Block Device Analysis

宝德技术研究院 李磊

# 块设备的表示

图1-1表示的是块设备在各个成员在系统中的表示，block device起了一个中间层的作用，衔接了vfs和磁盘设备。



# 基本数据结构

块设备涉及到的基本数据结构比较多。下面分别列出。

## 块设备

struct block\_device {

dev\_t bd\_dev; /\* not a kdev\_t - it's a search key \*/

//inode of block device

struct inode \* bd\_inode; /\* will die \*/

//调用do\_open打开该块设备的次数

int bd\_openers;

struct mutex bd\_mutex; /\* open/close mutex \*/

struct mutex bd\_mount\_mutex; /\* mount mutex \*/

//链表头,链表成员是inode->i\_devices, 该链表包含了表示该设备的设备特殊文件的所

//有inode

struct list\_head bd\_inodes;

void \* bd\_holder;

int bd\_holders;

#ifdef CONFIG\_SYSFS

struct list\_head bd\_holder\_list;

#endif

//如果该block\_device描述的是一个分区，则该变量指向描述主块设备的block\_device，

//反之，其指向本身

struct block\_device \* bd\_contains;

unsigned bd\_block\_size;

//包含在该设备上的分区

struct hd\_struct \* bd\_part;

/\* number of times partitions within this device have been opened. \*/

//内核中引用该设备内分区的次数

unsigned bd\_part\_count;

//设置为1时,表示该分区在内核中的信息无效,因为磁盘上的分区已经改变,

//下一次打开该设备时,将要重新扫描分区表

int bd\_invalidated;

struct gendisk \* bd\_disk;

//链表元素,链表头是全局变量all\_bdevs,用于记录所有的块设备

struct list\_head bd\_list;

struct backing\_dev\_info \*bd\_inode\_backing\_dev\_info;

/\*

\* Private data. You must have bd\_claim'ed the block\_device

\* to use this. NOTE: bd\_claim allows an owner to claim

\* the same device multiple times, the owner must take special

\* care to not mess up bd\_private for that case.

\*/

unsigned long bd\_private;

};

## 通用硬盘

struct gendisk {

//磁盘主设备号

int major; /\* major number of driver \*/

//与磁盘关联的第一个次设备号

int first\_minor;

//主次分区的总个数,如果为1,表示只有主分区,无法分配次分区

//与磁盘关联的次设备号范围

int minors; /\* maximum number of minors, =1 for

\* disks that can't be partitioned. \*/

char disk\_name[32]; /\* name of major driver \*/

//次分区的数组, 每个分区对应一个数组项

struct hd\_struct \*\*part; /\* [indexed by minor] \*/

//if part\_uevent\_suppress is positive, 检测到设备的分区改变时

//就不会向用户空间发送热插拔事件.

int part\_uevent\_suppress;

struct block\_device\_operations \*fops;

//指向磁盘请求队列的指针

struct request\_queue \*queue;

void \*private\_data;

//磁盘容量,单位是扇区

sector\_t capacity;

//磁盘类型的标志,GENHD\_FL\_UP表示磁盘初始化并且可以使用,

//GENHD\_FL\_REMOVABLE:如果是可移动磁盘就要设置该标志

int flags;

//标识该磁盘所属的硬件设备,指针指向驱动模型的一个对象

struct device \*driverfs\_dev;

struct kobject kobj;

struct kobject \*holder\_dir;

struct kobject \*slave\_dir;

//该指针指向这个数据结构记录磁盘中断的定时,由内核内置的随机数发生器使用

struct timer\_rand\_state \*random;

//如果磁盘是只读的，则置为1（写操作禁止），否则为0

int policy;

//写入磁盘的扇区数计数器,仅为raid使用

atomic\_t sync\_io; /\* RAID \*/

//统计磁盘队列使用情况的时间戳

unsigned long stamp;

//正在进行的io操作数

int in\_flight;

#ifdef CONFIG\_SMP

//统计每个cpu使用磁盘的情况

struct disk\_stats \*dkstats;

#else

struct disk\_stats dkstats;

#endif

};

## 分区

struct hd\_struct {

//起始扇区

sector\_t start\_sect;

//扇区数量

sector\_t nr\_sects;

struct kobject kobj;

struct kobject \*holder\_dir;

unsigned ios[2], sectors[2]; /\* READs and WRITEs \*/

//policy为1表示分区只读,partno表示扇区编号

int policy, partno;

};

## 各部分之间的联系

各部分的关系如图2-4，图中位于最顶部的block device对象是主设备，其下两个block device为从设备，例如sda是主设备，sda1,sda2为从设备。从设备中的字段bd\_contains指向其主设备。在block device对象中的bd\_part指向其对应分区的struct hd\_struct。bd\_disk指向通用磁盘对象gendisk。通用磁盘对象gendisk的part对象是一个数组，数组成员为struct hd\_struct \*。每一个hd\_struct对象都表示一个分区。



图2-4

# 块子系统初始化

## genhd\_device\_init

Block子系统出初始化是通过下面这段代码完成的. bdev\_map是全局的一个hash table，用于管理系统中全部的block device的。

static int \_\_init genhd\_device\_init(void)

{

bdev\_map = kobj\_map\_init(base\_probe, &block\_subsys\_lock);

blk\_dev\_init();

subsystem\_register(&block\_subsys);

return 0;

}

subsys\_initcall(genhd\_device\_init);

### subsys\_initcall

这里比较奇怪的地方是这个subsys\_initcall，查看其实现:

#ifndef MODULE

#ifndef \_\_ASSEMBLY\_\_

#define subsys\_initcall(fn) \_\_define\_initcall("4",fn)

#endif

#else

#define subsys\_initcall(fn) module\_init(fn)

#endif

如果使用编译模块的形式去编译genhd.c，那么subsys\_initcall就是module\_init。

如果不是以编译模块的形式去编译这个文件，并且在asm中引用了subsys\_initcall的定义，那么subsys\_initcall就是\_\_define\_initcall。

#define \_\_define\_initcall(level,fn) \

static initcall\_t \_\_initcall\_##fn \_\_attribute\_used\_\_ \

\_\_attribute\_\_((\_\_section\_\_(".initcall" level ".init"))) = fn

typedef int (\*initcall\_t)(void);

从下面的typedef可以看出，initcall\_t是一个函数指针。那么\_\_define\_initcall宏只是定义一个静态的函数指针，并且声明要把这个指针放在一个特定的段里而已。在我们这种情况下就是定义一个下面这样一个函数的函数指针放在.initcall4.init这样一个段中。

static int \_\_initcall\_ genhd\_device\_init (void)

问题又来了，在linux kernel的许多子系统都是通过这种方式定义的，那子系统初始化是如何初始化的呢？实际上子系统初始化的流出如图:



来看看这个do\_initcalls到底什么样子。

static void \_\_init do\_initcalls(void)

