Ext2文件系统挂载过程分析

# 1. VFS挂载流程

# 2. ext2挂载

## 2.1 get\_sb

先来看看文件系统注册时，ext2的file\_system\_type。

static struct file\_system\_type ext2\_fs\_type = {

.owner = THIS\_MODULE,

.name = "ext2",

.get\_sb = ext2\_get\_sb,

.kill\_sb = kill\_block\_super,

.fs\_flags = FS\_REQUIRES\_DEV,

};

在VFS执行挂载时，会调用具体文件系统的get\_sb。对于ext2来说get\_sb就是ext2\_get\_sb。

static int ext2\_get\_sb(struct file\_system\_type \*fs\_type,

int flags, const char \*dev\_name, void \*data, struct vfsmount \*mnt)

{

return get\_sb\_bdev(fs\_type, flags, dev\_name, data, ext2\_fill\_super, mnt);

}

这里是对get\_sb\_bdev的一个简单封装。

int get\_sb\_bdev(struct file\_system\_type \*fs\_type,

int flags, const char \*dev\_name, void \*data,

int (\*fill\_super)(struct super\_block \*, void \*, int),

struct vfsmount \*mnt)

{

struct block\_device \*bdev;

struct super\_block \*s;

int error = 0;

bdev = open\_bdev\_excl(dev\_name, flags, fs\_type);

if (IS\_ERR(bdev))

return PTR\_ERR(bdev);

/\*

\* once the super is inserted into the list by sget, s\_umount

\* will protect the lockfs code from trying to start a snapshot

\* while we are mounting

\*/

mutex\_lock(&bdev->bd\_mount\_mutex);

s = sget(fs\_type, test\_bdev\_super, set\_bdev\_super, bdev);

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

if (IS\_ERR(s))

goto error\_s;

if (s->s\_root) {

if ((flags ^ s->s\_flags) & MS\_RDONLY) {

up\_write(&s->s\_umount);

deactivate\_super(s);

error = -EBUSY;

goto error\_bdev;

}

close\_bdev\_excl(bdev);

} else {

char b[BDEVNAME\_SIZE];

s->s\_flags = flags;

strlcpy(s->s\_id, bdevname(bdev, b), sizeof(s->s\_id));

sb\_set\_blocksize(s, block\_size(bdev));

error = fill\_super(s, data, flags & MS\_SILENT ? 1 : 0);

if (error) {

up\_write(&s->s\_umount);

deactivate\_super(s);

goto error;

}

s->s\_flags |= MS\_ACTIVE;

bdev\_uevent(bdev, KOBJ\_MOUNT);

}

return simple\_set\_mnt(mnt, s);

error\_s:

error = PTR\_ERR(s);

error\_bdev:

close\_bdev\_excl(bdev);

error:

return error;

}

函数的主要工作有

●通过get\_sb\_bdev获取bdev。

●通过sget获取super block。

●根据文件系统特点，调用特定的fill\_super填充sb。

●调用simple\_set\_mnt完成super block和vfsmnt的关联。

下面逐个分析get\_sb\_bdev调用的函数。

### 2.1.1 open\_bdev\_excl

涉及到块设备，后续讨论。

struct block\_device \*open\_bdev\_excl(const char \*path, int flags, void \*holder)

{

struct block\_device \*bdev;

mode\_t mode = FMODE\_READ;

int error = 0;

bdev = lookup\_bdev(path);

if (IS\_ERR(bdev))

return bdev;

if (!(flags & MS\_RDONLY))

mode |= FMODE\_WRITE;

error = blkdev\_get(bdev, mode, 0);

if (error)

return ERR\_PTR(error);

error = -EACCES;

if (!(flags & MS\_RDONLY) && bdev\_read\_only(bdev))

goto blkdev\_put;

error = bd\_claim(bdev, holder);

if (error)

goto blkdev\_put;

return bdev;

blkdev\_put:

blkdev\_put(bdev);

return ERR\_PTR(error);

}

### 2.1.2 sget

struct super\_block \*sget(struct file\_system\_type \*type,

int (\*test)(struct super\_block \*,void \*),

int (\*set)(struct super\_block \*,void \*),

void \*data)

