**页高速缓存**

本文主要基于Linux4.4内核，分析了内核页高速缓存的实现。除了讲述页高速缓存的实现，本文的重点仍然脱离不了文件系统，文章会着重分析页高速缓存和文件系统之间的关系。另外本文中的部分内容会涉及内存管理、线性地址和物理地址、进程地址空间、通用块层等其它内核子系统的部分知识，整个文章是通过read系统调用串起来，可以说这是文章的主线。另外read系统调用必须在已经打开的文件上进行（即执行了open系统调用），关于这部分的知识请参考其它三份总结文档。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | 版本 | 修改内容简介 | 修改日期 | 修改人 |
| 1 | V1.0 | 页高速缓存分析完成 | 2016-04-14 | 李小龙 |

[一、 页高速缓存 3](#_Toc1281066686)

[1、 基树 3](#_Toc171731423)

[2、 内核数据结构address\_space 4](#_Toc547836637)

[3、 查找和增加页 6](#_Toc1671186843)

[二、 页高速缓存和文件读操作 7](#_Toc289058538)

[1、 sys\_read函数 7](#_Toc1559703822)

[2、 sys\_read函数的处理流程 9](#_Toc1892378302)

[3、 内核数据结构page 21](#_Toc358161320)

[4、 copy\_page\_to\_iter()函数的处理流程 27](#_Toc382844295)

[5、 kmap\_atomic()函数的定义跟踪 32](#_Toc839904562)

[6、 page到线性地址的转换 36](#_Toc1525696539)

[三、 页高速缓存和块缓冲区 42](#_Toc2140668321)

[四、 readpage方法 43](#_Toc1621586741)

[1、 普通文件的readpage方法 43](#_Toc1073379413)

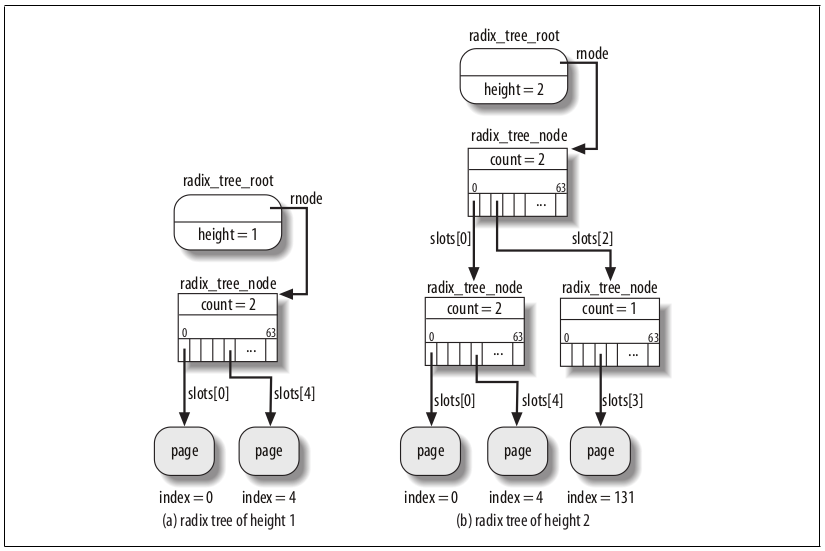
[2、 块设备文件的readpage方法 61](#_Toc1247253595)

1. **页高速缓存**

内核在读写磁盘文件时大部分情况下都会使用到页高速缓存，当所读取的文件不在磁盘高速缓存中时，首先会分配一个新页给文件对象，再将文件从磁盘读入页高速缓存，然后在从页高速缓存复制到用户空间的缓冲区。同理，在进行写操作时，如果页高速缓存不存在，则新分配一个页给文件，然后将数据从用户地址空间的缓冲区复制到页高速缓存，之后再将数据从也高速缓存写入到磁盘。

1. **基树**

对页高速缓存的管理采用一种称之为基树的数据结构，如下图（引用自《深入理解Linux内核》）：



radix\_tree\_root是基数的根节点，其中保存有树的深度height（不包括叶子节点），以及指向中间节点radix\_tree\_node的rnode。radix\_tree\_node的slots[]是一个void \*数组，每个元素指向下一个节点，叶子节点是页描述符page，该结构描述了物理内存（页框）的使用情况，index字段表示该页在文件中以页大小位单位的偏移量。

1. **内核数据结构address\_space**

对页高速缓存进行管理的最重要的内核数据结构就是address\_space，内核文件结构file和索引节点inode都有一个字段指向address\_space结构（file字段的f\_mapping和inode字段的i\_mapping），在创建file对象时将会用inode节点的i\_mapping字段给f\_mapping字段赋值。address\_space结构如下：

struct address\_space {

struct inode \*host; /\* owner: inode, block\_device \*/

struct radix\_tree\_root page\_tree; /\* radix tree of all pages \*/

spinlock\_t tree\_lock; /\* and lock protecting it \*/

atomic\_t i\_mmap\_writable;/\* count VM\_SHARED mappings \*/

struct rb\_root i\_mmap; /\* tree of private and shared mappings \*/

struct rw\_semaphore i\_mmap\_rwsem; /\* protect tree, count, list \*/

/\* Protected by tree\_lock together with the radix tree \*/

unsigned long nrpages; /\* number of total pages \*/

unsigned long nrshadows; /\* number of shadow entries \*/

pgoff\_t writeback\_index;/\* writeback starts here \*/

const struct address\_space\_operations \*a\_ops; /\* methods \*/

unsigned long flags; /\* error bits/gfp mask \*/

spinlock\_t private\_lock; /\* for use by the address\_space \*/

struct list\_head private\_list; /\* ditto \*/

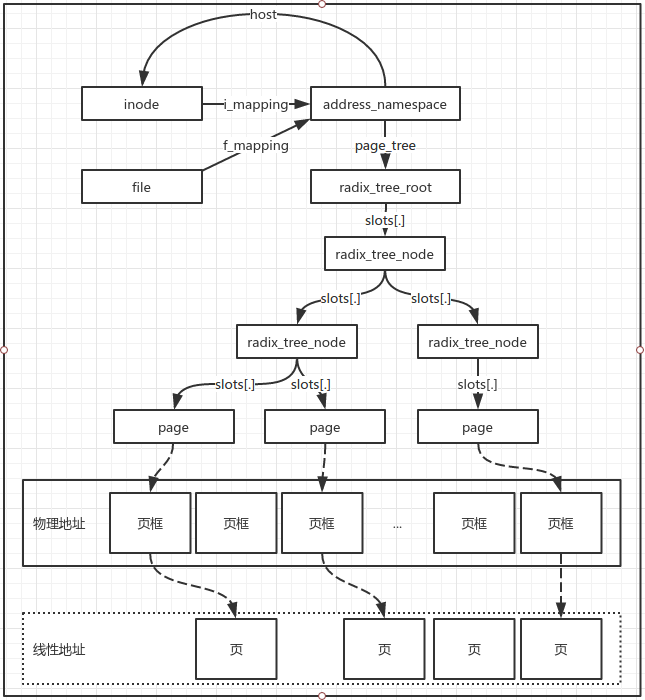
void \*private\_data; /\* ditto \*/

} \_\_attribute\_\_((aligned(sizeof(long))));

其中host字段指向拥有该address\_space结构的inode节点，page\_tree字段指向基树的根，nrpages字段表示基树中页page的个数。至此可以用下图来描述这些数据结构的关系。

下图看起来似乎很明确，但是其中有许多地方值得我们深入，最主要的一点就是：page是一个页描述符，它不是页表项，结构中也不包含任何关于线性地址和物理地址的相关字段。那么问题就来了，当我们读取文件时可以通过index找到该page描述符，但系统任然无从得知要访问的线性地址（cpu大多数寄存器保存的是线性地址）。那么系统是如何得到要访问文件的线性地址的？这个问题将在文章的后一部分做详尽阐述。在此先对下图做一个简要的说明。

打开一个文件时就会创建一个file结构，并且用对应的inode字段的一些值设置相关字段，其中就包括f\_mapping。他们都指向ddress\_namespace结构，该结构在创建inode节点时创建，page\_tree字段指向上面所将的基树。系统通过index找到基树中给定index的page。再通过该page得到物理页框号，然后在将页框号转换位对应的线性地址。至此就可以用线性地址访问文件内容。



1. **查找和增加页**

对页高速缓存的管理采用基树，那么对页进行查找其实就是在基树中查找符合条件的page页节点。当所访问的文件内容不在页高速缓存中时就会在相应的基树中添加一个页。

①、页查找

假设基树中除页节点外每个节点有64个孩子，即slots[]数组长度为64，所以每层最大索引位64n-1(n为深度）例如我们要查找index为131的页，131写成二进制是10000011，右移6位为二进制10，所以目标保存在第一层的slots[2]中，余数是3所以在slots[2]所指子树的slots[3]中保存有查找的page。内核中负责查找页的主要函数是find\_get\_page()，该函数接受address\_space对象和页偏移，返回找到的页的地址或者返回NULL。

②、增加页

当所需要的页不在高速缓存时就会分配一个新页page，然后将该页添加到基树中，如果是读操作，还会从磁盘读入相应的文件数据。内核用page\_cache\_alloc\_cold()函数分配一个新页，然后调用add\_to\_page\_cache\_lru()函数将刚分配的新页加入到基树中。

1. **页高速缓存和文件读操作**

本节主要从文件度操作来分析页高速缓存的管理，基于linux-4.4内核，x86\_64架构。主要涉及页高速缓存的查找，增加，以及之前所提到的如何将page转换为线性地址。

1. **sys\_read函数**

在linux-4.4内核中x86架构32位和64位的架构相关代码都在同一个文件夹arch/x86下面。应该是通过相应的宏来控制生成的代码是64位还是32位的。arch/x86/entry/syscalls目录下面的文件是与系统调用号有关的，分析该文件夹下的文件我们能够确定read系统调用的调用号是0（x86\_64），对应的系统调用函数名称是sys\_read。我们从此入手分析read系统调用。

Linux-4.4内核相对linux-2.6.18内核的一个系统调用函数结构的变化由sys\_read变成了一个宏函数：

SYSCALL\_DEFINE3(read, unsigned int, fd, char \_\_user \*, buf, size\_t, count)

{

struct fd f = fdget\_pos(fd);

ssize\_t ret = -EBADF;

if (f.file) {

loff\_t pos = file\_pos\_read(f.file);

ret = vfs\_read(f.file, buf, count, &pos);

if (ret >= 0)

file\_pos\_write(f.file, pos);

fdput\_pos(f);

}

return ret;

}

上述宏函数其实就是sys\_read函数，我们可以将相关的代码抽取出来，然后使用gcc的预处理选项-E查看预处理后的函数原形如下（添加了一些换行便于阅读）：

asmlinkage long sys\_read(\_\_MAP3(\_\_SC\_DECL,unsigned int, fd, char \_\_user \*, buf, size\_t, count)) \_\_attribute\_\_((alias(\_\_stringify(SyS\_read))));

static inline long SYSC\_read(\_\_MAP3(\_\_SC\_DECL,unsigned int, fd, char \_\_user \*, buf, size\_t, count));

asmlinkage long SyS\_read(\_\_MAP3(\_\_SC\_LONG,unsigned int, fd, char \_\_user \*, buf, size\_t, count));

asmlinkage long SyS\_read(\_\_MAP3(\_\_SC\_LONG,unsigned int, fd, char \_\_user \*, buf, size\_t, count))

{

long ret = SYSC\_read(\_\_MAP3(\_\_SC\_CAST,unsigned int, fd, char \_\_user \*, buf, size\_t, count));

\_\_MAP3(\_\_SC\_TEST,unsigned int, fd, char \_\_user \*, buf, size\_t, count);

asmlinkage\_protect(3, ret,\_\_MAP3(\_\_SC\_ARGS,unsigned int, fd, char \_\_user \*, buf, size\_t, count));

return ret;

}

static inline long SYSC\_read(\_\_MAP3(\_\_SC\_DECL,unsigned int, fd, char \_\_user \*, buf, size\_t, count))

{

struct fd f = fdget\_pos(fd);

ssize\_t ret = -EBADF;

if (f.file) {

loff\_t pos = file\_pos\_read(f.file);

ret = vfs\_read(f.file, buf, count, &pos);

if (ret >= 0)

file\_pos\_write(f.file, pos);

fdput\_pos(f);