{

initcall\_t \*call;

int count = preempt\_count();

for (call = \_\_initcall\_start; call < \_\_initcall\_end; call++) {

char \*msg = NULL;

char msgbuf[40];

int result;

if (initcall\_debug) {

printk("Calling initcall 0x%p", \*call);

print\_fn\_descriptor\_symbol(": %s()",

(unsigned long) \*call);

printk("\n");

}

result = (\*call)();

if (result && result != -ENODEV && initcall\_debug) {

sprintf(msgbuf, "error code %d", result);

msg = msgbuf;

}

if (preempt\_count() != count) {

msg = "preemption imbalance";

preempt\_count() = count;

}

if (irqs\_disabled()) {

msg = "disabled interrupts";

local\_irq\_enable();

}

if (msg) {

printk(KERN\_WARNING "initcall at 0x%p", \*call);

print\_fn\_descriptor\_symbol(": %s()",

(unsigned long) \*call);

printk(": returned with %s\n", msg);

}

}

/\* Make sure there is no pending stuff from the initcall sequence \*/

flush\_scheduled\_work();

}

其中最重要的部分就是for循环中的那个result = (\*call)()，这条语句就执行了\_\_initcall\_start到\_\_initcall\_end中的函数指针。在x86平台的链接脚本中定义了\_\_initcall\_start到\_\_initcall\_end的范围。

176 \_\_initcall\_start = .;

177 .initcall.init : AT(ADDR(.initcall.init) - LOAD\_OFFSET) {

178 \*(.initcall1.init)

179 \*(.initcall2.init)

180 \*(.initcall3.init)

181 \*(.initcall4.init)

182 \*(.initcall5.init)

183 \*(.initcall6.init)

184 \*(.initcall7.init)

185 }

186 \_\_initcall\_end = .;

### kobj\_map\_init

struct kobj\_map \*kobj\_map\_init(kobj\_probe\_t \*base\_probe, struct mutex \*lock)

{

struct kobj\_map \*p = kmalloc(sizeof(struct kobj\_map), GFP\_KERNEL);

struct probe \*base = kzalloc(sizeof(\*base), GFP\_KERNEL);

int i;

if ((p == NULL) || (base == NULL)) {

kfree(p);

kfree(base);

return NULL;

}

base->dev = 1;

base->range = ~0;

base->get = base\_probe;

//给backet初始化

for (i = 0; i < 255; i++)

p->probes[i] = base;

p->lock = lock;

return p;

}

这个函数主要就是建立起kobj\_map这样一个结构，linux中只有两个地方调用了这个函数，一个是字符设备的初始化，另一个就是块设备初始化。故系统中只有两个kobj\_map，分别管理字符设备和块设备。

### blk\_dev\_init

int \_\_init blk\_dev\_init(void)

{

int i;

kblockd\_workqueue = create\_workqueue("kblockd");

if (!kblockd\_workqueue)

panic("Failed to create kblockd\n");

request\_cachep = kmem\_cache\_create("blkdev\_requests",

sizeof(struct request), 0, SLAB\_PANIC, NULL, NULL);

requestq\_cachep = kmem\_cache\_create("blkdev\_queue",

sizeof(request\_queue\_t), 0, SLAB\_PANIC, NULL, NULL);

iocontext\_cachep = kmem\_cache\_create("blkdev\_ioc",

sizeof(struct io\_context), 0, SLAB\_PANIC, NULL, NULL);

for\_each\_possible\_cpu(i)

INIT\_LIST\_HEAD(&per\_cpu(blk\_cpu\_done, i));

open\_softirq(BLOCK\_SOFTIRQ, blk\_done\_softirq, NULL);

register\_hotcpu\_notifier(&blk\_cpu\_notifier);

blk\_max\_low\_pfn = max\_low\_pfn;

blk\_max\_pfn = max\_pfn;

return 0;

}

这个函数首先创建了一个名为kblockd的工作队列，这个工作队列主要是处理块设备的事件的。比如，一个块设备被拔除之后超过一定的期限并没有插回，这时候就需要将此块设备的插入标志清除，表示这个块设备目前不在系统中。然后就是对请求队列，请求，iocontext分配内存空间。

接下来这个函数对每个cpu分配一个blk\_cpu\_done链表。这个链表用于软中断，我们知道软中断贯彻了一个思想“谁触发，谁执行”（Who marks，Who runs）。中断的底半部机制中，在任意一时刻，系统只能有一个CPU可以执行Bottom Half代码，这对SMP来说是非常致命的，因为SMP完全发挥不了它应有的优势。软中断每个cpu只执行自己触发的那个中断的处理函数，这样每个cpu都可以处理自己的中断，SMP的优势有了用武之地。

open\_softirq这个函数用于将blk\_done\_softirq和软中断号BLOCK\_SOFTIRQ进行绑定。

我们再来说说blk\_cpu\_done这个链表，在request完成时，会调用函数blk\_complete\_request。这个函数将当前的request尾插到链表blk\_cpu\_done，并且触发block device软中断。软中断处理函数blk\_done\_softirq就会执行。在处理函数blk\_done\_softirq中，将request从链表blk\_cpu\_done中取下来，执行设备初始化时绑定的softirq\_done\_fn。

函数genhd\_device\_init的最后就是设置blk\_max\_low\_pfn和blk\_max\_pfn。

### subsystem\_register

函数genhd\_device\_init通过下面的语句将block系统注册到系统中。

subsystem\_register(&block\_subsys);

block\_subsys通过decl\_subsys这个宏完成初始化。

#define decl\_subsys(\_name,\_type,\_uevent\_ops) \

struct subsystem \_name##\_subsys = { \

.kset = { \

.kobj = { .name = \_\_stringify(\_name) }, \

.ktype = \_type, \

.uevent\_ops =\_uevent\_ops, \

} \

}

decl\_subsys(block, &ktype\_block, &block\_uevent\_ops);

接下来我们看看subsystem\_register这个函数到底做了什么。

int subsystem\_register(struct subsystem \* s)

{

int error;

//最终调用了kobject\_init

subsystem\_init(s);

pr\_debug("subsystem %s: registering\n",s->kset.kobj.name);

if (!(error = kset\_add(&s->kset))) {

if (!s->kset.subsys)

s->kset.subsys = s;

}

return error;

}

具体的kobject如何和sysfs关联的部分不是本文的中点，详细可见--《sysfs分析》一文。这里我们知道在/sys/下生成了block这个目录就可以了。

到此block子系统的初始化就完成了，我们回顾一下genhd\_device\_init做的工作：

1. 建立bdev\_map，bdev\_map是全局的一个hash表，用于管理所有的块设备。
2. 建立工作队列。
3. 注册软中断。
4. 分配请求队列，请求及iocontent缓冲区。
5. 初始化请求和软中断协作的blk\_cpu\_done。
6. 在/sys/下建立block这个目录。

# 块设备的注册与注销

块设备的注册通过调用函数register\_blkdev实现。

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

{

struct blk\_major\_name \*\*n, \*p;

int index, ret = 0;

mutex\_lock(&block\_subsys\_lock);