{

struct super\_block \*s = NULL;

struct list\_head \*p;

int err;

retry:

spin\_lock(&sb\_lock);

if (test) list\_for\_each(p, &type->fs\_supers) {

struct super\_block \*old;

old = list\_entry(p, struct super\_block, s\_instances);

if (!test(old, data))

continue;

if (!grab\_super(old))

goto retry;

if (s)

destroy\_super(s);

return old;

}

if (!s) {

spin\_unlock(&sb\_lock);

s = alloc\_super(type);

if (!s)

return ERR\_PTR(-ENOMEM);

goto retry;

}

err = set(s, data);

if (err) {

spin\_unlock(&sb\_lock);

destroy\_super(s);

return ERR\_PTR(err);

}

s->s\_type = type;

strlcpy(s->s\_id, type->name, sizeof(s->s\_id));

list\_add\_tail(&s->s\_list, &super\_blocks);

list\_add(&s->s\_instances, &type->fs\_supers);

spin\_unlock(&sb\_lock);

get\_filesystem(type);

return s;

}

这个函数在《sysfs分析》一文中已经有很详细的讲解，要注意的地方是在挂载ext2文件系统时，get\_sb给这个函数传入了test和set指针。这样的话super block就优先在已经分配的sb中搜索，而不是直接分配。这样也就使一个已经格式化成ext2文件系统的块设备可以被多次挂载在不同的挂载点，并且挂载后各挂载点内容相同。

### 2.1.3 simple\_set\_mnt

int simple\_set\_mnt(struct vfsmount \*mnt, struct super\_block \*sb)

{

mnt->mnt\_sb = sb;

mnt->mnt\_root = dget(sb->s\_root);

return 0;

}

这个函数非常简单，就是给vfsmnt赋一些简单的值。

## 2.2 ext2\_fill\_super

分析get\_sb\_bdev过程中，fill\_sb这个函数指针没有分析。对于ext2文件系统来说，这个函数指针实际上是ext2\_fill\_super，这个函数是ext2挂载过程的重点。

### 2.2.1 sb\_bread

这个函数是对\_\_bread非常简单的一个封装。

static inline struct buffer\_head \*

sb\_bread(struct super\_block \*sb, sector\_t block)

{

return \_\_bread(sb->s\_bdev, block, sb->s\_blocksize);

}

struct buffer\_head \*

\_\_bread(struct block\_device \*bdev, sector\_t block, int size)

{

struct buffer\_head \*bh = \_\_getblk(bdev, block, size);

if (likely(bh) && !buffer\_uptodate(bh))

bh = \_\_bread\_slow(bh);

return bh;

}

\_\_bread函数的主要作用是通过block\_device对象，请求的块号，请求的大小，返回我们需要的buffer head。首先调用\_\_getblk去获取bh, 然后调用\_\_bread\_slow与通用块层交互，真正的读取设备中的数据到bh中去。

\_\_bread函数调用过程相当复杂，下面我们列出\_\_bread的调用关系图。



#### 2.2.1.1 \_\_getblk

/\*

\* \_\_getblk will locate (and, if necessary, create) the buffer\_head

\* which corresponds to the passed block\_device, block and size. The

\* returned buffer has its reference count incremented.

\*

\* \_\_getblk() cannot fail - it just keeps trying. If you pass it an

\* illegal block number, \_\_getblk() will happily return a buffer\_head

\* which represents the non-existent block. Very weird.

\*

\* \_\_getblk() will lock up the machine if grow\_dev\_page's try\_to\_free\_buffers()

\* attempt is failing. FIXME, perhaps?

\*/

struct buffer\_head \*

\_\_getblk(struct block\_device \*bdev, sector\_t block, int size)

{

struct buffer\_head \*bh = \_\_find\_get\_block(bdev, block, size);

might\_sleep();

if (bh == NULL)

bh = \_\_getblk\_slow(bdev, block, size);

return bh;

}

正如注释所说，\_\_getblk会通过传入的block device对象，请求的块号，及请求的大小寻找一个符合条件的buffer head对象。如果没有找到这样一个buffer head，这个函数就会创建符合条件的buffer head。

还有一点需要注意的是\_\_getblk不会失败，即使传入一个非法的块号，比如这个块号超过了block device的块的总数。它也会返回一个对应于这个不存在的块的buffer head指针。

函数的流程:

1. 通过\_\_find\_get\_block在buffer head的lru数组中搜索符合条件的buffer head。如果lru中没有找到，就去页缓存中搜索。
2. 如果\_\_find\_get\_block搜索失败，接着调用\_\_getblk\_slow再次搜索，需要注意的是\_\_getblk\_slow可能会创建符合条件的buffer head。

##### \_\_find\_get\_block

struct buffer\_head \*

\_\_find\_get\_block(struct block\_device \*bdev, sector\_t block, int size)

{

struct buffer\_head \*bh = lookup\_bh\_lru(bdev, block, size);

if (bh == NULL) {

bh = \_\_find\_get\_block\_slow(bdev, block);

if (bh)

bh\_lru\_install(bh);

}

if (bh)

touch\_buffer(bh);

return bh;

}

正如函数\_\_getblk中提到的，\_\_ find\_get\_block会去buffer head的数组，及页缓存中搜索buffer head。函数lookup\_bh\_lru的作用就是在全局的buffer head lru数组中搜索buffer head。

###### lookup\_bh\_lru

static struct buffer\_head \*

lookup\_bh\_lru(struct block\_device \*bdev, sector\_t block, int size)

{

struct buffer\_head \*ret = NULL;

struct bh\_lru \*lru;

int i;

check\_irqs\_on();

bh\_lru\_lock();

lru = &\_\_get\_cpu\_var(bh\_lrus);

for (i = 0; i < BH\_LRU\_SIZE; i++) {

struct buffer\_head \*bh = lru->bhs[i];

if (bh && bh->b\_bdev == bdev &&

bh->b\_blocknr == block && bh->b\_size == size) {

if (i) {

while (i) {

lru->bhs[i] = lru->bhs[i - 1];

i--;

}

lru->bhs[0] = bh;

}

get\_bh(bh);

ret = bh;

break;

}

}

bh\_lru\_unlock();

return ret;

}

这里要注意的是bh\_lrus是一个per\_cpu变量，也就是每个cpu均有一个变量的副本。这个变量是编译的时候定义的。

static DEFINE\_PER\_CPU(struct bh\_lru, bh\_lrus) = {{ NULL }};

通过\_\_get\_cpu\_var获取了当前cpu的bh\_lrus地址。在for循环中将bh\_lrus中的bh依次取出，并且判断这个bh所属的block device，起始逻辑块号，映射的大小是否和传入的参数相同。如果匹配成功，就将这个bh移动到lru的最前端，并且返回这buffer head对象。如果没有找到，就返回NULL。

###### \_\_find\_get\_block\_slow

/\*

\* Various filesystems appear to want \_\_find\_get\_block to be non-blocking.

\* But it's the page lock which protects the buffers. To get around this,

\* we get exclusion from try\_to\_free\_buffers with the blockdev mapping's

\* private\_lock.

\*

\* Hack idea: for the blockdev mapping, i\_bufferlist\_lock contention

\* may be quite high. This code could TryLock the page, and if that

\* succeeds, there is no need to take private\_lock. (But if

\* private\_lock is contended then so is mapping->tree\_lock).

\*/

static struct buffer\_head \*

\_\_find\_get\_block\_slow(struct block\_device \*bdev, sector\_t block)

{

struct inode \*bd\_inode = bdev->bd\_inode;

struct address\_space \*bd\_mapping = bd\_inode->i\_mapping;

struct buffer\_head \*ret = NULL;

pgoff\_t index;

struct buffer\_head \*bh;

struct buffer\_head \*head;

struct page \*page;

int all\_mapped = 1;

index = block >> (PAGE\_CACHE\_SHIFT - bd\_inode->i\_blkbits);

page = find\_get\_page(bd\_mapping, index);

if (!page)

goto out;

spin\_lock(&bd\_mapping->private\_lock);

if (!page\_has\_buffers(page))

goto out\_unlock;

head = page\_buffers(page);

bh = head;

do {

if (bh->b\_blocknr == block) {

ret = bh;

get\_bh(bh);

goto out\_unlock;

}

if (!buffer\_mapped(bh))

all\_mapped = 0;

bh = bh->b\_this\_page;

} while (bh != head);