}

return ret;

}

由此可知宏函数的作用就是生成内部函数SyS\_read，并定义了一个外部别名sys\_read。

1. **sys\_read函数的处理流程**

从上面的代码可以看出sys\_read函数的操作：

①、根据文件描述符获取一个fd结构，该结构包含文件对象;

struct fd {

struct file \*file;

unsigned int flags;

};

②、得到文件指针pos;

③、调用vfs\_read函数进行文件读操作；

④、移动文件指针；

⑤、释放fd结构；

再来看看vfs\_read函数，该函数的主要操作在调用\_\_vfs\_read()函数

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

{

ssize\_t ret;

if (!(file->f\_mode & FMODE\_READ))

return -EBADF;

if (!(file->f\_mode & FMODE\_CAN\_READ))

return -EINVAL;

if (unlikely(!access\_ok(VERIFY\_WRITE, buf, count)))

return -EFAULT;

ret = rw\_verify\_area(READ, file, pos, count);

if (ret >= 0) {

count = ret;

ret = \_\_vfs\_read(file, buf, count, pos);

if (ret > 0) {

fsnotify\_access(file);

add\_rchar(current, ret);

}

inc\_syscr(current);

}

return ret;

}

\_\_vfs\_read()函数，该函数根据f\_op函数列表进行不同的操作

ssize\_t \_\_vfs\_read(struct file \*file, char \_\_user \*buf, size\_t count,

loff\_t \*pos)

{

if (file->f\_op->read)

return file->f\_op->read(file, buf, count, pos);

else if (file->f\_op->read\_iter)

return new\_sync\_read(file, buf, count, pos);

else

return -EINVAL;

}

以ext2为例来查看f\_op字段，可以确定调用的是new\_sync\_read()函数。

const struct file\_operations ext2\_file\_operations = {

.llseek = generic\_file\_llseek,

.read\_iter = generic\_file\_read\_iter,

.write\_iter = generic\_file\_write\_iter,

.unlocked\_ioctl = ext2\_ioctl,

#ifdef CONFIG\_COMPAT

.compat\_ioctl = ext2\_compat\_ioctl,

#endif

.mmap = ext2\_file\_mmap,

.open = dquot\_file\_open,

.release = ext2\_release\_file,

.fsync = ext2\_fsync,

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

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

};

new\_sync\_read函数实现如下，该函数定义了三个结构体变量，类型分别位iovec、kiocb以及iov\_iter，iov\_iter中包含iovec结构体信息。Iovec结构体包含用户态buffer指针和要读取的字节数，kiocb包含文件指针pos，函数最后要改变\*ppos的值，从而在sys\_read的第④步设置file->pos。

static ssize\_t new\_sync\_read(struct file \*filp, char \_\_user \*buf, size\_t len, loff\_t \*ppos)

{

struct iovec iov = { .iov\_base = buf, .iov\_len = len };

struct kiocb kiocb;

struct iov\_iter iter;

ssize\_t ret;

init\_sync\_kiocb(&kiocb, filp);

kiocb.ki\_pos = \*ppos;

iov\_iter\_init(&iter, READ, &iov, 1, len);

ret = filp->f\_op->read\_iter(&kiocb, &iter);

BUG\_ON(ret == -EIOCBQUEUED);

\*ppos = kiocb.ki\_pos;

return ret;

}

由上可知f\_op->read\_iter()函数在ext2文件系统中为generic\_file\_read\_iter()，从名字来看，该函数是一个通用函数，大多数文件系统应该依赖该函数实现读操作。函数实现如下，generic\_file\_read\_iter()函数的主要工作有，判断文件的打开方式是不是直接IO传送，如果是则先将赃页写回磁盘，然后调用driect\_IO()读取。如果不是直接IO传送调用do\_generic\_file\_read()函数，在这里我们主要查看后面一种情况。

/\*\*

\* generic\_file\_read\_iter - generic filesystem read routine

\* @iocb: kernel I/O control block

\* @iter: destination for the data read

\*

\* This is the "read\_iter()" routine for all filesystems

\* that can use the page cache directly.

\*/

ssize\_t

generic\_file\_read\_iter(struct kiocb \*iocb, struct iov\_iter \*iter)

{

struct file \*file = iocb->ki\_filp;

ssize\_t retval = 0;

loff\_t \*ppos = &iocb->ki\_pos;

loff\_t pos = \*ppos;

if (iocb->ki\_flags & IOCB\_DIRECT) {

struct address\_space \*mapping = file->f\_mapping;

struct inode \*inode = mapping->host;

size\_t count = iov\_iter\_count(iter);

loff\_t size;

if (!count)

goto out; /\* skip atime \*/

size = i\_size\_read(inode);

retval = filemap\_write\_and\_wait\_range(mapping, pos,

pos + count - 1);

if (!retval) {

struct iov\_iter data = \*iter;

retval = mapping->a\_ops->direct\_IO(iocb, &data, pos);

}

if (retval > 0) {

\*ppos = pos + retval;

iov\_iter\_advance(iter, retval);

}

/\*

\* Btrfs can have a short DIO read if we encounter

\* compressed extents, so if there was an error, or if

\* we've already read everything we wanted to, or if

\* there was a short read because we hit EOF, go ahead

\* and return. Otherwise fallthrough to buffered io for

\* the rest of the read. Buffered reads will not work for

\* DAX files, so don't bother trying.

\*/

if (retval < 0 || !iov\_iter\_count(iter) || \*ppos >= size ||

IS\_DAX(inode)) {

file\_accessed(file);

goto out;

}

}

retval = do\_generic\_file\_read(file, ppos, iter, retval);

out:

return retval;

}

do\_generic\_file\_read()函数是一个超过200行的大函数，所以比较阅读难度较大。

该函数主要包含一个大循环，在循环之前则是计算文件指针所处位置的页的页索引index，以及在页中的偏移offset。

for循环中我们主要看几个goto语句的标志：

①、find\_page：很明显是查找页框描述符page。

②、page\_ok：意思应该是找到了相应的页，并且数据已经读入页高速缓存。

③、readpage：找到了page但是数据并没有读入页高速缓存时，跳转到此从磁盘读入页高速缓存中。

④、no\_cached\_page：基树中没有找到page描述符（slots[.]==NULL），则跳转到此分配一个新页，并将新页加入到基树中。

/\*\*

\* do\_generic\_file\_read - generic file read routine

\* @filp: the file to read

\* @ppos: current file position

\* @iter: data destination

\* @written: already copied

\*

\* This is a generic file read routine, and uses the

\* mapping->a\_ops->readpage() function for the actual low-level stuff.

\*

\* This is really ugly. But the goto's actually try to clarify some

\* of the logic when it comes to error handling etc.

\*/

static ssize\_t do\_generic\_file\_read(struct file \*filp, loff\_t \*ppos,

struct iov\_iter \*iter, ssize\_t written)

{

struct address\_space \*mapping = filp->f\_mapping;

struct inode \*inode = mapping->host;

struct file\_ra\_state \*ra = &filp->f\_ra;

pgoff\_t index;

pgoff\_t last\_index;

pgoff\_t prev\_index;

unsigned long offset; /\* offset into pagecache page \*/

unsigned int prev\_offset;

int error = 0;

index = \*ppos >> PAGE\_CACHE\_SHIFT;

prev\_index = ra->prev\_pos >> PAGE\_CACHE\_SHIFT;

prev\_offset = ra->prev\_pos & (PAGE\_CACHE\_SIZE-1);

last\_index = (\*ppos + iter->count + PAGE\_CACHE\_SIZE-1) >> PAGE\_CACHE\_SHIFT;

offset = \*ppos & ~PAGE\_CACHE\_MASK;

for (;;) {

struct page \*page;

pgoff\_t end\_index;

loff\_t isize;

unsigned long nr, ret;

cond\_resched();

find\_page:

page = find\_get\_page(mapping, index);

if (!page) {

page\_cache\_sync\_readahead(mapping,

ra, filp,

index, last\_index - index);

page = find\_get\_page(mapping, index);

if (unlikely(page == NULL))

goto no\_cached\_page;

}

if (PageReadahead(page)) {

page\_cache\_async\_readahead(mapping,

ra, filp, page,

index, last\_index - index);

}

if (!PageUptodate(page)) {

if (inode->i\_blkbits == PAGE\_CACHE\_SHIFT ||

!mapping->a\_ops->is\_partially\_uptodate)

goto page\_not\_up\_to\_date;

if (!trylock\_page(page))

goto page\_not\_up\_to\_date;

/\* Did it get truncated before we got the lock? \*/

if (!page->mapping)

goto page\_not\_up\_to\_date\_locked;

if (!mapping->a\_ops->is\_partially\_uptodate(page,

offset, iter->count))

goto page\_not\_up\_to\_date\_locked;

unlock\_page(page);

}

page\_ok:

/\*

\* i\_size must be checked after we know the page is Uptodate.

\*

\* Checking i\_size after the check allows us to calculate

\* the correct value for "nr", which means the zero-filled

\* part of the page is not copied back to userspace (unless

\* another truncate extends the file - this is desired though).

\*/

isize = i\_size\_read(inode);

end\_index = (isize - 1) >> PAGE\_CACHE\_SHIFT;

if (unlikely(!isize || index > end\_index)) {

page\_cache\_release(page);

goto out;

}

/\* nr is the maximum number of bytes to copy from this page \*/

nr = PAGE\_CACHE\_SIZE;

if (index == end\_index) {

nr = ((isize - 1) & ~PAGE\_CACHE\_MASK) + 1;

if (nr <= offset) {

page\_cache\_release(page);

goto out;

}

}

nr = nr - offset;

/\* If users can be writing to this page using arbitrary

\* virtual addresses, take care about potential aliasing

\* before reading the page on the kernel side.

\*/

if (mapping\_writably\_mapped(mapping))

flush\_dcache\_page(page);

/\*

\* When a sequential read accesses a page several times,

\* only mark it as accessed the first time.

\*/

if (prev\_index != index || offset != prev\_offset)

mark\_page\_accessed(page);

prev\_index = index;

/\*

\* Ok, we have the page, and it's up-to-date, so

\* now we can copy it to user space...

\*/

ret = copy\_page\_to\_iter(page, offset, nr, iter);

offset += ret;

index += offset >> PAGE\_CACHE\_SHIFT;

offset &= ~PAGE\_CACHE\_MASK;

prev\_offset = offset;

page\_cache\_release(page);

written += ret;

if (!iov\_iter\_count(iter))

goto out;

if (ret < nr) {

error = -EFAULT;

goto out;

}

continue;

page\_not\_up\_to\_date:

/\* Get exclusive access to the page ... \*/

error = lock\_page\_killable(page);

if (unlikely(error))

goto readpage\_error;

page\_not\_up\_to\_date\_locked:

/\* Did it get truncated before we got the lock? \*/

if (!page->mapping) {

unlock\_page(page);

page\_cache\_release(page);

continue;

}

/\* Did somebody else fill it already? \*/

if (PageUptodate(page)) {

unlock\_page(page);

goto page\_ok;

}

readpage:

/\*

\* A previous I/O error may have been due to temporary

\* failures, eg. multipath errors.

\* PG\_error will be set again if readpage fails.

\*/

ClearPageError(page);

/\* Start the actual read. The read will unlock the page. \*/

error = mapping->a\_ops->readpage(filp, page);

if (unlikely(error)) {

if (error == AOP\_TRUNCATED\_PAGE) {

page\_cache\_release(page);

error = 0;

goto find\_page;

}

goto readpage\_error;

}

if (!PageUptodate(page)) {

error = lock\_page\_killable(page);

if (unlikely(error))

goto readpage\_error;

if (!PageUptodate(page)) {

if (page->mapping == NULL) {

/\*

\* invalidate\_mapping\_pages got it

\*/

unlock\_page(page);

page\_cache\_release(page);

goto find\_page;

}

unlock\_page(page);

shrink\_readahead\_size\_eio(filp, ra);

error = -EIO;

goto readpage\_error;

}

unlock\_page(page);

}

goto page\_ok;

readpage\_error:

/\* UHHUH! A synchronous read error occurred. Report it \*/

page\_cache\_release(page);

goto out;

no\_cached\_page:

/\*

\* Ok, it wasn't cached, so we need to create a new

\* page..