/\* temporary \*/

if (major == 0) {

//自动分配一个设备号,范围在1-254之间

for (index = ARRAY\_SIZE(major\_names)-1; index > 0; index--) {

if (major\_names[index] == NULL)

break;

}

if (index == 0) {

//如果循环完成后还没在major\_names找到合适的位置

printk("register\_blkdev: failed to get major for %s\n",

name);

ret = -EBUSY;

goto out;

}

major = index;

ret = major;

}

p = kmalloc(sizeof(struct blk\_major\_name), GFP\_KERNEL);

if (p == NULL) {

ret = -ENOMEM;

goto out;

}

p->major = major;

strlcpy(p->name, name, sizeof(p->name));

p->next = NULL;

//在手动指定major的情况下,major可能超过255,如果超过255,就使用%255后的下标

index = major\_to\_index(major);

for (n = &major\_names[index]; \*n; n = &(\*n)->next) {

//自动分配的major不会大于255, 例如设备号2和设备号257在同一链表中

if ((\*n)->major == major)

break;

}

if (!\*n)

//如果在链表中没找到major相同的项,就将p加入链表

\*n = p;

else

ret = -EBUSY;

if (ret < 0) {

printk("register\_blkdev: cannot get major %d for %s\n",

major, name);

kfree(p);

}

out:

mutex\_unlock(&block\_subsys\_lock);

return ret;

}

这个函数用于管理系统中块设备号，以防止系统中有多个设备使用同一个设备号。全局hash表major\_names共有255个backet，每个backet以主设备好决定。当传入的major为0时，这个时候系统会自动分配一个主设备号，如果成功就会将这个主设备号返回，代码中if (major == 0) 这个判断语句就是做这个事情的。主设备号找到后，接下来的任务就是以backet为链表头，将major % 255相同的设备号连成一个链表，以方便查找。笔者做过在ramdisk的驱动中将register\_blkdev这行去掉，直接给设备号赋一个没有被系统占用的值，对ramdisk没有任何影响。这说明register\_blkdev仅仅用于管理设备号，防止设备号冲突。在后续将会分析的add\_disk中调用的blk\_register\_region函数才是真正意义上的“注册”。

设备的注销是通过调用函数unregister\_blkdev实现的

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

{

struct blk\_major\_name \*\*n;

struct blk\_major\_name \*p = NULL;

int index = major\_to\_index(major);

int ret = 0;

mutex\_lock(&block\_subsys\_lock);

for (n = &major\_names[index]; \*n; n = &(\*n)->next)

if ((\*n)->major == major)

break;

//判断是什么原因结束循环的

if (!\*n || strcmp((\*n)->name, name))

//错误退出循环

ret = -EINVAL;

else {

p = \*n;

\*n = p->next;

}

mutex\_unlock(&block\_subsys\_lock);

kfree(p);

return ret;

}

有了register\_blkdev的分析，分析unregister\_blkdev就容易多了。首先这个函数通过major % 255找到全局hash表major\_names中的backet，接下来以backet为起点，遍历backet为链表头的链表，如果链表成员的major和参数major相等，说明找到了我们需要的项，然后将其释放就好。

# 五．分配请求队列

分配请求队列是通过函数blk\_alloc\_queue实现的，blk\_alloc\_queue只是一个简单的封装，实际工作通过blk\_alloc\_queue\_node完成。

request\_queue\_t \*blk\_alloc\_queue(gfp\_t gfp\_mask)

{

return blk\_alloc\_queue\_node(gfp\_mask, -1);

}

request\_queue\_t \*blk\_alloc\_queue\_node(gfp\_t gfp\_mask, int node\_id)

{

request\_queue\_t \*q;

q = kmem\_cache\_alloc\_node(requestq\_cachep, gfp\_mask, node\_id);

if (!q)

return NULL;

memset(q, 0, sizeof(\*q));

init\_timer(&q->unplug\_timer);

snprintf(q->kobj.name, KOBJ\_NAME\_LEN, "%s", "queue");

q->kobj.ktype = &queue\_ktype;

kobject\_init(&q->kobj);

q->backing\_dev\_info.unplug\_io\_fn = blk\_backing\_dev\_unplug;

q->backing\_dev\_info.unplug\_io\_data = q;

mutex\_init(&q->sysfs\_lock);

return q;

}

这个函数首先在请求队列缓冲池中分配一个请求队列，然后完成一些unplug的数据填充以及kobj的初始化。

# 六．自定义一个请求处理函数

如果块设备驱动程序中调用了blk\_queue\_make\_request，实际上就是修改了内核为请求队列提供的一个默认的make\_request\_fn函数\_\_make\_request。驱动程序如果采用这种方式，需要在调用blk\_queue\_make\_request之前，显式的调用比如blk\_alloc\_queue来为自己产生一个请求队列对象。

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

{

/\*

\* set defaults

\*/

//请求队列中最大请求个数为128

q->nr\_requests = BLKDEV\_MAX\_RQ;

//设置单个请求所能处理的最大物理段数

blk\_queue\_max\_phys\_segments(q, MAX\_PHYS\_SEGMENTS);

//设置单个请求所能处理的最大硬段数(分散-聚集DMA操作中的最大不同内存区数)

blk\_queue\_max\_hw\_segments(q, MAX\_HW\_SEGMENTS);

//make\_request\_fn在submit\_bio中会用到

q->make\_request\_fn = mfn;

q->backing\_dev\_info.ra\_pages = (VM\_MAX\_READAHEAD \* 1024) / PAGE\_CACHE\_SIZE;

q->backing\_dev\_info.state = 0;

q->backing\_dev\_info.capabilities = BDI\_CAP\_MAP\_COPY;

blk\_queue\_max\_sectors(q, SAFE\_MAX\_SECTORS);

blk\_queue\_hardsect\_size(q, 512);

blk\_queue\_dma\_alignment(q, 511);

//设置拥塞界限

blk\_queue\_congestion\_threshold(q);

q->nr\_batching = BLK\_BATCH\_REQ;

q->unplug\_thresh = 4; /\* hmm \*/

q->unplug\_delay = (3 \* HZ) / 1000; /\* 3 milliseconds \*/

if (q->unplug\_delay == 0)

q->unplug\_delay = 1;

//初始化工作队列的工作成员

INIT\_WORK(&q->unplug\_work, blk\_unplug\_work, q);

q->unplug\_timer.function = blk\_unplug\_timeout;

q->unplug\_timer.data = (unsigned long)q;

/\*

\* by default assume old behaviour and bounce for any highmem page

\*/

blk\_queue\_bounce\_limit(q, BLK\_BOUNCE\_HIGH);

blk\_queue\_activity\_fn(q, NULL, NULL);

}

这个函数对请求队列做了一系列初始化，其中最重要的是q->make\_request\_fn = mfn;这条语句。

# 七．获取gendisk对象

当设备驱动需要一个gendisk对象时，应该调用alloc\_disk函数。其代码如下：

struct gendisk \*alloc\_disk(int minors)