/\* we might be here because some of the buffers on this page are

\* not mapped. This is due to various races between

\* file io on the block device and getblk. It gets dealt with

\* elsewhere, don't buffer\_error if we had some unmapped buffers

\*/

if (all\_mapped) {

//here is where i don't really understand

printk("\_\_find\_get\_block\_slow() failed. "

"block=%llu, b\_blocknr=%llu\n",

(unsigned long long)block,

(unsigned long long)bh->b\_blocknr);

printk("b\_state=0x%08lx, b\_size=%zu\n",

bh->b\_state, bh->b\_size);

printk("device blocksize: %d\n", 1 << bd\_inode->i\_blkbits);

}

out\_unlock:

spin\_unlock(&bd\_mapping->private\_lock);

page\_cache\_release(page);

out:

return ret;

}

首先还是一如既往的先看注释，注释说：

貌似，各种各样的文件系统都希望在使用\_\_find\_get\_block时，不阻塞。但是buffer是被page lock保护着的（注意page lock会引起休眠）。为了解决这个问题，在与try\_to\_free\_buffers互斥访问block device的mapping时，使用的是private\_lock（这是一个自旋锁）。

函数大体流程：

1. 通过block device指针获取其inode，进而获取inode的地址空间。
2. 根据传入的块号，计算页在地址空间的index。
3. 既然得到了page在address\_space中的index,就调用find\_get\_page获取page。
4. 如果page存在缓冲区，就遍历缓存区，如果找到一个起始块号和参数的块号相同的缓冲区，就将这个缓冲区返回，否则返回空。

###### bh\_lru\_install

static void bh\_lru\_install(struct buffer\_head \*bh)

{

struct buffer\_head \*evictee = NULL;

struct bh\_lru \*lru;

check\_irqs\_on();

bh\_lru\_lock();

lru = &\_\_get\_cpu\_var(bh\_lrus);

if (lru->bhs[0] != bh) {

struct buffer\_head \*bhs[BH\_LRU\_SIZE];

int in;

int out = 0;

get\_bh(bh);

bhs[out++] = bh;

for (in = 0; in < BH\_LRU\_SIZE; in++) {

struct buffer\_head \*bh2 = lru->bhs[in];

if (bh2 == bh) {

\_\_brelse(bh2);

} else {

if (out >= BH\_LRU\_SIZE) {

BUG\_ON(evictee != NULL);

evictee = bh2;

} else {

bhs[out++] = bh2;

}

}

}

while (out < BH\_LRU\_SIZE)

bhs[out++] = NULL;

memcpy(lru->bhs, bhs, sizeof(bhs));

}

bh\_lru\_unlock();

if (evictee)

\_\_brelse(evictee);

}

这个函数的主要作用是将参数bh加入到bh的lru数组中去,并且使bh位于lru的首部，也就是数组的第一个位置。首先，bh\_lru\_install通过\_\_get\_cpu\_var获取per-cpu变量bh\_lrus，取其地址赋值给变量lru。然后判断lru第一个成员是否和传入的bh相等，即判断bh是否已经位于lru的首部，如果不相等，就构造一个类型及长度和lru数组一致的数组bhs，并且将bh放入bhs[0], 然后将lru中所有成员copy到bhs中，最后将lru重新指向bhs完成了bh的插入。这里需要注意的是数组bhs和数组lru长度一致，bh相对于lru增加了成员bh，也就是说bhs中将不会拷贝lru数组的最后一个成员。实际确实是这个样子的，最后一个成员不但不会拷贝，而且通过函数\_\_brelse释放。

还有一点需要注意的是：可能在lru中存在多个成员与bh相同，在从lru给bhs赋值时，这些成员是不会copy到数组bhs中去的，也就回造成bhs有效成员的个数小于数组长度，这时候就需要将数组剩下的部分用0填充，这就是while循环在这里的作用。

##### \_\_getblk\_slow

static struct buffer\_head \*

\_\_getblk\_slow(struct block\_device \*bdev, sector\_t block, int size)

{

/\* Size must be multiple of hard sectorsize \*/

if (unlikely(size & (bdev\_hardsect\_size(bdev)-1) ||

(size < 512 || size > PAGE\_SIZE))) {

printk(KERN\_ERR "getblk(): invalid block size %d requested\n",

size);

printk(KERN\_ERR "hardsect size: %d\n",

bdev\_hardsect\_size(bdev));

dump\_stack();

return NULL;

