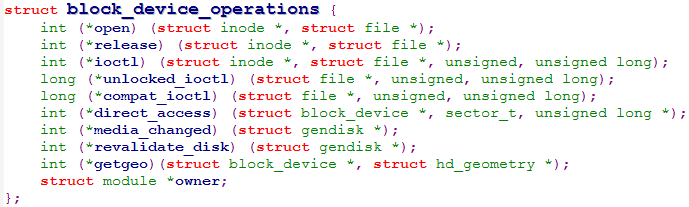
再写一个数据到块设备，发现拷贝命令结束时，并没有立即去写，这时“umount”这个块设备。就立即开始去写了。

上面的现象都是要么读，要么写，就像电梯一样，一路先上到顶，再下来。

五.开始试着分区：



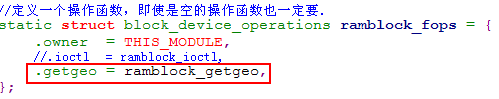
不知道“柱面”数。现在可能有的磁盘已经没有这种结构了，但是为了兼容这些“fdisk” 老工具，要假装说自已有多少个“磁头”，多少个“柱面”。这些信息就是由“block\_device\_operations（块设备的操作）.



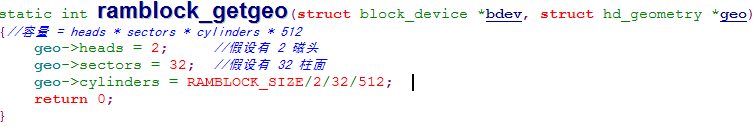
其中：

int (\*getgeo)(struct block\_device \*, struct hd\_geometry \*);

geometyr 几何的意思。获得几何属性。



要加上“.getgeo = ramblock\_getgeo”这个函数。用内存模拟磁盘，没有磁头，柱面，扇区的概念，但为了用这个“fdisk”老工具，也得假装有这些属性。



容量 = 磁头 \* 柱面 \* 扇区 \* 512 （容量 = heads \* sectors \* cylinders \* 512 ）

磁头：即有多少面。这里假设有 2 面。

柱面：有多少环，这里假设有 32 环。

扇区：一环里有多少个扇区，这个得通过“公式”计算出来。

重新加载有“磁头、柱面、扇区”信息的驱动。

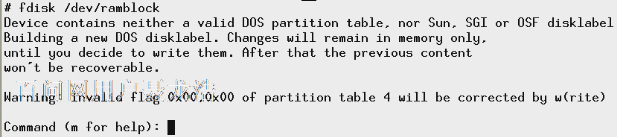


上面说没有识别的分区表，还是因为分配空间时，里面都是0，并没有分区信息。

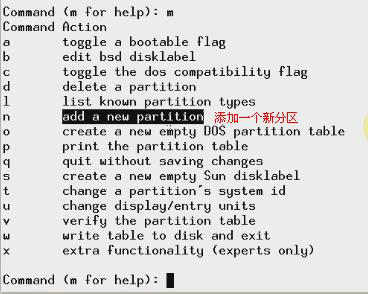
ramblock\_buf = kzalloc (RAMBLOCK\_SIZE, GFP\_KERNEL); //分配内存



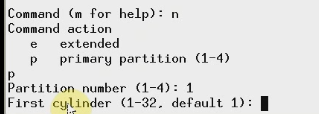
上面次设备号为“0”表示整个磁盘。



给这个1M 的空间分区。



“n”添加一个新分区。



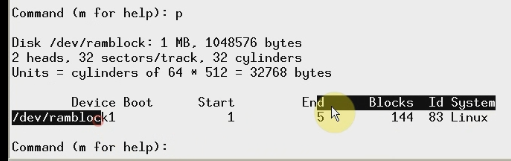


之前代码中构造几何信息时，柱面定的是32 个。

添加主分区。第一个“柱面 1-32”。Default 1:默认柱面为1.这里保持默认。

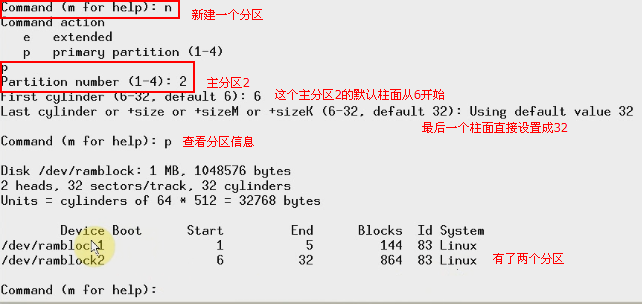


接着是结束柱面，这里定为“5”

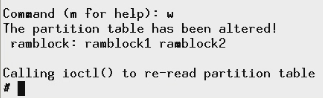


这时可以查看，有一个分区出来了。

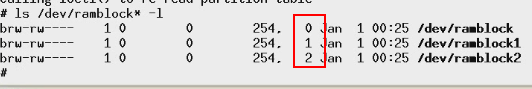
下面再增加一个分区：



最后写入硬盘



所谓分区表就是 磁盘 里的第一个扇区。



次设备号为“0”表示整个磁盘。

次设备号为“1”表示磁盘上的第一个主分区。

次设备号为“2”表示磁盘上的第二个主分区。



这时可以分别格式化，分别挂接。

若以后，硬盘也有像内存一样直接用“memcpy()”直接像内存一样访问时，那块设备就非常的简单了。但现在这只是美好的愿望。现在还是要把“memcpy()”替换成其他方法。这将是块设备复杂的地方。但块设备的框架如上所述，却是并不复杂的。

框架：

LINUX 驱动程序中老种驱动都构造了一个结构体。

APP: open,read,write "1.txt"

------------------------------------------ 文件读写

文件系统：vfat,ext2,ext3,yaffs （通过文件系统把文件的读写转换成对扇区的读写）

------------------ll\_rw\_block（）-------------- 扇区读写

块设备驱动程序

------------------------------------------

硬件：硬盘、FLASH

1，以面向对象的思想分配 gendisk 结构体。用alloc\_disk 函数。

2，设置 gendisk 结构体。

①，分配/设置一个队列：request\_queue\_t. （提供读写能力）用 blk\_init\_queue 函数。

②，设置 gendisk 其他信息。（提供磁盘属性：磁盘容量，扇区大小等）

3，注册 gendisk 结构体。用 add\_disk 函数。

操作不用我们关心，格式化、读写文件等都是由“文件系统”这一层将文件的读写转换成对扇区的读写的。例如我们cp xxx /temp （复制文件假如硬盘挂接在/temp下）调用“ll\_rw\_block”会把读写放到你的队列中去。会调用你“队列请求处理函数”来处理。只要你写好“队列请求处理”函数即可。