{

return alloc\_disk\_node(minors, -1);

}

struct gendisk \*alloc\_disk\_node(int minors, int node\_id)

{

struct gendisk \*disk;

disk = kmalloc\_node(sizeof(struct gendisk), GFP\_KERNEL, node\_id);

if (disk) {

memset(disk, 0, sizeof(struct gendisk));

if (!init\_disk\_stats(disk)) {

kfree(disk);

return NULL;

}

if (minors > 1) {

//给次分区分配空间

int size = (minors - 1) \* sizeof(struct hd\_struct \*);

disk->part = kmalloc\_node(size, GFP\_KERNEL, node\_id);

if (!disk->part) {

kfree(disk);

return NULL;

}

memset(disk->part, 0, size);

}

//如果minors <= 1就说明目前分区只有主分区

disk->minors = minors;

//加入block\_subsys,将disk的kobj.kset指向block\_subsys的kset

kobj\_set\_kset\_s(disk,block\_subsys);

kobject\_init(&disk->kobj);

rand\_initialize\_disk(disk);

}

return disk;

}

alloc\_disk的实际工作是在alloc\_disk\_node中完成的。后者先通过kmalloc\_node分配一个gendisk对象，然后为次分区分配空间来容纳各个分区的分区信息。然后修改disk->kobj.ket使其指向块子系统的kset，以便后续调用kobject\_add时将磁盘目录加入块子系统在/sys/block的目录下。

# 八．向系统添加一个块设备

void add\_disk(struct gendisk \*disk)

{

disk->flags |= GENHD\_FL\_UP;

blk\_register\_region(MKDEV(disk->major, disk->first\_minor),

disk->minors, NULL, exact\_match, exact\_lock, disk);

register\_disk(disk);

blk\_register\_queue(disk);

}

当使用add\_disk向系统加入一个块设备后，在/dev/下就有了一个块设备同名的节点。这个节点是内核数据在用户层的体现。节点的生成过程如图。首先在驱动中调用add\_disk添加块设备。然后内核通过向udev发送设备添加事件，udev接到事件后通过系统调用mknod在/dev下生成我们需要的设备节点。如图8-1：



图8-1

在函数sys\_mknodat中比较有意思的是new\_decode\_dev这个函数。这个函数负责将外部表示的设备号转换成内部使用的dev\_t。这是由于历史原因造成的。在内核开发早期，设备号使用一个16位的整数（通常是unsigned short）来表示主从设备号，主从设备号各占8位。但是后来设备号不够用了，于是就将设备号扩展成32位（通常是unsigned int型），然而这个设备号又需要兼容早期16位的设备号，于是将新的设备号划分成3部分，设备号中0-7位用于从设备号，8-19位用于主设备号，最后的20-31位用于从设备号的补充。内核内部为了使用方便，规定从设备号占0-19位，主设备号占20-31位，具体这些划分，如图8-2



图8-2

一个设备号在经历图8-1的过程后在init\_special\_inode中被设置了设备号。这样在打开这个设备时，就可以通过inode获取这个设备号。另外一点是，块设备的inode操作函数集也被赋值为def\_blk\_fops。

void init\_special\_inode(struct inode \*inode, umode\_t mode, dev\_t rdev)

{

inode->i\_mode = mode;

if (S\_ISCHR(mode)) {

inode->i\_fop = &def\_chr\_fops;

inode->i\_rdev = rdev;

} else if (S\_ISBLK(mode)) {

inode->i\_fop = &def\_blk\_fops;

inode->i\_rdev = rdev;

} else if (S\_ISFIFO(mode))

inode->i\_fop = &def\_fifo\_fops;

else if (S\_ISSOCK(mode))

inode->i\_fop = &bad\_sock\_fops;

else

printk(KERN\_DEBUG "init\_special\_inode: bogus i\_mode (%o)\n",

mode);

}

## blk\_register\_region

void blk\_register\_region(dev\_t dev, unsigned long range, struct module \*module,

struct kobject \*(\*probe)(dev\_t, int \*, void \*),

int (\*lock)(dev\_t, void \*), void \*data)

{

//在bdev\_map增加设备节点

kobj\_map(bdev\_map, dev, range, module, probe, lock, data);

}

int kobj\_map(struct kobj\_map \*domain, dev\_t dev, unsigned long range,

struct module \*module, kobj\_probe\_t \*probe,

int (\*lock)(dev\_t, void \*), void \*data)

{

//主设备的个数

unsigned n = MAJOR(dev + range - 1) - MAJOR(dev) + 1;

unsigned index = MAJOR(dev);

unsigned i;

struct probe \*p;

if (n > 255)

n = 255;

p = kmalloc(sizeof(struct probe) \* n, GFP\_KERNEL);

if (p == NULL)

return -ENOMEM;

for (i = 0; i < n; i++, p++) {

p->owner = module;

p->get = probe;

p->lock = lock;

p->dev = dev;

p->range = range;

p->data = data;

}

mutex\_lock(domain->lock);

for (i = 0, p -= n; i < n; i++, p++, index++) {

struct probe \*\*s = &domain->probes[index % 255];

while (\*s && (\*s)->range < range)

s = &(\*s)->next;

//将p插到s前面,保证range从小到大排列

p->next = \*s;

\*s = p;

}

mutex\_unlock(domain->lock);

return 0;

}

在genhd\_device\_init中已经调用kobj\_map\_init建立过bdev\_map这样一个全局的kobj\_map对象了。kobj\_map的工作就是将块设备根据设备号加入到bdev\_map这个散列表中。散列表中每一条链表中的元素根据range，也就是块设备的分区个数从小到大排序。这个参数data是从add\_disk中传来的，是一个gendisk对象。这样在后续的查找过程中就可以通过设备号获取gendisk实例。

## register\_disk

void register\_disk(struct gendisk \*disk)

{

struct block\_device \*bdev;

char \*s;

int i;

struct hd\_struct \*p;

int err;

//register\_disk中，disk->kobj.name还没有名字，必须从disk->disk\_name中copy过来

strlcpy(disk->kobj.name,disk->disk\_name,KOBJ\_NAME\_LEN);

/\* ewww... some of these buggers have / in name... \*/

s = strchr(disk->kobj.name, '/');

if (s)

\*s = '!';

//在/sys/block下生成目录disk->disk\_name

if ((err = kobject\_add(&disk->kobj)))

return;

//在/sys/block/disk->disk\_name/下创建符号连接subsystem， 指向/sys/block

disk\_sysfs\_symlinks(disk);

//在/sys/block/disk->disk\_name/下创建子目录holders,slaves， 指向/sys/block

disk\_sysfs\_add\_subdirs(disk);

/\* No minors to use for partitions \*/

if (disk->minors == 1)

goto exit;

/\* No such device (e.g., media were just removed) \*/

if (!get\_capacity(disk))

goto exit;

bdev = bdget\_disk(disk, 0);

if (!bdev)

goto exit;