\*/

page = page\_cache\_alloc\_cold(mapping);

if (!page) {

error = -ENOMEM;

goto out;

}

error = add\_to\_page\_cache\_lru(page, mapping, index,

mapping\_gfp\_constraint(mapping, GFP\_KERNEL));

if (error) {

page\_cache\_release(page);

if (error == -EEXIST) {

error = 0;

goto find\_page;

}

goto out;

}

goto readpage;

}

out:

ra->prev\_pos = prev\_index;

ra->prev\_pos <<= PAGE\_CACHE\_SHIFT;

ra->prev\_pos |= prev\_offset;

\*ppos = ((loff\_t)index << PAGE\_CACHE\_SHIFT) + offset;

file\_accessed(filp);

return written ? written : error;

}

为了便于分析，我们假设要访问的文件内容的已经被读入到页高速缓存中，而且内容是最新的，即他进程没有修改过该文件。那么该函数的工作流程就是：

①、计算页index和页内偏移offset;

②、进入for循环（因为读取的数据可能跨越几个不同的页），从高速缓存中读入指定字节的数据；

③、在for循环中调用find\_get\_page()函数查找page；

④、计算本次循环要读取的字节数；

⑤、调用copy\_page\_to\_iter()函数将数据从内核地址空间的页高速缓存复制到用户态缓存中；

⑥、修改要读取的剩余字节数、偏移、下一页的index；

⑥、如果没有读完，且不是最后一页跳转到for循环的开始处，否则跳转到out然后返回。

1. **内核数据结构page**

page结构体定义如下，该结构体的定义比较复杂，但我们不用仔细研究每个字段的含义，我们只关注有无和线性地址或者物理地址有关的字段，发现只有用红色字体标明的那部分可能和线性地址有关。

struct page {

/\* First double word block \*/

unsigned long flags; /\* Atomic flags, some possibly

\* updated asynchronously \*/

union {

struct address\_space \*mapping; /\* If low bit clear, points to

\* inode address\_space, or NULL.

\* If page mapped as anonymous

\* memory, low bit is set, and

\* it points to anon\_vma object:

\* see PAGE\_MAPPING\_ANON below.

\*/

void \*s\_mem; /\* slab first object \*/

};

/\* Second double word \*/

struct {

union {

pgoff\_t index; /\* Our offset within mapping. \*/

void \*freelist; /\* sl[aou]b first free object \*/

};

union {

#if defined(CONFIG\_HAVE\_CMPXCHG\_DOUBLE) && \

defined(CONFIG\_HAVE\_ALIGNED\_STRUCT\_PAGE)

/\* Used for cmpxchg\_double in slub \*/

unsigned long counters;

#else

/\*

\* Keep \_count separate from slub cmpxchg\_double data.

\* As the rest of the double word is protected by

\* slab\_lock but \_count is not.

\*/

unsigned counters;

#endif

struct {

union {

/\*

\* Count of ptes mapped in

\* mms, to show when page is

\* mapped & limit reverse map

\* searches.

\*

\* Used also for tail pages

\* refcounting instead of

\* \_count. Tail pages cannot

\* be mapped and keeping the

\* tail page \_count zero at

\* all times guarantees

\* get\_page\_unless\_zero() will

\* never succeed on tail

\* pages.

\*/

atomic\_t \_mapcount;

struct { /\* SLUB \*/

unsigned inuse:16;

unsigned objects:15;

unsigned frozen:1;

};

int units; /\* SLOB \*/

};

atomic\_t \_count; /\* Usage count, see below. \*/

};

unsigned int active; /\* SLAB \*/

};

};

/\*

\* Third double word block

\*

\* WARNING: bit 0 of the first word encode PageTail(). That means

\* the rest users of the storage space MUST NOT use the bit to

\* avoid collision and false-positive PageTail().

\*/

union {

struct list\_head lru; /\* Pageout list, eg. active\_list

\* protected by zone->lru\_lock !

\* Can be used as a generic list

\* by the page owner.

\*/

struct { /\* slub per cpu partial pages \*/

struct page \*next; /\* Next partial slab \*/

#ifdef CONFIG\_64BIT

int pages; /\* Nr of partial slabs left \*/

int pobjects; /\* Approximate # of objects \*/

#else

short int pages;

short int pobjects;

#endif

};

struct rcu\_head rcu\_head; /\* Used by SLAB

\* when destroying via RCU

\*/

/\* Tail pages of compound page \*/

struct {

unsigned long compound\_head; /\* If bit zero is set \*/

/\* First tail page only \*/

#ifdef CONFIG\_64BIT

/\*

\* On 64 bit system we have enough space in struct page

\* to encode compound\_dtor and compound\_order with

\* unsigned int. It can help compiler generate better or

\* smaller code on some archtectures.

\*/

unsigned int compound\_dtor;

unsigned int compound\_order;

#else

unsigned short int compound\_dtor;

unsigned short int compound\_order;

#endif

};

#if defined(CONFIG\_TRANSPARENT\_HUGEPAGE) && USE\_SPLIT\_PMD\_PTLOCKS

struct {

unsigned long \_\_pad; /\* do not overlay pmd\_huge\_pte

\* with compound\_head to avoid

\* possible bit 0 collision.

\*/

pgtable\_t pmd\_huge\_pte; /\* protected by page->ptl \*/

内核数据结构page

};

#endif

};

/\* Remainder is not double word aligned \*/

union {

unsigned long private; /\* Mapping-private opaque data:

\* usually used for buffer\_heads

\* if PagePrivate set; used for

\* swp\_entry\_t if PageSwapCache;

\* indicates order in the buddy

\* system if PG\_buddy is set.

\*/

#if USE\_SPLIT\_PTE\_PTLOCKS

#if ALLOC\_SPLIT\_PTLOCKS

spinlock\_t \*ptl;

#else

spinlock\_t ptl;

#endif

#endif

struct kmem\_cache \*slab\_cache; /\* SL[AU]B: Pointer to slab \*/

};

#ifdef CONFIG\_MEMCG

struct mem\_cgroup \*mem\_cgroup;

#endif

/\*

\* On machines where all RAM is mapped into kernel address space,

\* we can simply calculate the virtual address. On machines with

\* highmem some memory is mapped into kernel virtual memory

\* dynamically, so we need a place to store that address.

\* Note that this field could be 16 bits on x86 ... ;)

\*

\* Architectures with slow multiplication can define

\* WANT\_PAGE\_VIRTUAL in asm/page.h

\*/

#if defined(WANT\_PAGE\_VIRTUAL)

void \*virtual; /\* Kernel virtual address (NULL if

not kmapped, ie. highmem) \*/

#endif /\* WANT\_PAGE\_VIRTUAL \*/

#ifdef CONFIG\_KMEMCHECK

/\*

\* kmemcheck wants to track the status of each byte in a page; this

\* is a pointer to such a status block. NULL if not tracked.

\*/

void \*shadow;

#endif

#ifdef LAST\_CPUPID\_NOT\_IN\_PAGE\_FLAGS

int \_last\_cpupid;

#endif

}

根据红色部分的注释我们了解到virtual字段的定义和高端内存有关，当不存在高端内存时WANT\_PAGE\_VIRTUAL没有定义，同时该字段页不会被定义。至于什么是高端内存，主要和CPU架构和物理内存有关，在80x86中线性地址空间是4G，但是安装的实际物理内存可以超过4G，所以物理内存不能完全映射到线性地址空间，超过4G的这部分物理内存不能被直接访问，因此内核定义了高端内存，将超过4G的物理内存通过内核映射的方式，映射到线性地址空间的第四个GB的最后128MB内存中。需要说明的是在Linux2.6.18版的内核中对i386架构定义了WANT\_PAGE\_VIRTUAL宏，但在Linux4.4版的内核中i368和x86\_64的代码进行了合并，对i386已经没有了WANT\_PAGE\_VIRTUAL的定义，所以推测这种内核映射是通过其它方式实现的（应该和Kconfig中的CONFING\_HIGHMEM有关）。

再来回到我们之前的问题，内核是如何将文件指针转换位线性地址空间中的地址的？我们发现上面virtual字段虽然和线性地址有关，但仍然不能解决我们的问题，所以我们查看具体的拷贝函数copy\_page\_to\_iter()。

1. **copy\_page\_to\_iter()函数的处理流程**

copy\_page\_to\_iter()函数负责将数据从页高速缓存读取到用户态缓存区。该函数接收四个参数，页描述符page的指针，该页中的偏移两offset，读取的数据的字节数bytes，以及iov\_iter描述符的指针，该结构中保存有用户态缓冲区的指针。

size\_t copy\_page\_to\_iter(struct page \*page, size\_t offset, size\_t bytes,

struct iov\_iter \*i)

{

if (i->type & (ITER\_BVEC|ITER\_KVEC)) {

void \*kaddr = kmap\_atomic(page);

size\_t wanted = copy\_to\_iter(kaddr + offset, bytes, i);

kunmap\_atomic(kaddr);

return wanted;

} else

return copy\_page\_to\_iter\_iovec(page, offset, bytes, i);

}

看到void \*kaddr = kmap\_atomic(page);这句时我们前面的问题似乎有了答案，内核应该是通过该函数实现由page到线性地址的转换。我们首先看看if语句的条件，在向后回退找到i->type被设置的地方，发现其在new\_sync\_read()函数中通过iov\_iter\_init()函数设置，通过下面的几段代码不难推测enum中的值指出结构体iov\_iter中的union字段的具体值是哪种类型，而且通过函数的调用参数和函数的定义，可以发现在sys\_read中iov\_iter是iovec类型的，所以此处条件不满足，执行else部分。

iov\_iter\_init(&iter, READ, &iov, 1, len);

enum {

ITER\_IOVEC = 0,

ITER\_KVEC = 2,

ITER\_BVEC = 4,

};

struct iov\_iter {

int type;

size\_t iov\_offset;

size\_t count;

union {

const struct iovec \*iov;

const struct kvec \*kvec;

const struct bio\_vec \*bvec;

};

unsigned long nr\_segs;

};

void iov\_iter\_init(struct iov\_iter \*i, int direction,

const struct iovec \*iov, unsigned long nr\_segs,

size\_t count)

{

/\* It will get better. Eventually... \*/

if (segment\_eq(get\_fs(), KERNEL\_DS)) {

direction |= ITER\_KVEC;

i->type = direction;

i->kvec = (struct kvec \*)iov;

} else {

i->type = direction;

i->iov = iov;

}

i->nr\_segs = nr\_segs;

i->iov\_offset = 0;

i->count = count;

}

If语句的else部分是一个函数调用，该函数接收的参数和调用它的copy\_page\_to\_iter函数的参数一样。

static size\_t copy\_page\_to\_iter\_iovec(struct page \*page, size\_t offset, size\_t bytes,

struct iov\_iter \*i)

{

size\_t skip, copy, left, wanted;

const struct iovec \*iov;

char \_\_user \*buf;

void \*kaddr, \*from;

if (unlikely(bytes > i->count))

bytes = i->count;

if (unlikely(!bytes))

return 0;

wanted = bytes;

iov = i->iov;

skip = i->iov\_offset;

buf = iov->iov\_base + skip;

copy = min(bytes, iov->iov\_len - skip);

if (!fault\_in\_pages\_writeable(buf, copy)) {

kaddr = kmap\_atomic(page);

from = kaddr + offset;

/\* first chunk, usually the only one \*/

left = \_\_copy\_to\_user\_inatomic(buf, from, copy);

copy -= left;

skip += copy;

from += copy;

bytes -= copy;

while (unlikely(!left && bytes)) {

iov++;

buf = iov->iov\_base;

copy = min(bytes, iov->iov\_len);

left = \_\_copy\_to\_user\_inatomic(buf, from, copy);

copy -= left;

skip = copy;

from += copy;

bytes -= copy;

}

if (likely(!bytes)) {

kunmap\_atomic(kaddr);

goto done;

}

offset = from - kaddr;

buf += copy;

kunmap\_atomic(kaddr);

copy = min(bytes, iov->iov\_len - skip);