}

for (;;) {

struct buffer\_head \* bh;

bh = \_\_find\_get\_block(bdev, block, size);

if (bh)

return bh;

if (!grow\_buffers(bdev, block, size))

free\_more\_memory();

}

}

前面分析函数\_\_getblk时提到过，\_\_getblk首先调用\_\_find\_get\_block在lru及页缓存中搜索符合条件的buffer head，如果没有找到就调用\_\_getblk\_slow。这个函数会创建符合条件的buffer head。创建buffer head时，对buffer head的大小有严格的要求，这就是函数一进来就对size做判断的原因。调用\_\_getblk\_slow创建的buffer head的大小有如下要求：

1. size必须以硬件的块大小对齐，也就是说，size必须是硬件块大小的整数倍。
2. size不能小于512字节及不能大于PAGE\_SIZE（对x86来说，就是4096byte）。

在for循环中，做了两件事。一件是调用我们在前面分析过的\_\_find\_get\_block函数搜索buffer head。另一件是调用grow\_buffers去创建需要的buffer head。\_\_find\_get\_block函数已经在前面分析过了，这里不再赘述。下面我们一起看看grow\_buffers。

###### grow\_buffers

static int

grow\_buffers(struct block\_device \*bdev, sector\_t block, int size)

{

struct page \*page;

pgoff\_t index;

int sizebits;

sizebits = -1;

do {

sizebits++;

} while ((size << sizebits) < PAGE\_SIZE);

index = block >> sizebits;

block = index << sizebits;

/\* Create a page with the proper size buffers.. \*/

page = grow\_dev\_page(bdev, block, index, size);

if (!page)

return 0;

unlock\_page(page);

page\_cache\_release(page);

return 1;

}

在\_\_getblk\_slow中对size做了限制，这里的do{}while循环算出了一个page中最多有多少个buffer组成。接下来的两次位移将block关于页面大小对齐。然后调用grow\_dev\_page创建buffer, 并且返回包含buffer的page。

grow\_dev\_page

static struct page \*

grow\_dev\_page(struct block\_device \*bdev, sector\_t block,

pgoff\_t index, int size)

{

struct inode \*inode = bdev->bd\_inode;

struct page \*page;

struct buffer\_head \*bh;

page = find\_or\_create\_page(inode->i\_mapping, index, GFP\_NOFS);

if (!page)

return NULL;

BUG\_ON(!PageLocked(page));

if (page\_has\_buffers(page)) {

bh = page\_buffers(page);

if (bh->b\_size == size) {

init\_page\_buffers(page, bdev, block, size);

return page;

}

if (!try\_to\_free\_buffers(page))

goto failed;

}

/\*

\* Allocate some buffers for this page

\*/

bh = alloc\_page\_buffers(page, size, 0);

if (!bh)

goto failed;

/\*

\* Link the page to the buffers and initialise them. Take the

\* lock to be atomic wrt \_\_find\_get\_block(), which does not

\* run under the page lock.

\*/

spin\_lock(&inode->i\_mapping->private\_lock);

link\_dev\_buffers(page, bh);

init\_page\_buffers(page, bdev, block, size);

spin\_unlock(&inode->i\_mapping->private\_lock);

return page;

failed:

BUG();

unlock\_page(page);

page\_cache\_release(page);

return NULL;

}

由于buffer依赖于page，所以在创建buffer之前，应该创建page。find\_or\_create\_page的作用就是根据index在页缓存中查找page，如果index对应的page不存在，就创建，最后返回一个page对象。接着判断page内是否有buffer，如果页内存在块缓存，取出第一个buffer head，对其大小进行判断，判断是否与size相等。这里只取出第一个buffer的原因是页内所有的buffer大小都是相等的，随意取一个就行。如果buffer的大小和size相等，就调用init\_page\_buffers初始化页内所有的buffer。如果不相等，就将页内所有的buffer全部释放掉，然后以size对页内buffer重新分配。

init\_page\_buffers

static void

init\_page\_buffers(struct page \*page, struct block\_device \*bdev,

sector\_t block, int size)