/\* scan partition table, but suppress uevents \*/

bdev->bd\_invalidated = 1;

disk->part\_uevent\_suppress = 1;

err = blkdev\_get(bdev, FMODE\_READ, 0);

disk->part\_uevent\_suppress = 0;

if (err < 0)

goto exit;

blkdev\_put(bdev);

exit:

/\* announce disk after possible partitions are already created \*/

kobject\_uevent(&disk->kobj, KOBJ\_ADD);

/\* announce possible partitions \*/

for (i = 1; i < disk->minors; i++) {

p = disk->part[i-1];

if (!p || !p->nr\_sects)

continue;

kobject\_uevent(&p->kobj, KOBJ\_ADD);

}

}

这个函数有些长，但主要目的就是在/sys/和/dev/下生成块设备相关文件。首先设置disk的kobj的name，再通过kobject\_add时就在/sys/block/下生成了磁盘名称的目录。然后调用两个sysfs中的接口在磁盘目录下创建符号链接和子目录，这部分我不目前不太关心，继续向下看。当disk->minors为1的时候，说明这个磁盘只有主分区，没有次分区，函数就退出了。我们在这里做个假定，假设我们目前的磁盘驱动disk->minors = 4，并且disk->capacity = 0x1000000，即16MB。到了调用bdget\_disk的地方。bdget\_disk根据gendisk对象获取主分区的block\_device实例。函数blkdev\_get增加block device的引用计数，blkdev\_put释放其引用计数。kobject\_uevent向用户程序udev发送消息，udev就在/dev/下建立gendisk的设备文件。最后一个循环用户在/dev/下建立分区的设备文件。

### bdget\_disk

static inline struct block\_device \*bdget\_disk(struct gendisk \*disk, int index)

{

return bdget(MKDEV(disk->major, disk->first\_minor) + index);

}

struct block\_device \*bdget(dev\_t dev)

{

struct block\_device \*bdev;

struct inode \*inode;

/\*这里先在inode的哈希表中进行查找与dev设备号对应的inode，如果没找到的话，

则通过bdev伪文件系统创建bdev\_inode(包含inode和block device的结构体)\*/

inode = iget5\_locked(bd\_mnt->mnt\_sb, hash(dev),

bdev\_test, bdev\_set, &dev);

if (!inode)

return NULL;

bdev = &BDEV\_I(inode)->bdev;

if (inode->i\_state & I\_NEW) {

bdev->bd\_contains = NULL;

bdev->bd\_inode = inode;

bdev->bd\_block\_size = (1 << inode->i\_blkbits);

bdev->bd\_part\_count = 0;

bdev->bd\_invalidated = 0;

inode->i\_mode = S\_IFBLK;

inode->i\_rdev = dev;

inode->i\_bdev = bdev;

inode->i\_data.a\_ops = &def\_blk\_aops;

mapping\_set\_gfp\_mask(&inode->i\_data, GFP\_USER);

inode->i\_data.backing\_dev\_info = &default\_backing\_dev\_info;

spin\_lock(&bdev\_lock);

list\_add(&bdev->bd\_list, &all\_bdevs);

spin\_unlock(&bdev\_lock);

unlock\_new\_inode(inode);

}

return bdev;

}

MKDEV相信大家都很熟悉，就是通过主设备号和次设备号生成一个设备号，具体就是通过主设备号左移和次设备号或运算得出来的值，目前我们表示不关心。 在我们这种情况下，bdget获取的就是磁盘上第一个分区的设备号。然后调用iget5\_locked获取设备相关的inode，接着调用BDEV\_I获取block device对象。BDEV\_I实际上也只是container\_of的一个封装。这里涉及到了bdev伪文件系统，我们目前明白能通过inode找到bdev即可。

struct bdev\_inode {

struct block\_device bdev;

struct inode vfs\_inode;

};

接下来判断通过iget5\_locked获取的inode状态是否是I\_NEW，也就是判断这个inode是否是通过调用iget5\_locked生成的。如果是刚生成的，就根据当前块设备对这个inode初始化。注意还需要将这个bdev加入全局all\_bdevs链表，并解锁inode。函数最后返回block device对象。

#### iget5\_locked

在函数bdget中我们大概了解了bdget的作用，也就是通过设备号获取inode.这一小节我们来仔细分析这个函数。

struct inode \*iget5\_locked(struct super\_block \*sb, unsigned long hashval,

int (\*test)(struct inode \*, void \*),

int (\*set)(struct inode \*, void \*), void \*data)

{

struct hlist\_head \*head = inode\_hashtable + hash(sb, hashval);

struct inode \*inode;

//在高速缓存中搜索inode

inode = ifind(sb, head, test, data, 1);

if (inode)

return inode;

/\*

\* get\_new\_inode() will do the right thing, re-trying the search

\* in case it had to block at any point.

\*/

//如果调用ifind没有找到inode

return get\_new\_inode(sb, head, test, set, data);

}

通过bdget传入的参数中sb是bdev伪文件系统的super block。hashval是块设备的主设备号加上其次设备号。test函数指针是bdev\_test, set函数指针为bdev\_set。

这个函数首先根据传入的hashval和sb计算得出inode所在的链表头。然后调用ifind在高速缓存中搜索inode，如果搜索到，就返回inode，如果没有搜索到，就调用get\_new\_inode创建一个inode，然后返回这个刚创建的inode对象。需要注意的是只有调用get\_new\_inode创建的inode的flag有LOCK,NEW标志,并且加入到hashtable中去了。

##### ifind

ifind用于在缓冲中搜索inode.

static struct inode \*ifind(struct super\_block \*sb,

struct hlist\_head \*head, int (\*test)(struct inode \*, void \*),

void \*data, const int wait)

{

struct inode \*inode;

spin\_lock(&inode\_lock);

inode = find\_inode(sb, head, test, data);

if (inode) {

\_\_iget(inode);

spin\_unlock(&inode\_lock);

if (likely(wait))

//等待这个inode激活或者删除完成

wait\_on\_inode(inode);

return inode;

}

spin\_unlock(&inode\_lock);

return NULL;

}

搜索inode的工作主要在find\_inode函数中完成。find\_inode返回后，还需要inode将其\_\_I\_LOCK标志清除。

static struct inode \* find\_inode(struct super\_block \* sb, struct hlist\_head \*head, int (\*test)(struct inode \*, void \*), void \*data)

{

struct hlist\_node \*node;

struct inode \* inode = NULL;

repeat:

hlist\_for\_each (node, head) {

inode = hlist\_entry(node, struct inode, i\_hash);

if (inode->i\_sb != sb)

continue;

if (!test(inode, data))

continue;

//如果inode将要被销毁,等待inode释放完毕

if (inode->i\_state & (I\_FREEING|I\_CLEAR|I\_WILL\_FREE)) {

\_\_wait\_on\_freeing\_inode(inode);

goto repeat;