}

/\* Too bad - revert to non-atomic kmap \*/

kaddr = kmap(page);

from = kaddr + offset;

left = \_\_copy\_to\_user(buf, from, copy);

copy -= left;

skip += copy;

from += copy;

bytes -= copy;

while (unlikely(!left && bytes)) {

iov++;

buf = iov->iov\_base;

copy = min(bytes, iov->iov\_len);

left = \_\_copy\_to\_user(buf, from, copy);

copy -= left;

skip = copy;

from += copy;

bytes -= copy;

}

kunmap(page);

done:

if (skip == iov->iov\_len) {

iov++;

skip = 0;

}

i->count -= wanted - bytes;

i->nr\_segs -= iov - i->iov;

i->iov = iov;

i->iov\_offset = skip;

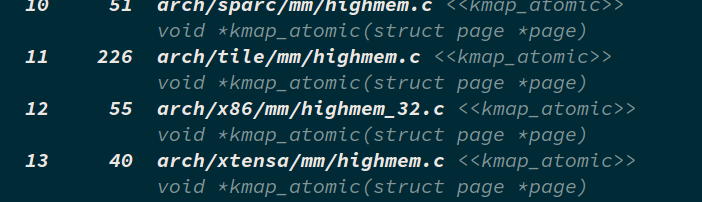
return wanted - bytes;

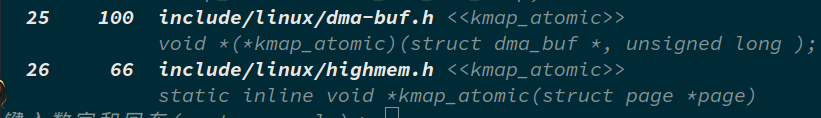
}

大概浏览一下代码，又发现了前面提到了kmap\_atomic()函数，看来是时候分析这个函数的具体工作了，从该函数的调用语句kaddr = kmap\_atomic(page)看函数的作用就是得到page描述的物理页框对应在线性地址空间中的线性地址。

1. **kmap\_atomic()函数的定义跟踪**

在此开始前我们首先要找到x86\_64架构中kmap\_atomic()函数的定义，该函数的定义是依赖于架构的。为了说明如何阅读内核代码，在此我们除了找到x86\_64架构的kmap\_atomic()函数外，还会查找i386架构中该函数的定义。首先在内核代码中查找该函数的定义，可以看到有很多，但根据名称和所在目录我们找到和上述架构有关的两个函数定义分别是下图中的12条和26条，从名字上感觉12条是i386架构的定义，26条自然是x86\_64所采用的函数定义。





不过，上述结论都是推断，我们还需进一步验证。进入上述两个定义处查找线索。12条中的定义如下，在此之前没有宏定义控制是否使用这个函数。

/\*highmem\_32.c\*/

//第12条的定义

#include <linux/highmem.h>

......

void \*kmap\_atomic(struct page \*page)

{

return kmap\_atomic\_prot(page, kmap\_prot);

}

EXPORT\_SYMBOL(kmap\_atomic);

第26条的函数定义如下，该处的定义通过两个宏来控制，一个是CONFIG\_HIGHMEM，另一个是ARCH\_HAS\_KMAP，只有当同时没有定义这两个宏时才会使用该处的函数定义。CONFIG\_HIGHMEM之前提到过和高端内存有关，在32位的系统上会被定义，ARCH\_HAS\_KMAP显然是和架构有关的，在内核代码中查找该宏的定义，我们发现只有一个地方定义了该宏，在arch/parisc/文件夹下面，显然只有parisc架构才会定义这个宏，而其他架构则不会定义，这样i386架构定义了CONFIG\_HIGHMEM所以不会使用该处的kmap\_atomic定义，使用第12条的定义。X86\_64架构这两个宏都没有定义因而使用的是26条的定义。

/\*highmem.h\*/

//第26条的定义

#ifdef CONFIG\_HIGHMEM

......

#else /\* CONFIG\_HIGHMEM \*/

......

#ifndef ARCH\_HAS\_KMAP

......

static inline void \*kmap\_atomic(struct page \*page)

{

preempt\_disable();

pagefault\_disable();

return page\_address(page);

}

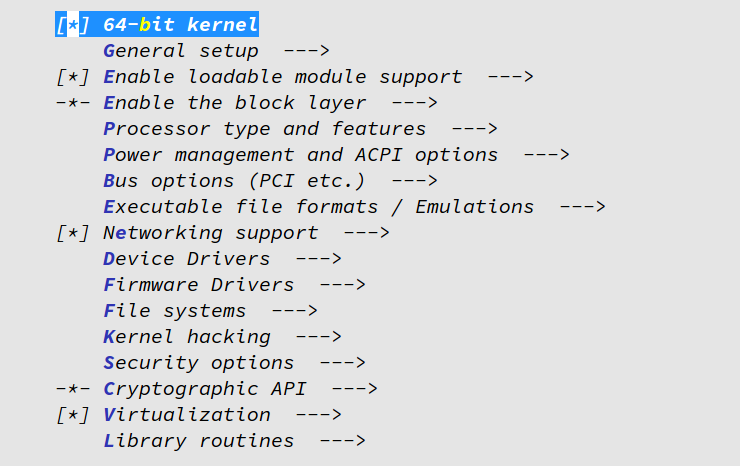
......

#endif

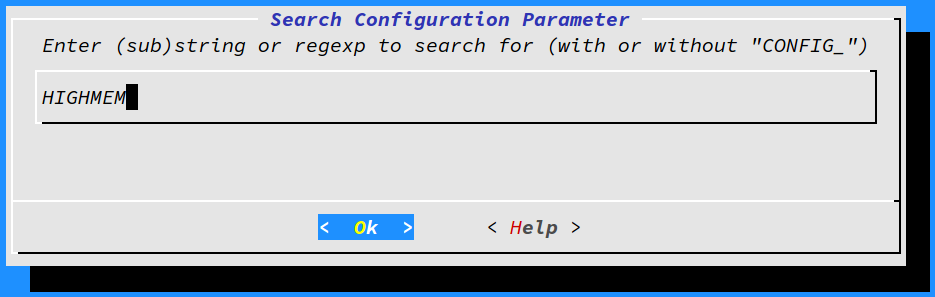
#endif /\* CONFIG\_HIGHMEM \*/

接下来我们验证CONFIG\_HIGHMEM只会在在32位条件下被定义：

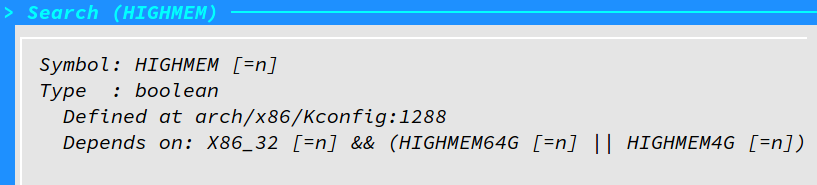
①、首先在内核代码根目录执行make menuconfig，出现下面的内核配置界面，我们可以看到当前设置为64位内核。



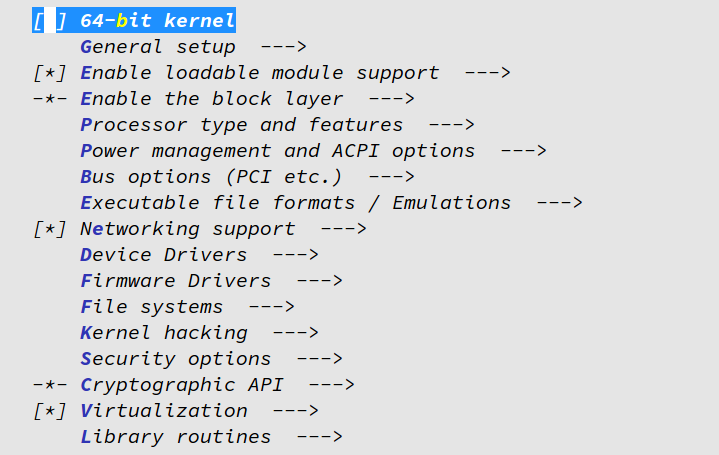
②、根据提示按下/键可以进行查找，进入如下界面



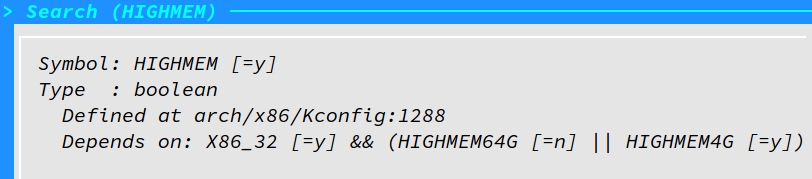
③、我们要查找CONFIG\_HIGHMEM，根据提示不带前缀CONFIG，所以输入HIGHMEM查找，结果如下，第一条就是我们要的查找结果，可以看到在64位内核中，CONFIG\_HIGHMEM没有设置。



④、设置内核版本为32位



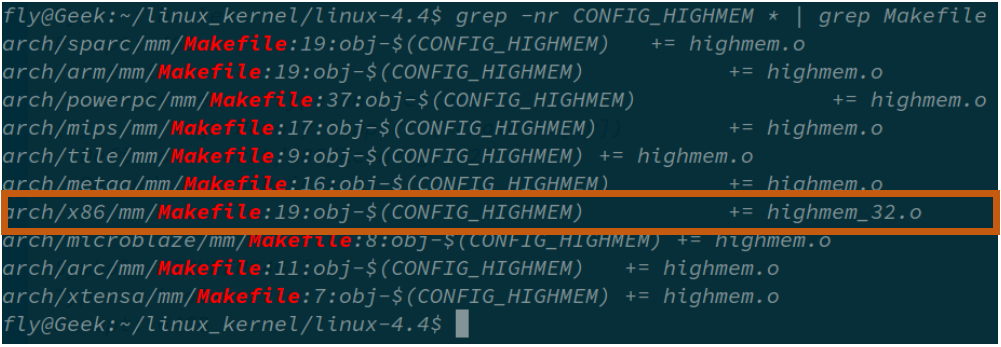
⑤、进行同样的查找，结果如下，可以看到32位内核中CONFIG\_HIGHMEM被设置。



⑥、现在我们根据上面两条定义所在的文件关系：highmem\_32.c包含highmem.h文件，推测在某个Makefile文件中会根据CONFIG\_HIGHMEM将highmem\_32.c文件包含到要编译的源文件中。我们在源代码目录下执行：

grep -r CONFIG\_HIGHMEM \* | grep Makefile

得到的结果如下，图中用橘黄色标识的那行证实了我们的推测，在arch/x86/mm/Makefile文件的第19行将highmem\_32.c文件包含到编译文件中，生成highmem\_32.o目标。



1. **page到线性地址的转换**

通过上面的分析我们找到了x86\_64的kmap\_automic()函数的定义。该函数中只有三个函数调用，根据函数名称推测prermpt\_disable()函数调用是禁止优先权抢占，pagefault\_disable()是禁止缺页中断，这里只是推测至于是不是？为什么这么做？我们不对此做分析。我们着重看page\_address()函数调用。

static inline void \*kmap\_atomic(struct page \*page)

{

preempt\_disable();

pagefault\_disable();

return page\_address(page);

}

在内核代码中查找page\_address()函数，在同一个文件中存在三个不同的定义，我们来分析x86\_64架构具体使用哪个函数定义。X86\_64没有定义CONFIG\_HIGHMEM和WANT\_PAGE\_VIRTUAL，所以不会定义HASHED\_PAGE\_VIRTUAL宏，从而不会使用第一个（绿色）定义，也不会使用第二个（蓝色）定义，使用的是第三个（紫色）的定义。

#if defined(CONFIG\_HIGHMEM) && !defined(WANT\_PAGE\_VIRTUAL)

#define HASHED\_PAGE\_VIRTUAL

#endif

#if defined(WANT\_PAGE\_VIRTUAL)

static inline void \*page\_address(const struct page \*page)

{

return page->virtual;

}

static inline void set\_page\_address(struct page \*page, void \*address)

{

page->virtual = address;

}

#define page\_address\_init() do { } while(0)

#endif

#if defined(HASHED\_PAGE\_VIRTUAL)

void \*page\_address(const struct page \*page);

void set\_page\_address(struct page \*page, void \*virtual);

void page\_address\_init(void);