{

struct buffer\_head \*head = page\_buffers(page);

struct buffer\_head \*bh = head;

int uptodate = PageUptodate(page);

do {

if (!buffer\_mapped(bh)) {

init\_buffer(bh, NULL, NULL);

bh->b\_bdev = bdev;

bh->b\_blocknr = block;

if (uptodate)

set\_buffer\_uptodate(bh);

set\_buffer\_mapped(bh);

}

block++;

bh = bh->b\_this\_page;

} while (bh != head);

}

这个函数依次给buffer head的b\_bdev和b\_blocknr赋值。buffer是否uptodate是根据buffer所处的page是否uptodate决定的。一旦buffer head的b\_bdev和b\_blocknr赋值后就认为buffer已经被map过了。init\_buffer仅仅是将buffer head的b\_end\_io和b\_private赋值为NULL罢了。

try\_to\_free\_buffers

int try\_to\_free\_buffers(struct page \*page)

{

struct address\_space \* const mapping = page->mapping;

struct buffer\_head \*buffers\_to\_free = NULL;

int ret = 0;

BUG\_ON(!PageLocked(page));

if (PageWriteback(page))

return 0;

if (mapping == NULL) { /\* can this still happen? \*/

ret = drop\_buffers(page, &buffers\_to\_free);

goto out;

}

spin\_lock(&mapping->private\_lock);

ret = drop\_buffers(page, &buffers\_to\_free);

if (ret) {

/\*

\* If the filesystem writes its buffers by hand (eg ext3)

\* then we can have clean buffers against a dirty page. We

\* clean the page here; otherwise later reattachment of buffers

\* could encounter a non-uptodate page, which is unresolvable.

\* This only applies in the rare case where try\_to\_free\_buffers

\* succeeds but the page is not freed.

\*/

clear\_page\_dirty(page);

}

spin\_unlock(&mapping->private\_lock);

out:

if (buffers\_to\_free) {

struct buffer\_head \*bh = buffers\_to\_free;

do {

struct buffer\_head \*next = bh->b\_this\_page;

free\_buffer\_head(bh);

bh = next;

} while (bh != buffers\_to\_free);

}

return ret;

}

释放页内buffer的函数比较有意思，我们可以继续关注下。释放buffers的时候牵扯到一个问题，如果page中的buffer全部被释放后，page所在在地址空间中的tag需要修改。page并不关联任何的地址空间，地址空间中的tag就不需要修改，buffer就能被直接释放掉。这就是在函数一开始判断mapping是否为NULLd的原因。

正如我们上面提到的try\_to\_free\_buffers会对page是否关联address space两种情况分别处理。当page没有关联address space的时候，调用drop\_buffers解除page和buffer head之间的关联，然后跳转到out将所有的buffer head全部释放。

当page关联address space的时候，这种情况比上一种情况多调用了clear\_page\_dirty来修改address space中的tag。

drop\_buffers

static int

drop\_buffers(struct page \*page, struct buffer\_head \*\*buffers\_to\_free)

{

struct buffer\_head \*head = page\_buffers(page);

struct buffer\_head \*bh;

bh = head;

do {

if (buffer\_write\_io\_error(bh) && page->mapping)

set\_bit(AS\_EIO, &page->mapping->flags);

if (buffer\_busy(bh))

goto failed;

bh = bh->b\_this\_page;

} while (bh != head);

do {

struct buffer\_head \*next = bh->b\_this\_page;

if (!list\_empty(&bh->b\_assoc\_buffers))

\_\_remove\_assoc\_queue(bh);

bh = next;

} while (bh != head);

\*buffers\_to\_free = head;

\_\_clear\_page\_buffers(page);

return 1;

failed:

return 0;

}

这个函数的主题流程分为3步：

1. 逐个检查page中的块缓存是否处于busy的状态，如果buffer busy, 就返回。
2. 依次将buffer head从address space的链表中移除。struct buffer\_head中，有一个成员b\_assoc\_buffers，这个成员一般会与inode关联起来。如果buffer head已经连接到inode的private list中去了，那在释放这个buffer的时候，需要将buffer head从链表中移除。如果没有在inode的private list中，这个buffer head就可以直接释放。



1. 将page->private清空。page->private指向了块缓存的第一个成员，将private置空后，表示这个page中没有任何的buffer。

alloc\_page\_buffers

struct buffer\_head \*alloc\_page\_buffers(struct page \*page, unsigned long size,

int retry)