}

break;

}

return node ? inode : NULL;

}

这个函数搜索inode hash表，寻找匹配的inode，这里匹配的条件是inode->i\_sb 要和参数sb相等并且调用test返回true。这个test在我们这种情况是bdev\_test。

static int bdev\_test(struct inode \*inode, void \*data)

{

return BDEV\_I(inode)->bdev.bd\_dev == \*(dev\_t \*)data;

}

根据bdget\_disk分析，这个data就是gendisk的设备号。继续分析ifind函数，如果匹配完全通过，还需要检查inode的状态。I\_FREEING，I\_CLEAR，I\_WILL\_FREE这三种状态只有在释放inode时候会置位。如果这三中状态中任何一种被置位，说明我们幸幸苦苦找到的inode正在处于被释放的状态，我们需要等这个inode完全被释放。

##### get\_new\_inode

static struct inode \* get\_new\_inode(struct super\_block \*sb, struct hlist\_head \*head, int (\*test)(struct inode \*, void \*), int (\*set)(struct inode \*, void \*), void \*data)

{

struct inode \* inode;

inode = alloc\_inode(sb);

if (inode) {

struct inode \* old;

spin\_lock(&inode\_lock);

/\* We released the lock, so.. \*/

//根据sb和data在高速缓存中找到inode

old = find\_inode(sb, head, test, data);

if (!old) {

//在缓存中没找到inode, 才使用刚才分配的inode

if (set(inode, data))

goto set\_failed;

inodes\_stat.nr\_inodes++;

list\_add(&inode->i\_list, &inode\_in\_use);

list\_add(&inode->i\_sb\_list, &sb->s\_inodes);

hlist\_add\_head(&inode->i\_hash, head);

//inode还不能使用

inode->i\_state = I\_LOCK|I\_NEW;

spin\_unlock(&inode\_lock);

/\* Return the locked inode with I\_NEW set, the

\* caller is responsible for filling in the contents

\*/

return inode;

}

/\*

\* Uhhuh, somebody else created the same inode under

\* us. Use the old inode instead of the one we just

\* allocated.

\*/

\_\_iget(old);

spin\_unlock(&inode\_lock);

destroy\_inode(inode);

inode = old;

wait\_on\_inode(inode);

}

return inode;

set\_failed:

spin\_unlock(&inode\_lock);

destroy\_inode(inode);

return NULL;

}

接上文，如果通过ifind在高速缓存中没有搜索到inode，就会调用get\_new\_inode分配一个。这个函数首先会分配一个inode，当inode分配成功后就会再次调用find\_inode在高速缓存中搜索。如果高速缓存中还是没有搜索到符合条件的inode，这个时候就将刚分配的inode加入inode\_in\_use及sb管理的链表中，当然也需要将这个inode加入inode hash表，并且将系统中inode计数自加一。但由于inode是刚分配的，需要置位I\_LOCK|I\_NEW，后续工作将inode完全赋值完毕后才能解锁inode。如果刚才在高速缓存中搜索到了符合条件的inode，这时候就需要将分配的inode释放，因为我们已经有了符合条件的inode，只需要等待其他进程解锁这个inode。

还有一点需要注意的是，分配inode并初始化inode的过程中，需要调用bdev\_set设置bdev伪文件系统block device的设备号。

### blkdev\_get

int blkdev\_get(struct block\_device \*bdev, mode\_t mode, unsigned flags)

{

/\*

\* This crockload is due to bad choice of ->open() type.

\* It will go away.

\* For now, block device ->open() routine must \_not\_

\* examine anything in 'inode' argument except ->i\_rdev.

\*/

//将结构体初始化为0

struct file fake\_file = {};

struct dentry fake\_dentry = {};

fake\_file.f\_mode = mode;

fake\_file.f\_flags = flags;

fake\_file.f\_dentry = &fake\_dentry;

fake\_dentry.d\_inode = bdev->bd\_inode;

return do\_open(bdev, &fake\_file, BD\_MUTEX\_NORMAL);

}

实际上的工作就是增加块设备的引用计数。

#### do\_open

这个函数非常长，我打算分两次来分析。我们目前假设这个函数的调用链是register\_disk->blkdev\_get->do\_open。并且当前块设备没有被分区。通过这条调用链就初始化了gendisk的bdev对象。

1 static int

2 do\_open(struct block\_device \*bdev, struct file \*file, unsigned int subclass)

3 {

4 struct module \*owner = NULL;

5 struct gendisk \*disk;

6 int ret = -ENXIO;

7 int part;

8

9 file->f\_mapping = bdev->bd\_inode->i\_mapping;

10 lock\_kernel();

11 //函数返回后，part中保存了分区个数

12 disk = get\_gendisk(bdev->bd\_dev, &part);

13 if (!disk) {

14 unlock\_kernel();

15 bdput(bdev);

16 return ret;

17 }

18 owner = disk->fops->owner;

19

20 mutex\_lock\_nested(&bdev->bd\_mutex, subclass);

21

22 //此block device没有被打开过

23 if (!bdev->bd\_openers) {

24 bdev->bd\_disk = disk;

25 bdev->bd\_contains = bdev;

26 //没有使用fdisk之前，是不会有分区信息的，即没有sda1,sda2等设备

27 if (!part) {

28 //block device是一个磁盘

29 struct backing\_dev\_info \*bdi;

30 if (disk->fops->open) {

31 //调用磁盘的open

32 ret = disk->fops->open(bdev->bd\_inode, file);

33 if (ret)

34 //如果open失败

35 goto out\_first;

36 }

37 //disk->fops->open有可能会修改bd\_openers

38 if (!bdev->bd\_openers) {

39 //给bdev的block size 赋值。一个扇区512字节,左移9位

40 bd\_set\_size(bdev,(loff\_t)get\_capacity(disk)<<9);

41 bdi = blk\_get\_backing\_dev\_info(bdev);

42 if (bdi == NULL)

43 bdi = &default\_backing\_dev\_info;

44 bdev->bd\_inode->i\_data.backing\_dev\_info = bdi;

45 }

46 if (bdev->bd\_invalidated)

47 //重新扫描分区

48 rescan\_partitions(disk, bdev);

49 } else {

50 //是分区

51 struct hd\_struct \*p;

52 //主block device的对象

53 struct block\_device \*whole;

54 whole = bdget\_disk(disk, 0);

55 ret = -ENOMEM;

56 if (!whole)

57 goto out\_first;

58 ret = blkdev\_get\_whole(whole, file->f\_mode, file->f\_flags);

59 if (ret)

60 goto out\_first;

61 //bdev的主设备指向whole

62 bdev->bd\_contains = whole;

63 mutex\_lock\_nested(&whole->bd\_mutex, BD\_MUTEX\_WHOLE);

64 whole->bd\_part\_count++;

65 p = disk->part[part - 1];

66 bdev->bd\_inode->i\_data.backing\_dev\_info =

67 whole->bd\_inode->i\_data.backing\_dev\_info;