#endif

#if !defined(HASHED\_PAGE\_VIRTUAL) && !defined(WANT\_PAGE\_VIRTUAL)

#define page\_address(page) lowmem\_page\_address(page)

#define set\_page\_address(page, address) do { } while(0)

#define page\_address\_init() do { } while(0)

#endif

查看lowmem\_page\_address()的定义，果然和之前page结构体中出现的注释解释的一样，可以通过简单的计算得到线性地址（原文：we can simply calculate the virtual address），函数的返回语句中出现了三个函数或者宏定义，\_\_va()，PFN\_PHYS()，page\_to\_pfn()。根据名字和参数大概推测函数的功能：page\_to\_pfn()根据page描述符得到页框号（pfn:page frame number），PFN\_PHYS()对页框号做处理至于PHYS是什么不太清楚，\_\_va()将得到的结果转变位线性地址（virtual address）。

static \_\_always\_inline void \*lowmem\_page\_address(const struct page \*page)

{

return \_\_va(PFN\_PHYS(page\_to\_pfn(page)));

}

接下来我们首先看看PFN\_PHYS()的实现，看看它对页框号具体做了什么。它的定义如下，该宏定义做了两件事情，一个是移位操作，一个是类型转换。根据phys\_addr\_t我们知道PHYS的意思是：物理地址，这样该宏定义的作用就很明显了，将一个物理页框号转变成一个物理地址。至于为什么通过向左移位PAGE\_SHIFT位就能得到物理地址，我们可以猜测，页框大小是size=2^PAGE\_SHIFT字节，即每size字节可以看成一组，这样每组的第一个字节的位置就可以将页框号左移PAGE\_SHIFT位得到。

#define PFN\_PHYS(x) ((phys\_addr\_t)(x) << PAGE\_SHIFT)

接下来再来看\_\_va()的实现，通过前面的推断\_\_va()实现物理地址到线性地址的转换，同样推测它会怎么转换，我们知道内核所在的物理地址是低端物理内存，而降这些物理地址映射到线性地址空间时，对应的是线性地址空间的高端地址，所以我们推测\_\_va()宏就是将物理地址移位或者加上一个数实现的，这个数可能是3G(十六进制的0xC0000000)，因为内核物理地址被映射到线性地址空间的第四个GB。查看\_\_va()的定义如下：

#define \_\_va(x) ((void \*)((unsigned long)(x)+PAGE\_OFFSET))

果然是加上一个数，看来第一个推测被证实了，再看看这个数具体是不是0xC0000000，找到PAGE\_OFFSET的定义如下：

#define PAGE\_OFFSET ((unsigned long)\_\_PAGE\_OFFSET)

再转到\_\_PAGE\_OFFSET的定义如下：

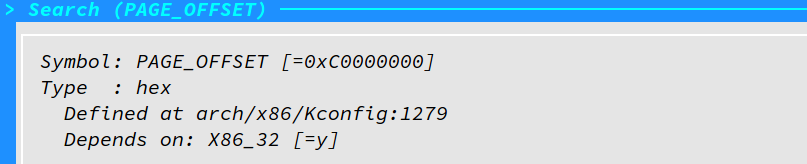
#define \_\_PAGE\_OFFSET \_AC(0xffff880000000000, UL)

看来第二个推测不正确，x86\_64中应该是0xffff880000000000，\_AC()只是将参数连接起来，UL指示这个数是无符号长整型。不过我们来看看我们错在哪里了，在查找\_\_PAGE\_OFFSET的定义时，我们选择的结果是最后一条，查找结果如下，结果中倒数第二个看来和32位系统有关。



#define \_\_PAGE\_OFFSET \_AC(CONFIG\_PAGE\_OFFSET, UL)

传递给\_AC()宏函数的参数是CONFIG\_PAGE\_OFFSET，我们到Kconfig中去查找它的定义，结果显示在32位系统中它的值是0xC0000000，所以之前我们的推测是32位内核的结果。

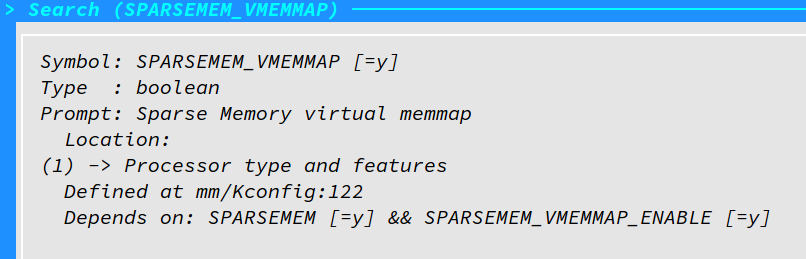


最后再来查看page\_to\_pfn的定义，它指向\_\_page\_to\_pfn函数。

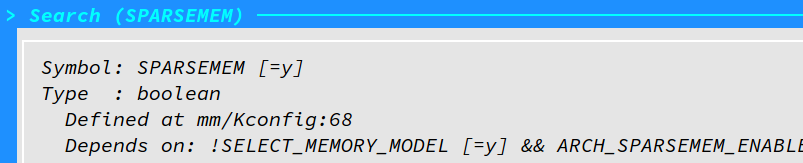
#define page\_to\_pfn \_\_page\_to\_pfn

\_\_page\_to\_pfn函数存在多个定义，也是通过宏定义来控制内核使用哪个函数。在make menuconfig配置界面中分别查找这些控制宏，32位和64位的查找结果如下：

64位



32位



存在四个\_page\_to\_pfn函数，如下根据前面的查找结果x86\_64架构使用绿色的定义，i386则采用蓝色的定义。我们分析x86\_64架构的定义，发现它是通过将page的地址减去vmemmap的地址实现将page转换为页框号的。

/\*

\* supports 3 memory models.

\*/

#if defined(CONFIG\_FLATMEM)

#define \_\_pfn\_to\_page(pfn) (mem\_map + ((pfn) - ARCH\_PFN\_OFFSET))

#define \_\_page\_to\_pfn(page) ((unsigned long)((page) - mem\_map) + \

ARCH\_PFN\_OFFSET)

#elif defined(CONFIG\_DISCONTIGMEM)

#define \_\_pfn\_to\_page(pfn) \

({ unsigned long \_\_pfn = (pfn); \

unsigned long \_\_nid = arch\_pfn\_to\_nid(\_\_pfn); \

NODE\_DATA(\_\_nid)->node\_mem\_map + arch\_local\_page\_offset(\_\_pfn, \_\_nid);\

})

#define \_\_page\_to\_pfn(pg) \

({ const struct page \*\_\_pg = (pg); \

struct pglist\_data \*\_\_pgdat = NODE\_DATA(page\_to\_nid(\_\_pg)); \

(unsigned long)(\_\_pg - \_\_pgdat->node\_mem\_map) + \

\_\_pgdat->node\_start\_pfn; \

})

#elif defined(CONFIG\_SPARSEMEM\_VMEMMAP)

/\* memmap is virtually contiguous. \*/

#define \_\_pfn\_to\_page(pfn) (vmemmap + (pfn))

#define \_\_page\_to\_pfn(page) (unsigned long)((page) - vmemmap)

#elif defined(CONFIG\_SPARSEMEM)

/\*

\* Note: section's mem\_map is encoded to reflect its start\_pfn.

\* section[i].section\_mem\_map == mem\_map's address - start\_pfn;

\*/

#define \_\_page\_to\_pfn(pg) \

({ const struct page \*\_\_pg = (pg); \

int \_\_sec = page\_to\_section(\_\_pg); \

(unsigned long)(\_\_pg - \_\_section\_mem\_map\_addr(\_\_nr\_to\_section(\_\_sec))); \

})

#define \_\_pfn\_to\_page(pfn) \

({ unsigned long \_\_pfn = (pfn); \

struct mem\_section \*\_\_sec = \_\_pfn\_to\_section(\_\_pfn); \

\_\_section\_mem\_map\_addr(\_\_sec) + \_\_pfn; \

})

#endif /\* CONFIG\_FLATMEM/DISCONTIGMEM/SPARSEMEM \*/

找到vmemmap的定义是一个强制类型转换，查找发现VMEMMAP\_START是一个常数。

#define vmemmap ((struct page \*)VMEMMAP\_START)

根据提示Documentation/x86/x86\_64/mm.txt文件是对内存映射的描述。

/\* See Documentation/x86/x86\_64/mm.txt for a description of the memory map. \*/

#define VMEMMAP\_START \_AC(0xffffea0000000000, UL)

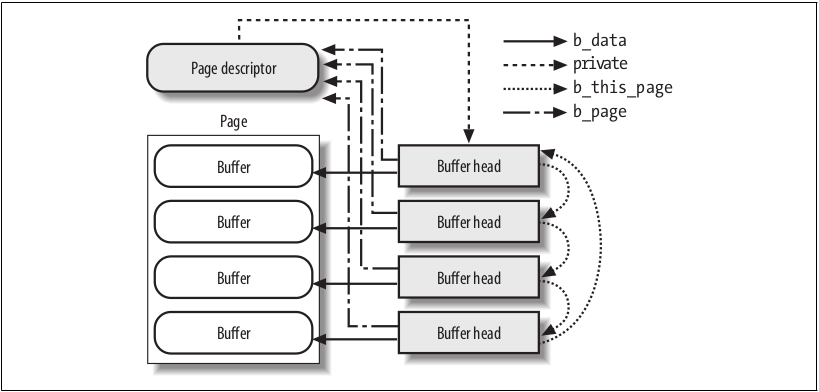
Documentation/x86/x86\_64/mm.txt中对上述常量的描述如下：

ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)

所以所有的page结构都存储在上述范围的虚拟地址空间中，组成一个数组，VMEMMAP\_START存放数组的起始地址。

至此，我们对文件读操作的流程有了一个清晰的认识。

1. **页高速缓存和块缓冲区**

在上一章节我们介绍了当读文件是文件数据已经在页高速缓存中的情况，当页不在高速缓存中时可以同过readpage函数读激活IO从磁盘将数据读入到页高速缓存中。对普通的文件数据和快设备文件这种操作有所不同。普通文件数据的读取是系统调用sys\_read()函数实现的，具体到函数调用中，就是readpage()函数。在进行挂载操作时要读取磁盘超级块、组描述符表、等磁盘控制信息，在路径查找时要读取inode节点表，在创建文件和分配文件数据块时要读取inode位图、和数据块位图，或者打开一个块设备文件，这些要读取块设备文件。在《Ext2文件系统分析》那份总结报告中我们已经介绍了内核vfs的数据结构和磁盘文件系统数据结构之间的关系，并提到块缓冲区，但我们并没有详细介绍块缓冲区的管理。本章主要介绍块缓冲区的管理。

页高速缓存中存放的数据是按块组织的，即一个页中往往存放多个块。这些块通过buffer\_head缓冲区首部组成一个链表，page结构的private字段指向链表中的第一个buffer\_head，buffer\_head的b\_this\_page指向下一个元素，b\_page指向拥有该缓冲区首部的page，b\_data指向页中的块缓冲区。下图引用自《深入理解Linux内核》。

1. **readpage方法**

address\_space对象的a\_ops字段该存放操作函数列表，其中的readpage()函数负责将磁盘中的数据读取到页高速缓存中，对普通文件和块设备文件有不同的操作。

1. **普通文件的readpage方法**

对ext2文件系统中的文件来说，其readpage指向ext2\_readpage()函数。

static int ext2\_readpage(struct file \*file, struct page \*page)

{

return mpage\_readpage(page, ext2\_get\_block);

}

ext2\_readpage()是对mpge\_readpage()函数的封装，只是在不同的文件系统中传入了不同的私有函数，对ext2文件系统第二个参数是ext2\_get\_block函数的地址。

/\*

\* This isn't called much at all

\*/

int mpage\_readpage(struct page \*page, get\_block\_t get\_block)

{

struct bio \*bio = NULL;

sector\_t last\_block\_in\_bio = 0;

struct buffer\_head map\_bh;

unsigned long first\_logical\_block = 0;

gfp\_t gfp = mapping\_gfp\_constraint(page->mapping, GFP\_KERNEL);

map\_bh.b\_state = 0;

map\_bh.b\_size = 0;

bio = do\_mpage\_readpage(bio, page, 1, &last\_block\_in\_bio,

&map\_bh, &first\_logical\_block, get\_block, gfp);

if (bio)

mpage\_bio\_submit(READ, bio);

return 0;