{

struct buffer\_head \*bh, \*head;

long offset;

try\_again:

head = NULL;

offset = PAGE\_SIZE;

while ((offset -= size) >= 0) {

bh = alloc\_buffer\_head(GFP\_NOFS);

if (!bh)

goto no\_grow;

bh->b\_bdev = NULL;

bh->b\_this\_page = head;

bh->b\_blocknr = -1;

head = bh;

bh->b\_state = 0;

atomic\_set(&bh->b\_count, 0);

bh->b\_private = NULL;

bh->b\_size = size;

/\* Link the buffer to its page \*/

set\_bh\_page(bh, page, offset);

init\_buffer(bh, NULL, NULL);

}

return head;

/\*

\* In case anything failed, we just free everything we got.

\*/

no\_grow:

if (head) {

do {

bh = head;

head = head->b\_this\_page;

free\_buffer\_head(bh);

} while (head);

}

/\*

\* Return failure for non-async IO requests. Async IO requests

\* are not allowed to fail, so we have to wait until buffer heads

\* become available. But we don't want tasks sleeping with

\* partially complete buffers, so all were released above.

\*/

if (!retry)

return NULL;

/\* We're \_really\_ low on memory. Now we just

\* wait for old buffer heads to become free due to

\* finishing IO. Since this is an async request and

\* the reserve list is empty, we're sure there are

\* async buffer heads in use.

\*/

free\_more\_memory();

goto try\_again;

}

这个函数用于在page内分配buffer。在while循环内首先分配buffer head，然后设置buffer head的各项成员。最终会分配PAGE\_SIZE/size个buffer。这些buffer组成一个单向链表，链表头在函数结束时返回。如果在分配buffer head时失败，就会跳转到标签no\_grow中，将之前分配的buffer head全部清除。这个函数的retry参数用于控制函数在buffer head分配失败的时候是否会再次分配。例如在同步的io请求调用这个函数时，允许分配失败，这个时候retry参数应该设置为0，如果在异步的io请求过程中调用此函数，是不允许分配失败的，这时候，retry应该设置为1。

link\_dev\_buffers

static inline void

link\_dev\_buffers(struct page \*page, struct buffer\_head \*head)

{

struct buffer\_head \*bh, \*tail;

bh = head;

do {

tail = bh;

bh = bh->b\_this\_page;

} while (bh);

tail->b\_this\_page = head;

attach\_page\_buffers(page, head);

}

在函数alloc\_page\_buffers执行完毕后page关联的buffer head还只是单向链表，在执行link\_dev\_buffers是，需要将这个单向链表的尾和头连起来，组成单向循环链表。attach\_page\_buffers，设置page的private指针指向buffer head的首成员。

#### \_\_bread\_slow

static struct buffer\_head \*\_\_bread\_slow(struct buffer\_head \*bh)

{

lock\_buffer(bh);

if (buffer\_uptodate(bh)) {

unlock\_buffer(bh);

return bh;

} else {

get\_bh(bh);

bh->b\_end\_io = end\_buffer\_read\_sync;

submit\_bh(READ, bh);

wait\_on\_buffer(bh);

if (buffer\_uptodate(bh))

return bh;

}

brelse(bh);

return NULL;

}

在分析这个函数时，还是需要看看这个函数在\_\_bread中被调用的上下文。

struct buffer\_head \*bh = \_\_getblk(bdev, block, size);

if (likely(bh) && !buffer\_uptodate(bh))

bh = \_\_bread\_slow(bh);

虽然通过\_\_getblk获取了buffer head，这些buffer已经建立好了映射，但是如果这些buffer中的内容和磁盘上的不一致，也就是我们所说的not uptodate，这时候就需要将磁盘上的内容更新到buffer中。这就是上面这个代码段的意义。在if第一个分支，判断buffer head是否uptodate，如果已经uptodate了，解除buffer head的锁定，返回已经最新的buffer head。if第二个分支，当然是buffer head不是uptodate的情况，这个时候向通用块层提交请求，然后等待请求执行完成，解锁buffer head，再次判断buffer head是否为uptodate，如果是，就成功的返回buffer head。如果未否，说明buffer head获取失败，将释放buffer head。