68 if (!(disk->flags & GENHD\_FL\_UP) || !p || !p->nr\_sects) {

69 whole->bd\_part\_count--;

70 mutex\_unlock(&whole->bd\_mutex);

71 ret = -ENXIO;

72 goto out\_first;

73 }

74 kobject\_get(&p->kobj);

75 bdev->bd\_part = p;

76 bd\_set\_size(bdev, (loff\_t) p->nr\_sects << 9);

77 mutex\_unlock(&whole->bd\_mutex);

78 }

79 } else {

80 //block device被打开过

81 put\_disk(disk);

82 module\_put(owner);

83 //如果此block device是一个主设备

84 if (bdev->bd\_contains == bdev) {

85 if (bdev->bd\_disk->fops->open) {

86 //调用具体设备的open

87 ret = bdev->bd\_disk->fops->open(bdev->bd\_inode, file);

88 if (ret)

89 goto out;

90 }

91 if (bdev->bd\_invalidated)

92 rescan\_partitions(bdev->bd\_disk, bdev);

93 } else {

94 mutex\_lock\_nested(&bdev->bd\_contains->bd\_mutex,

95 BD\_MUTEX\_PARTITION);

96 bdev->bd\_contains->bd\_part\_count++;

97 mutex\_unlock(&bdev->bd\_contains->bd\_mutex);

98 }

99 }

100 bdev->bd\_openers++;

101 mutex\_unlock(&bdev->bd\_mutex);

102 unlock\_kernel();

103 return 0;

104

105 out\_first:

106 bdev->bd\_disk = NULL;

107 bdev->bd\_inode->i\_data.backing\_dev\_info = &default\_backing\_dev\_info;

108 if (bdev != bdev->bd\_contains)

109 \_\_blkdev\_put(bdev->bd\_contains, BD\_MUTEX\_WHOLE);

110 bdev->bd\_contains = NULL;

111 put\_disk(disk);

112 module\_put(owner);

113 out:

114 mutex\_unlock(&bdev->bd\_mutex);

115 unlock\_kernel();

116 if (ret)

117 bdput(bdev);

118 return ret;

119 }

函数在第12获取了bdev所属的gendisk对象。由于目前这个bdev还没有初始化过，需要将bdev->disk指向gendisk对象。函数中也会调用gendisk的open操作函数。由于是do\_open是通过register调用的，这里的bdev->bd\_openers肯定是0,23行的if语句判断成功。bdev没有分区，故bdev->bd\_contains 指向bdev自己，并且part这个临时变量的值也是0。接着对主分区的bdev的大小设置为磁盘的容量。然后给bdev关联的inode设置备用设备信息。然后调用rescan\_partition重新扫描磁盘的分区。既然调用过了do\_open，bdev->bd\_openers需要自加，表示打开过几次了。

##### get\_gendisk

get\_gendisk通过一个设备号返回一个gendisk实例，同时将分区个数赋值给part。

实际工作由kobj\_lookup完成。

struct gendisk \*get\_gendisk(dev\_t dev, int \*part)

{

//通过dev number获取kobject

struct kobject \*kobj = kobj\_lookup(bdev\_map, dev, part);

//通过kboject获取gendisk

return kobj ? to\_disk(kobj) : NULL;

}

struct kobject \*kobj\_lookup(struct kobj\_map \*domain, dev\_t dev, int \*index)

{

struct kobject \*kobj;

struct probe \*p;

unsigned long best = ~0UL;

retry:

mutex\_lock(domain->lock);

for (p = domain->probes[MAJOR(dev) % 255]; p; p = p->next) {

struct kobject \*(\*probe)(dev\_t, int \*, void \*);

struct module \*owner;

void \*data;

//链表中的元素, range从小到大排列, p->dev应该要<=dev,

//p->dev + p->range - 1也就是最大的一个次设备号成员应该>=dev,

//初始化后p->dev就是主设备号, p->range为(-1)UL

if (p->dev > dev || p->dev + p->range - 1 < dev)

continue;

//p->range最大为~0,即0xffffffff, 故p->range - 1必须要<best

if (p->range - 1 >= best)

break;

if (!try\_module\_get(p->owner))

//模块获取失败

continue;

owner = p->owner;

data = p->data;

probe = p->get;

best = p->range - 1;

//获取次设备号

\*index = dev - p->dev;

if (p->lock && p->lock(dev, data) < 0) {

module\_put(owner);

continue;

}

mutex\_unlock(domain->lock);

kobj = probe(dev, index, data);

/\* Currently ->owner protects \_only\_ ->probe() itself. \*/

module\_put(owner);

if (kobj)

return kobj;

goto retry;

}

mutex\_unlock(domain->lock);

return NULL;

}

kobj\_lookup在bdev\_map中搜索。前面已经分析过了bdev\_map的初始化，bdev\_map中插入对象。现在就需要调用kobj\_lookup在bdev\_map中搜索了。在分析bdev\_map中插入对象的时候提到过，散列表中链表的元素是按照range成员从小到大排序的，当参数dev大于等于p->dev，并且p->dev + p->range - 1>= dev时，这个p就是我们要找的对象。当然还需要检查p的range不能超过链表中range的最大值~0UL。函数快结束的时候调用了在probe函数指针，用于获取kobject对象。这个在kboj\_map的时候就给这个函数指针赋值了，在块设备使用中这个函数实际上是exact\_match。

static struct kobject \*exact\_match(dev\_t dev, int \*part, void \*data)

{

struct gendisk \*p = data;

return &p->kobj;

}

在kobj\_map的时候data是设备号关联的gendisk。通过调用这个函数，就获取了gendisk对象的kobject。

# 九．块设备的打开

这一节我们来讨论块设备的打开过程。在上一节提到过，调用add\_disk后会在/dev下生成块设备的节点。这个节点的inode已经被赋值完毕，其操作函数就是def\_blk\_fops。在经历open系统调用->sys\_open->do\_sys\_open-> do\_filp\_open-> nameidata\_to\_filp-> \_\_dentry\_open调用链后就会使用块设备inode操作函数集中的open.

const struct file\_operations def\_blk\_fops = {

.open = blkdev\_open,

.release = blkdev\_close,

.llseek = block\_llseek,

.read = generic\_file\_read,

.write = blkdev\_file\_write,

.aio\_read = generic\_file\_aio\_read,

.aio\_write = blkdev\_file\_aio\_write,

.mmap = generic\_file\_mmap,

.fsync = block\_fsync,

.unlocked\_ioctl = block\_ioctl,

#ifdef CONFIG\_COMPAT

.compat\_ioctl = compat\_blkdev\_ioctl,

#endif

.readv = generic\_file\_readv,

.writev = generic\_file\_write\_nolock,

.sendfile = generic\_file\_sendfile,

.splice\_read = generic\_file\_splice\_read,

.splice\_write = generic\_file\_splice\_write,

};

## blkdev\_open

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

{

struct block\_device \*bdev;

int res;

/\*

\* Preserve backwards compatibility and allow large file access

\* even if userspace doesn't ask for it explicitly. Some mkfs

\* binary needs it. We might want to drop this workaround

\* during an unstable branch.