}

先来看函数的第二个参数的类型，发现它是一个函数类型，返回的是一个int类型，根据函数的名称get\_block我们推测这个函数是获取块号也可能会进一步获取数据块内容。

typedef int (get\_block\_t)(struct inode \*inode, sector\_t iblock,

struct buffer\_head \*bh\_result, int create);

根据函数的参数它传入了一个buffer\_head类型的指针，所以是后一种情形的可能性更大点。

int ext2\_get\_block(struct inode \*inode, sector\_t iblock, struct buffer\_head \*bh\_result, int create)

{

unsigned max\_blocks = bh\_result->b\_size >> inode->i\_blkbits;

int ret = ext2\_get\_blocks(inode, iblock, max\_blocks,

bh\_result, create);

if (ret > 0) {

bh\_result->b\_size = (ret << inode->i\_blkbits);

ret = 0;

}

return ret;

}

再来看看mpage\_readpage()函数的具体流程，起始处的几个定义看来会和确定数据所在的磁盘物理块位置有关系，根据mapping\_gfp\_constraint()函数的定义和内存映射有关。然后调用do\_mpage\_readpage()函数，看来该函数完成主要工作，最后调用mpage\_bio\_submit()函数，根据函数名称推测其主要作用就是项下层（通用块层提交bio），让下层负责具体的数据读取。

/\* Restricts the given gfp\_mask to what the mapping allows. \*/

static inline gfp\_t mapping\_gfp\_constraint(struct address\_space \*mapping,

gfp\_t gfp\_mask)

{

return mapping\_gfp\_mask(mapping) & gfp\_mask;

}

根据之前的推测和梳理，do\_mapge\_readpage()函数是我们分析的重点，它的功能明显是获取一个bio，bio结构中很可能能够确定所读取的数据在磁盘上的位置。首先来看函数功能注释，大概意思是：为磁盘块建立映射，并尽最大可能建立bio结构，如果存在不连续的磁盘物理块把它们提交到通用块层。

我们首先根据名称和初始值推测函数开始处定义的几个变量：

注：以下只是推测，所以含义可能有误，我们将在后文更正。

inode:明显是该页的所有者（索引节点指针）；

blkbits:块大小可以用多少位的二进制数表示；

blocks\_per\_page:每页多少块；

blocksize:块大小（以字节为单位）；

block\_in\_file:相对文件起始处该块所在的块号（即以块为单位的相对文件起始位置的编号）；

last\_block:最后一块；

last\_block\_in\_file：和block\_in\_file对应；

blocks[]:无法推测；

上述几个变量的最后四个我们只是根据名称来推测，下面再结合下面的代码来分析其含义：

**block\_in\_file** = (sector\_t)page->index << (PAGE\_CACHE\_SHIFT - blkbits);

page->index是以页为大小的文件偏移（相对文件开始位置），PAGE\_CACHE\_SHIFT是页大小位数，blkbits是块大小位数，所以左移(PAGE\_CACHE\_SHIFT - blkbits)就是算出该页的第一块相当于文件起始位置的块号。

**last\_block** = block\_in\_file + nr\_pages \* blocks\_per\_page;

显然是最后一个块的块号（相对文件起始位置），该块中可能没有文件数据。这里要注意，在这里调用do\_mpage\_readpage()函数时，nr\_pages的值为1。

**last\_block\_in\_file** = (i\_size\_read(inode) + blocksize - 1) >> blkbits;

因为last\_block表示的可能不是文件的最后一个数据块，所以用last\_block\_in\_file表示文件的最后一个数据块块号（相当文件起始位置）

/\*

\* This is the worker routine which does all the work of mapping the disk

\* blocks and constructs largest possible bios, submits them for IO if the

\* blocks are not contiguous on the disk.

\*

\* We pass a buffer\_head back and forth and use its buffer\_mapped() flag to

\* represent the validity of its disk mapping and to decide when to do the next

\* get\_block() call.

\*/

static struct bio \*

do\_mpage\_readpage(struct bio \*bio, struct page \*page, unsigned nr\_pages,

sector\_t \*last\_block\_in\_bio, struct buffer\_head \*map\_bh,

unsigned long \*first\_logical\_block, get\_block\_t get\_block,

gfp\_t gfp)

{

struct inode \*inode = page->mapping->host;

const unsigned blkbits = inode->i\_blkbits;

const unsigned blocks\_per\_page = PAGE\_CACHE\_SIZE >> blkbits;

const unsigned blocksize = 1 << blkbits;

sector\_t block\_in\_file;

sector\_t last\_block;

sector\_t last\_block\_in\_file;

sector\_t blocks[MAX\_BUF\_PER\_PAGE];

unsigned page\_block;

unsigned first\_hole = blocks\_per\_page;

struct block\_device \*bdev = NULL;

int length;

int fully\_mapped = 1;

unsigned nblocks;

unsigned relative\_block;

if (page\_has\_buffers(page))

goto confused;

block\_in\_file = (sector\_t)page->index << (PAGE\_CACHE\_SHIFT - blkbits);

last\_block = block\_in\_file + nr\_pages \* blocks\_per\_page;

last\_block\_in\_file = (i\_size\_read(inode) + blocksize - 1) >> blkbits;

if (last\_block > last\_block\_in\_file)

last\_block = last\_block\_in\_file;

page\_block = 0;

/\*

\* Map blocks using the result from the previous get\_blocks call first.

\*/

nblocks = map\_bh->b\_size >> blkbits;

if (buffer\_mapped(map\_bh) && block\_in\_file > \*first\_logical\_block &&

block\_in\_file < (\*first\_logical\_block + nblocks)) {

unsigned map\_offset = block\_in\_file - \*first\_logical\_block;

unsigned last = nblocks - map\_offset;

for (relative\_block = 0; ; relative\_block++) {

if (relative\_block == last) {

clear\_buffer\_mapped(map\_bh);

break;

}

if (page\_block == blocks\_per\_page)

break;

blocks[page\_block] = map\_bh->b\_blocknr + map\_offset +

relative\_block;

page\_block++;

block\_in\_file++;

}

bdev = map\_bh->b\_bdev;

}

/\*

\* Then do more get\_blocks calls until we are done with this page.

\*/

map\_bh->b\_page = page;

while (page\_block < blocks\_per\_page) {

map\_bh->b\_state = 0;

map\_bh->b\_size = 0;

if (block\_in\_file < last\_block) {

map\_bh->b\_size = (last\_block-block\_in\_file) << blkbits;

if (get\_block(inode, block\_in\_file, map\_bh, 0))

goto confused;

\*first\_logical\_block = block\_in\_file;

}

if (!buffer\_mapped(map\_bh)) {

fully\_mapped = 0;

if (first\_hole == blocks\_per\_page)

first\_hole = page\_block;

page\_block++;

block\_in\_file++;

continue;

}

/\* some filesystems will copy data into the page during

\* the get\_block call, in which case we don't want to

\* read it again. map\_buffer\_to\_page copies the data

\* we just collected from get\_block into the page's buffers

\* so readpage doesn't have to repeat the get\_block call

\*/

if (buffer\_uptodate(map\_bh)) {

map\_buffer\_to\_page(page, map\_bh, page\_block);

goto confused;

}

if (first\_hole != blocks\_per\_page)

goto confused; /\* hole -> non-hole \*/

/\* Contiguous blocks? \*/

if (page\_block && blocks[page\_block-1] != map\_bh->b\_blocknr-1)

goto confused;

nblocks = map\_bh->b\_size >> blkbits;

for (relative\_block = 0; ; relative\_block++) {

if (relative\_block == nblocks) {

clear\_buffer\_mapped(map\_bh);

break;

} else if (page\_block == blocks\_per\_page)

break;

blocks[page\_block] = map\_bh->b\_blocknr+relative\_block;

page\_block++;

block\_in\_file++;

}

bdev = map\_bh->b\_bdev;

}

if (first\_hole != blocks\_per\_page) {

zero\_user\_segment(page, first\_hole << blkbits, PAGE\_CACHE\_SIZE);

if (first\_hole == 0) {

SetPageUptodate(page);

unlock\_page(page);

goto out;

}

} else if (fully\_mapped) {

SetPageMappedToDisk(page);

}

if (fully\_mapped && blocks\_per\_page == 1 && !PageUptodate(page) &&

cleancache\_get\_page(page) == 0) {

SetPageUptodate(page);

goto confused;

}

/\*

\* This page will go to BIO. Do we need to send this BIO off first?

\*/

if (bio && (\*last\_block\_in\_bio != blocks[0] - 1))

bio = mpage\_bio\_submit(READ, bio);

alloc\_new:

if (bio == NULL) {

if (first\_hole == blocks\_per\_page) {

if (!bdev\_read\_page(bdev, blocks[0] << (blkbits - 9),

page))

goto out;

}

bio = mpage\_alloc(bdev, blocks[0] << (blkbits - 9),

min\_t(int, nr\_pages, BIO\_MAX\_PAGES), gfp);

if (bio == NULL)

goto confused;

}

length = first\_hole << blkbits;

if (bio\_add\_page(bio, page, length, 0) < length) {

bio = mpage\_bio\_submit(READ, bio);

goto alloc\_new;

}

relative\_block = block\_in\_file - \*first\_logical\_block;

nblocks = map\_bh->b\_size >> blkbits;

if ((buffer\_boundary(map\_bh) && relative\_block == nblocks) ||

(first\_hole != blocks\_per\_page))

bio = mpage\_bio\_submit(READ, bio);

else

\*last\_block\_in\_bio = blocks[blocks\_per\_page - 1];

out:

return bio;

confused:

if (bio)

bio = mpage\_bio\_submit(READ, bio);

if (!PageUptodate(page))

block\_read\_full\_page(page, get\_block);

else

unlock\_page(page);

goto out;

}

上述函数拥有几百行代码，但为了理解执行流程，还是需要仔细分析一下，在这里我就遇到了一个问题，buffer\_mapped函数的定义在哪里，看看下面的宏定义（buffer\_head.h文件中）。所以buffer\_mapped()函数就是为了测试BH\_Mapped位，根据注释该位判断这个块缓冲区是否存在磁盘映射。回退到前面发现buffer\_head结构是新创建的，其大部分字段都被赋值为0，所以if语句不会执行。

/\*

\* macro tricks to expand the set\_buffer\_foo(), clear\_buffer\_foo()

\* and buffer\_foo() functions.

\*/

#define BUFFER\_FNS(bit, name) \

static inline void set\_buffer\_##name(struct buffer\_head \*bh) \

{ \

set\_bit(BH\_##bit, &(bh)->b\_state); \

} \

static inline void clear\_buffer\_##name(struct buffer\_head \*bh) \

{ \

clear\_bit(BH\_##bit, &(bh)->b\_state); \

} \

static inline int buffer\_##name(const struct buffer\_head \*bh) \

{ \

return test\_bit(BH\_##bit, &(bh)->b\_state); \

}

BUFFER\_FNS(Mapped, mapped)

BH\_Mapped, /\* Has a disk mapping \*/

接下来我们分析while循环部分，根据注释它会多次调用get\_block函数，对页进行处理。

循环开始的条件语句中对buffer\_head的b\_size字段进行填充，然后调用传入的get\_block函数，看来给函数不出错会返回0，这里是前面提到的ext2\_get\_block()函数，为了便于阅读，我们在此再次贴上它的实现。可以看出该函数先计算出了块数，然后调用extw\_get\_blocks(),然后重新设置了b\_size字段，看来ext2\_get\_blocks()函数的返回值是实际读取的数据块个数。

int ext2\_get\_block(struct inode \*inode, sector\_t iblock, struct buffer\_head \*bh\_result, int create)

{

unsigned max\_blocks = bh\_result->b\_size >> inode->i\_blkbits;

int ret = ext2\_get\_blocks(inode, iblock, max\_blocks,

bh\_result, create);

if (ret > 0) {

bh\_result->b\_size = (ret << inode->i\_blkbits);

ret = 0;

}

return ret;

}

现在来分析ext2\_get\_block，看到这么长的定义是不是吓了一跳？没关系，我们来看看函数注释，再来决定要不要看函数的具体实现。注释中提到了分配，看来这个函数和块分配有关系，如果深入这里，那么这篇文章可能要写上百页了，所以就略过这个函数的分析。但由于我们分析的是文件读操作过程，所以这里并不会涉及磁盘数据块的分配，所以对读操作而言这个函数可能就是在内存中建立了磁盘数据块的映射。下面给出几篇网络文章详细介绍：

【参考】

1、<http://blog.chinaunix.net/uid-52662-id-2107876.html>

2、<http://blog.csdn.net/yunsongice/article/details/5822495>

3、<http://blog.csdn.net/onlyg/article/details/6835022>

/\*

\* Allocation strategy is simple: if we have to allocate something, we will

\* have to go the whole way to leaf. So let's do it before attaching anything

\* to tree, set linkage between the newborn blocks, write them if sync is

\* required, recheck the path, free and repeat if check fails, otherwise

\* set the last missing link (that will protect us from any truncate-generated

\* removals - all blocks on the path are immune now) and possibly force the

\* write on the parent block.

\* That has a nice additional property: no special recovery from the failed

\* allocations is needed - we simply release blocks and do not touch anything

\* reachable from inode.

\*

\* `handle' can be NULL if create == 0.

\*

\* return > 0, # of blocks mapped or allocated.

\* return = 0, if plain lookup failed.

\* return < 0, error case.

\*/

static int ext2\_get\_blocks(struct inode \*inode,

sector\_t iblock, unsigned long maxblocks,

struct buffer\_head \*bh\_result,

int create)

{

int err = -EIO;

int offsets[4];

Indirect chain[4];

Indirect \*partial;

ext2\_fsblk\_t goal;

int indirect\_blks;

int blocks\_to\_boundary = 0;

int depth;

struct ext2\_inode\_info \*ei = EXT2\_I(inode);

int count = 0;

ext2\_fsblk\_t first\_block = 0;

BUG\_ON(maxblocks == 0);

depth = ext2\_block\_to\_path(inode,iblock,offsets,&blocks\_to\_boundary);

if (depth == 0)

return (err);

partial = ext2\_get\_branch(inode, depth, offsets, chain, &err);

/\* Simplest case - block found, no allocation needed \*/

if (!partial) {

first\_block = le32\_to\_cpu(chain[depth - 1].key);

clear\_buffer\_new(bh\_result); /\* What's this do? \*/

count++;

/\*map more blocks\*/

while (count < maxblocks && count <= blocks\_to\_boundary) {

ext2\_fsblk\_t blk;

if (!verify\_chain(chain, chain + depth - 1)) {

/\*

\* Indirect block might be removed by

\* truncate while we were reading it.

\* Handling of that case: forget what we've

\* got now, go to reread.

\*/

err = -EAGAIN;

count = 0;

break;

}

blk = le32\_to\_cpu(\*(chain[depth-1].p + count));

if (blk == first\_block + count)

count++;

else

break;

}

if (err != -EAGAIN)

goto got\_it;

}

/\* Next simple case - plain lookup or failed read of indirect block \*/

if (!create || err == -EIO)

goto cleanup;

mutex\_lock(&ei->truncate\_mutex);

/\*

\* If the indirect block is missing while we are reading

\* the chain(ext2\_get\_branch() returns -EAGAIN err), or

\* if the chain has been changed after we grab the semaphore,

\* (either because another process truncated this branch, or

\* another get\_block allocated this branch) re-grab the chain to see if

\* the request block has been allocated or not.

\*

\* Since we already block the truncate/other get\_block

\* at this point, we will have the current copy of the chain when we

\* splice the branch into the tree.

\*/

if (err == -EAGAIN || !verify\_chain(chain, partial)) {

while (partial > chain) {

brelse(partial->bh);

partial--;

}

partial = ext2\_get\_branch(inode, depth, offsets, chain, &err);

if (!partial) {

count++;

mutex\_unlock(&ei->truncate\_mutex);

if (err)

goto cleanup;

clear\_buffer\_new(bh\_result);

goto got\_it;

}

}

/\*

\* Okay, we need to do block allocation. Lazily initialize the block

\* allocation info here if necessary

\*/

if (S\_ISREG(inode->i\_mode) && (!ei->i\_block\_alloc\_info))

ext2\_init\_block\_alloc\_info(inode);

goal = ext2\_find\_goal(inode, iblock, partial);

/\* the number of blocks need to allocate for [d,t]indirect blocks \*/

indirect\_blks = (chain + depth) - partial - 1;

/\*

\* Next look up the indirect map to count the totoal number of

\* direct blocks to allocate for this branch.

\*/

count = ext2\_blks\_to\_allocate(partial, indirect\_blks,

maxblocks, blocks\_to\_boundary);

/\*

\* XXX ???? Block out ext2\_truncate while we alter the tree

\*/

err = ext2\_alloc\_branch(inode, indirect\_blks, &count, goal,

offsets + (partial - chain), partial);

if (err) {

mutex\_unlock(&ei->truncate\_mutex);

goto cleanup;

}

if (IS\_DAX(inode)) {

/\*

\* block must be initialised before we put it in the tree

\* so that it's not found by another thread before it's

\* initialised

\*/

err = dax\_clear\_blocks(inode, le32\_to\_cpu(chain[depth-1].key),

1 << inode->i\_blkbits);

if (err) {

mutex\_unlock(&ei->truncate\_mutex);

goto cleanup;

}

}

ext2\_splice\_branch(inode, iblock, partial, indirect\_blks, count);

mutex\_unlock(&ei->truncate\_mutex);

set\_buffer\_new(bh\_result);

got\_it:

map\_bh(bh\_result, inode->i\_sb, le32\_to\_cpu(chain[depth-1].key));

if (count > blocks\_to\_boundary)

set\_buffer\_boundary(bh\_result);

err = count;

/\* Clean up and exit \*/

partial = chain + depth - 1; /\* the whole chain \*/

cleanup:

while (partial > chain) {

brelse(partial->bh);

partial--;

}

return err;

}

接下来回到主线，调用完get\_block函数后，设置了first\_logical\_block指向的值。接下来有时一个if语句，不过根据代码中的名字推测，这部分字段应该和文件洞的处理有关。所以if条件的意思是如果块没有被映射，那么这部分将被执行，本次循环结束处理下一个块。

if (block\_in\_file < last\_block) {

map\_bh->b\_size = (last\_block-block\_in\_file) << blkbits;

if (get\_block(inode, block\_in\_file, map\_bh, 0))

goto confused;

\*first\_logical\_block = block\_in\_file;

}

假设文件不存在洞，继续向下执行。接下来的if条件语句根据注释我们可以略过。接下来执行到下面这个if条件，如果之前文件没有hole出现，则first\_hole == blocks\_per\_page所以在我们的假定条件下（不存在hole）不会执行if中的代码。

if (first\_hole != blocks\_per\_page)

goto confused; /\* hole -> non-hole \*/

接下来的if语句看来和页中的块是否连续有关，不过具体为什么用这条语句判断连续还是不太明白，所以暂时跳过这里，继续向下分析。

/\* Contiguous blocks? \*/

if (page\_block && blocks[page\_block-1] != map\_bh->b\_blocknr-1)

goto confused;

接下来计算了这个块缓冲区所包含的磁盘数据块个数，然后是最后一个for循环，while循环终于要到尽头了，^\_^↑↑。for循环很简单就是将所有buffer\_head中的磁盘逻辑块号保存到blocks[]数组中。最后设置了bdev一个设备描述符。

nblocks = map\_bh->b\_size >> blkbits;

for (relative\_block = 0; ; relative\_block++) {

if (relative\_block == nblocks) {

clear\_buffer\_mapped(map\_bh);

break;

} else if (page\_block == blocks\_per\_page)

break;

blocks[page\_block] = map\_bh->b\_blocknr+relative\_block;

page\_block++;

block\_in\_file++;

}

bdev = map\_bh->b\_bdev;

}

回过头上面遗留的问题：那条if语句是怎么判断块号是否是连续的？

通过调用get\_block()函数，buffer\_head的b\_blocknr字段会保存若干个连续块中的第一个块的磁盘逻辑块号，借助其b\_size字段和块大小，可以算出这次映射了多少个块，然后将这些连续的块的块号通过上述for循环保存到blocks[]数组中，然后继续while循环，调用get\_block()又获得若干连续块，而buffer\_head的b\_blocknr字段保存起始的逻辑块号，所以b\_blocknr-1如果和上次获取的连续块的最后一块的逻辑块号相等，则两次获取的逻辑快是连续的。

while循环结束，接下来的if语句在假设条件（文件不存在洞）下不满足，且fully\_mapped==1，所以执行else if部分，调用SetPageMappedToDisk()函数，根据函数名称，其作用是将页做为整体映射。

if (first\_hole != blocks\_per\_page) {

zero\_user\_segment(page, first\_hole << blkbits, PAGE\_CACHE\_SIZE);

if (first\_hole == 0) {

SetPageUptodate(page);

unlock\_page(page);

goto out;

}

} else if (fully\_mapped) {

SetPageMappedToDisk(page);

}

接下来的if语句判断页是不是最新的，如果不是更新页，可能按块更新。

if (fully\_mapped && blocks\_per\_page == 1 && !PageUptodate(page) &&

cleancache\_get\_page(page) == 0) {

SetPageUptodate(page);

goto confused;

}

由于在调用这个函数时传入的bio指针为空，所以下面的if条件不满足。

/\*

\* This page will go to BIO. Do we need to send this BIO off first?

\*/

if (bio && (\*last\_block\_in\_bio != blocks[0] - 1))

bio = mpage\_bio\_submit(READ, bio);

alloc\_new标志的下面是分配一个bio描述符，然后调用bio\_add\_page()函数，将页加入到bio中，查看该函数发现，只要执行成功返回值必然等于length，所以不会执行if中的语句。

length = first\_hole << blkbits;

if (bio\_add\_page(bio, page, length, 0) < length) {

bio = mpage\_bio\_submit(READ, bio);

goto alloc\_new;

}

下面的一个if语句也不会被执行，所以执行else部分，然后返回。返回后在mpage\_readpage()函数中bio不为空，所以执行mpage\_bio\_submit()函数调用。这样bio描述符被提交到通用块层，进行数据传输。

IO数据传输完成后就会调用mpage\_end\_io()函数，设置页描述符的PG\_uptodate，唤醒阻塞进程，清除bio描述符。

1. **块设备文件的readpage方法**

普通文件系统的readpage方法依赖与具体的文件系统，根据之前对file，inode，address\_space等结构的分析，我们知道，inode是address\_space的所有者，所以在创建inode时就会创建相应的address\_space对象，并根据文件系统类型设置它的字段，其中包括a\_ops字段，readpage方法就是a\_ops字段指向的函数列表中的一个函数。之前介绍路径查找过程时讲到会调用父目录inode节点的lookup()操作函数，从磁盘查找一个文件的inode节点号，并在内存中建立该文件的inode索引节点和目录项dentry，所以我们从这里开始。普通文件和设备文件有不同的readpage方法，继而有不同的a\_ops，所以首先查看内核ext2文件系统中这种差别是在什么地方处理的（显然这种区分是对文件类型的区分，而文件类型最原始的存放地是ext2\_inode结构的i\_mode字段）。

首先是lookup()函数，在ext2文件系统中具体是ext2\_lookup()，看来调用ext2\_iget()建立索引节点和address\_space对象。

static struct dentry \*ext2\_lookup(struct inode \* dir, struct dentry \*dentry, unsigned int flags)

{

struct inode \* inode;

ino\_t ino;

if (dentry->d\_name.len > EXT2\_NAME\_LEN)

return ERR\_PTR(-ENAMETOOLONG);

ino = ext2\_inode\_by\_name(dir, &dentry->d\_name);

inode = NULL;

if (ino) {

inode = ext2\_iget(dir->i\_sb, ino);

if (inode == ERR\_PTR(-ESTALE)) {

ext2\_error(dir->i\_sb, \_\_func\_\_,

"deleted inode referenced: %lu",

(unsigned long) ino);

return ERR\_PTR(-EIO);