\*/

filp->f\_flags |= O\_LARGEFILE;

/\*

\* 这里为什么不用I\_BDEV的原因是调用这个函数的时候,

\* 可能在bdev fs中还没有生成过相应的节点,使用bd\_acquire更加安全

\*/

bdev = bd\_acquire(inode);

res = do\_open(bdev, filp, BD\_MUTEX\_NORMAL);

if (res)

return res;

if (!(filp->f\_flags & O\_EXCL) )

return 0;

if (!(res = bd\_claim(bdev, filp))){

return 0;

}

blkdev\_put(bdev);

return res;

}

块设备的打开就是通过blkdev\_open完成的。首先给filp->f\_flags添加O\_LARGEFILE标志，这个标志用于支持打开大文件。这里的大文件是指大小超过4GB的文件。通常使用c语言操作文件，文件内的偏移是unsigned int类型的，这样只能操作大小小于4GB的文件。通过在应用层open函数中加入O\_LARGEFILE标志，后续文件操作就可以使用64位的系统函数，这样就突破了4GB的限制。对于内核来说，我们默认就需要支持大文件，故在打开时就需要给予这个标志。

接下来遇到了bd\_acquire这个函数。需要注意的是函数blkdev\_open的参数中inode实际上是/dev这个挂载点下的inode，不是实际block device的inode。在/dev中的inode仅仅起一个标记作用，标记一个设备文件。块设备相关的inode实际上是保存在bdev pseudo-filesystem。bd\_acquire就通过dev这个tmpfs中的inode去获取bdev pseudo filesystem中的inode，继而获取block device实例。一个块设备只有在被打开时候才会有block device实例。

函数最后调用了bd\_claim，通过bdev->bd\_holder记录打开这个block device的struct file指针。

### bd\_acquire

static struct block\_device \*bd\_acquire(struct inode \*inode)

{

struct block\_device \*bdev;

spin\_lock(&bdev\_lock);

/\*

\* 如果这个设备调用过bdget在bdev中生成过bdev\_inode的节点,

\* 比如调用过register\_disk

\* 就可以通过inode->i\_bdev直接获取block\_device对象

\*/

bdev = inode->i\_bdev;

if (bdev) {

//增加引用计数

atomic\_inc(&bdev->bd\_inode->i\_count);

spin\_unlock(&bdev\_lock);

return bdev;

}

spin\_unlock(&bdev\_lock);

/\*

\* 通过设备号的信息来获取block device实例,这个设备号的传递过程:

\* 设备驱动中注册

\*->add\_disk->udev->shmem\_mknod->shmem\_get\_inode->init\_special\_inode

\*/

bdev = bdget(inode->i\_rdev);

if (bdev) {

spin\_lock(&bdev\_lock);

if (!inode->i\_bdev) {

/\*

\* We take an additional bd\_inode->i\_count for inode,

\* and it's released in clear\_inode() of inode.

\* So, we can access it via ->i\_mapping always

\* without igrab().

\*/

atomic\_inc(&bdev->bd\_inode->i\_count);

inode->i\_bdev = bdev;

inode->i\_mapping = bdev->bd\_inode->i\_mapping;

list\_add(&inode->i\_devices, &bdev->bd\_inodes);

}

spin\_unlock(&bdev\_lock);

}

return bdev;

}

这个函数就是blkdev\_open的核心部分了。这里首先通过inode->i\_bdev获取block device对象，如果不为空，就直接返回。这里要注意的是这个inode是挂载在/dev上的tmpfs中的inode。如果之前没有调用过bd\_acquire，inode->i\_bdev一定为空。就我们目前的情况，inode->i\_bdev是一个空指针。既然为空，这个函数就需要调用bdget在bdev pseudo filesystem中获取block device对象。获取block device对象后就将这个对象给/dev 中的inode->i\_bdev赋值，以免下次再次进行繁重的搜索工作。

### do\_open

这个函数就是打开工作的核心了。分析看第八章的do\_open。需要提到的一点是，在调用register\_disk后虽然会在bdev pseudo filesystem中添加bdev\_inode节点，但是在注册函数退出时会将这个节点删除。当块设备被打开的时候，节点又会创建，直到调用blkdev\_close后这个节点才会被删除。

### bd\_claim

int bd\_claim(struct block\_device \*bdev, void \*holder)

{

int res;

spin\_lock(&bdev\_lock);

/\* first decide result \*/

if (bdev->bd\_holder == holder)

//bdev已经被holder hold过一次了

res = 0; /\* already a holder \*/

else if (bdev->bd\_holder != NULL)

//bdev被其他的holder hold住了

res = -EBUSY; /\* held by someone else \*/

else if (bdev->bd\_contains == bdev)

//bdev没有被其他的holder hold住,并且是主设备

res = 0; /\* is a whole device which isn't held \*/

else if (bdev->bd\_contains->bd\_holder == bd\_claim)

//bdev的holder是NULL,并且是从设备

res = 0; /\* is a partition of a device that is being partitioned \*/

else if (bdev->bd\_contains->bd\_holder != NULL)

//主设备被其他holder hold住了,次设备也就不能被hold住

res = -EBUSY; /\* is a partition of a held device \*/

else

res = 0; /\* is a partition of an un-held device \*/

/\* now impose change \*/

if (res==0) {

/\* note that for a whole device bd\_holders

\* will be incremented twice, and bd\_holder will

\* be set to bd\_claim before being set to holder

\*/

/\*

\* 对于主设备来说,bd\_holders会自加2次,bd\_holder也会被设置两次.

\* 第一次设置为bd\_claim,第二次设置为holder(一般是struct file \*)

\*/

bdev->bd\_contains->bd\_holders ++;

bdev->bd\_contains->bd\_holder = bd\_claim;

bdev->bd\_holders++;

bdev->bd\_holder = holder;

}

spin\_unlock(&bdev\_lock);

return res;

}

这个函数使block device对象记录其持有者，并用bd\_holders记录被持有的次数。

### 从分区设备的打开

前面我们提到的都是主设备的打开操作。现在我们继续讨论从分区设备的打开操作。我们的主设备是/dev/ramhda，首先我们需要创建从分区。使用fdisk创建一个从分区设备后，在/dev/下就有了ramhda1这个分区设备。从分区设备的打开过程也是从blkdev\_open开始的，故又要开始分析blkdev\_open了。和主分区一样，从分区也会调用bd\_acquire在bdev pseudo filesystem下面生成block\_inode节点，并返回一个block device对象。真正的打开操作还是在do\_open这个函数中完成的。

这一次调用do\_open时，打开的并不是主设备而是从设备了。故在跟前面相比，这次在do\_open中的分支也不一样了。

## blk\_register\_queue

# 请求队列