}

}

return d\_splice\_alias(inode, dentry);

}

跳转到ext2\_iget()函数，发现该段代码中正好有我们需要的inode->i\_mode字段，而且明显是根据不同的文件类型执行不同的操作，对普通文件aops指向ext2\_aops，对块设备文件，调用init\_special\_inode()函数，目前linux采用12位的主设备号和20位的此设备号（以前是8+8）。

struct inode \*ext2\_iget (struct super\_block \*sb, unsigned long ino)

{

struct ext2\_inode\_info \*ei;

struct buffer\_head \* bh;

struct ext2\_inode \*raw\_inode;

struct inode \*inode;

long ret = -EIO;

int n;

uid\_t i\_uid;

gid\_t i\_gid;

inode = iget\_locked(sb, ino);

if (!inode)

return ERR\_PTR(-ENOMEM);

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

return inode;

ei = EXT2\_I(inode);

ei->i\_block\_alloc\_info = NULL;

raw\_inode = ext2\_get\_inode(inode->i\_sb, ino, &bh);

if (IS\_ERR(raw\_inode)) {

ret = PTR\_ERR(raw\_inode);

goto bad\_inode;

}

inode->i\_mode = le16\_to\_cpu(raw\_inode->i\_mode);

i\_uid = (uid\_t)le16\_to\_cpu(raw\_inode->i\_uid\_low);

i\_gid = (gid\_t)le16\_to\_cpu(raw\_inode->i\_gid\_low);

if (!(test\_opt (inode->i\_sb, NO\_UID32))) {

i\_uid |= le16\_to\_cpu(raw\_inode->i\_uid\_high) << 16;

i\_gid |= le16\_to\_cpu(raw\_inode->i\_gid\_high) << 16;

}

i\_uid\_write(inode, i\_uid);

i\_gid\_write(inode, i\_gid);

set\_nlink(inode, le16\_to\_cpu(raw\_inode->i\_links\_count));

inode->i\_size = le32\_to\_cpu(raw\_inode->i\_size);

inode->i\_atime.tv\_sec = (signed)le32\_to\_cpu(raw\_inode->i\_atime);

inode->i\_ctime.tv\_sec = (signed)le32\_to\_cpu(raw\_inode->i\_ctime);

inode->i\_mtime.tv\_sec = (signed)le32\_to\_cpu(raw\_inode->i\_mtime);

inode->i\_atime.tv\_nsec = inode->i\_mtime.tv\_nsec = inode->i\_ctime.tv\_nsec = 0;

ei->i\_dtime = le32\_to\_cpu(raw\_inode->i\_dtime);

/\* We now have enough fields to check if the inode was active or not.

\* This is needed because nfsd might try to access dead inodes

\* the test is that same one that e2fsck uses

\* NeilBrown 1999oct15

\*/

if (inode->i\_nlink == 0 && (inode->i\_mode == 0 || ei->i\_dtime)) {

/\* this inode is deleted \*/

brelse (bh);

ret = -ESTALE;

goto bad\_inode;

}

inode->i\_blocks = le32\_to\_cpu(raw\_inode->i\_blocks);

ei->i\_flags = le32\_to\_cpu(raw\_inode->i\_flags);

ei->i\_faddr = le32\_to\_cpu(raw\_inode->i\_faddr);

ei->i\_frag\_no = raw\_inode->i\_frag;

ei->i\_frag\_size = raw\_inode->i\_fsize;

ei->i\_file\_acl = le32\_to\_cpu(raw\_inode->i\_file\_acl);

ei->i\_dir\_acl = 0;

if (S\_ISREG(inode->i\_mode))

inode->i\_size |= ((\_\_u64)le32\_to\_cpu(raw\_inode->i\_size\_high)) << 32;

else

ei->i\_dir\_acl = le32\_to\_cpu(raw\_inode->i\_dir\_acl);

ei->i\_dtime = 0;

inode->i\_generation = le32\_to\_cpu(raw\_inode->i\_generation);

ei->i\_state = 0;

ei->i\_block\_group = (ino - 1) / EXT2\_INODES\_PER\_GROUP(inode->i\_sb);

ei->i\_dir\_start\_lookup = 0;

/\*

\* NOTE! The in-memory inode i\_data array is in little-endian order

\* even on big-endian machines: we do NOT byteswap the block numbers!

\*/

for (n = 0; n < EXT2\_N\_BLOCKS; n++)

ei->i\_data[n] = raw\_inode->i\_block[n];

if (S\_ISREG(inode->i\_mode)) {

inode->i\_op = &ext2\_file\_inode\_operations;

if (test\_opt(inode->i\_sb, NOBH)) {

inode->i\_mapping->a\_ops = &ext2\_nobh\_aops;

inode->i\_fop = &ext2\_file\_operations;

} else {

inode->i\_mapping->a\_ops = &ext2\_aops;

inode->i\_fop = &ext2\_file\_operations;

}

} else if (S\_ISDIR(inode->i\_mode)) {

inode->i\_op = &ext2\_dir\_inode\_operations;

inode->i\_fop = &ext2\_dir\_operations;

if (test\_opt(inode->i\_sb, NOBH))

inode->i\_mapping->a\_ops = &ext2\_nobh\_aops;

else

inode->i\_mapping->a\_ops = &ext2\_aops;

} else if (S\_ISLNK(inode->i\_mode)) {

if (ext2\_inode\_is\_fast\_symlink(inode)) {

inode->i\_link = (char \*)ei->i\_data;

inode->i\_op = &ext2\_fast\_symlink\_inode\_operations;

nd\_terminate\_link(ei->i\_data, inode->i\_size,

sizeof(ei->i\_data) - 1);

} else {

inode->i\_op = &ext2\_symlink\_inode\_operations;

if (test\_opt(inode->i\_sb, NOBH))

inode->i\_mapping->a\_ops = &ext2\_nobh\_aops;

else

inode->i\_mapping->a\_ops = &ext2\_aops;

}

} else {

inode->i\_op = &ext2\_special\_inode\_operations;

if (raw\_inode->i\_block[0])

init\_special\_inode(inode, inode->i\_mode,

old\_decode\_dev(le32\_to\_cpu(raw\_inode->i\_block[0])));

else

init\_special\_inode(inode, inode->i\_mode,

new\_decode\_dev(le32\_to\_cpu(raw\_inode->i\_block[1])));

}

brelse (bh);

ext2\_set\_inode\_flags(inode);

unlock\_new\_inode(inode);

return inode;

bad\_inode:

iget\_failed(inode);

return ERR\_PTR(ret);

}

查看init\_special\_inode()函数的实现我们发现并没有对i\_mapping->aops字段赋值的操作，不过倒是设置了i\_fop和i\_rdev字段。那么还有什么地方会对aops字段进行设置，应该在open操作的时候也可以设置。

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 = &pipefifo\_fops;

else if (S\_ISSOCK(mode))

; /\* leave it no\_open\_fops \*/

else

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

" inode %s:%lu\n", mode, inode->i\_sb->s\_id,

inode->i\_ino);

}

为了减小篇幅直接从打开文件过程中调用的一个函数do\_dentry\_open开始，当然找到这个函数也是要花点精力，我们着重看红色字体表明的那部分。

static int do\_dentry\_open(struct file \*f,

struct inode \*inode,

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

const struct cred \*cred)

{

static const struct file\_operations empty\_fops = {};

int error;

f->f\_mode = OPEN\_FMODE(f->f\_flags) | FMODE\_LSEEK |

FMODE\_PREAD | FMODE\_PWRITE;

path\_get(&f->f\_path);

f->f\_inode = inode;

f->f\_mapping = inode->i\_mapping;

if (unlikely(f->f\_flags & O\_PATH)) {

f->f\_mode = FMODE\_PATH;

f->f\_op = &empty\_fops;

return 0;

}

if (f->f\_mode & FMODE\_WRITE && !special\_file(inode->i\_mode)) {

error = get\_write\_access(inode);

if (unlikely(error))

goto cleanup\_file;

error = \_\_mnt\_want\_write(f->f\_path.mnt);

if (unlikely(error)) {

put\_write\_access(inode);

goto cleanup\_file;

}

f->f\_mode |= FMODE\_WRITER;

}

/\* POSIX.1-2008/SUSv4 Section XSI 2.9.7 \*/

if (S\_ISREG(inode->i\_mode))

f->f\_mode |= FMODE\_ATOMIC\_POS;

f->f\_op = fops\_get(inode->i\_fop);

if (unlikely(WARN\_ON(!f->f\_op))) {

error = -ENODEV;

goto cleanup\_all;

}

error = security\_file\_open(f, cred);

if (error)

goto cleanup\_all;

error = break\_lease(inode, f->f\_flags);

if (error)

goto cleanup\_all;

if (!open)

open = f->f\_op->open;

if (open) {

error = open(inode, f);

if (error)

goto cleanup\_all;

}

if ((f->f\_mode & (FMODE\_READ | FMODE\_WRITE)) == FMODE\_READ)

i\_readcount\_inc(inode);

if ((f->f\_mode & FMODE\_READ) &&

likely(f->f\_op->read || f->f\_op->read\_iter))

f->f\_mode |= FMODE\_CAN\_READ;

if ((f->f\_mode & FMODE\_WRITE) &&

likely(f->f\_op->write || f->f\_op->write\_iter))

f->f\_mode |= FMODE\_CAN\_WRITE;

f->f\_flags &= ~(O\_CREAT | O\_EXCL | O\_NOCTTY | O\_TRUNC);

file\_ra\_state\_init(&f->f\_ra, f->f\_mapping->host->i\_mapping);

return 0;

cleanup\_all:

fops\_put(f->f\_op);

if (f->f\_mode & FMODE\_WRITER) {

put\_write\_access(inode);

\_\_mnt\_drop\_write(f->f\_path.mnt);

}

cleanup\_file:

path\_put(&f->f\_path);

f->f\_path.mnt = NULL;

f->f\_path.dentry = NULL;

f->f\_inode = NULL;

return error;

}

用红色字体标明的那些语句显示：f->f\_op和inode->i\_fop的值相同，之后调用了f\_op->open函数。之前我们介绍块设备文件的i\_fop指向def\_blk\_fops，所以open函数实际是blkdev\_open函数。普通文件则没有open函数。

块设备文件的f\_op字段：

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

.open = blkdev\_open,

.release = blkdev\_close,

.llseek = block\_llseek,

.read\_iter = blkdev\_read\_iter,

.write\_iter = blkdev\_write\_iter,

.mmap = generic\_file\_mmap,

.fsync = blkdev\_fsync,

.unlocked\_ioctl = block\_ioctl,

#ifdef CONFIG\_COMPAT

.compat\_ioctl = compat\_blkdev\_ioctl,

#endif

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

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

};

普通文件的f\_op字段：

const struct file\_operations ext2\_file\_operations = {

.llseek = generic\_file\_llseek,

.read\_iter = generic\_file\_read\_iter,

.write\_iter = generic\_file\_write\_iter,

.unlocked\_ioctl = ext2\_ioctl,

#ifdef CONFIG\_COMPAT

.compat\_ioctl = ext2\_compat\_ioctl,

#endif

.mmap = ext2\_file\_mmap,

.open = dquot\_file\_open,

.release = ext2\_release\_file,

.fsync = ext2\_fsync,

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

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

};

blkdev\_open()函数的定义如下，在该函数中把文件的f\_mapping字段设置为拥有该文件索引节点的块设备的块设备描述符的inode索引节点的i\_mapping字段（可能有点拗口，总之这个块设备文件的inode和块设备描述符的bd\_inode所指的索引节点不是同一个）。因此，同一磁盘的所有块设备文件具有相同的address\_space结构。

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

{

struct block\_device \*bdev;

/\*

\* 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;

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

filp->f\_mode |= FMODE\_NDELAY;

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

filp->f\_mode |= FMODE\_EXCL;

if ((filp->f\_flags & O\_ACCMODE) == 3)

filp->f\_mode |= FMODE\_WRITE\_IOCTL;

bdev = bd\_acquire(inode);

if (bdev == NULL)

return -ENOMEM;

filp->f\_mapping = bdev->bd\_inode->i\_mapping;

return blkdev\_get(bdev, filp->f\_mode, filp);

}