

文件IO——read

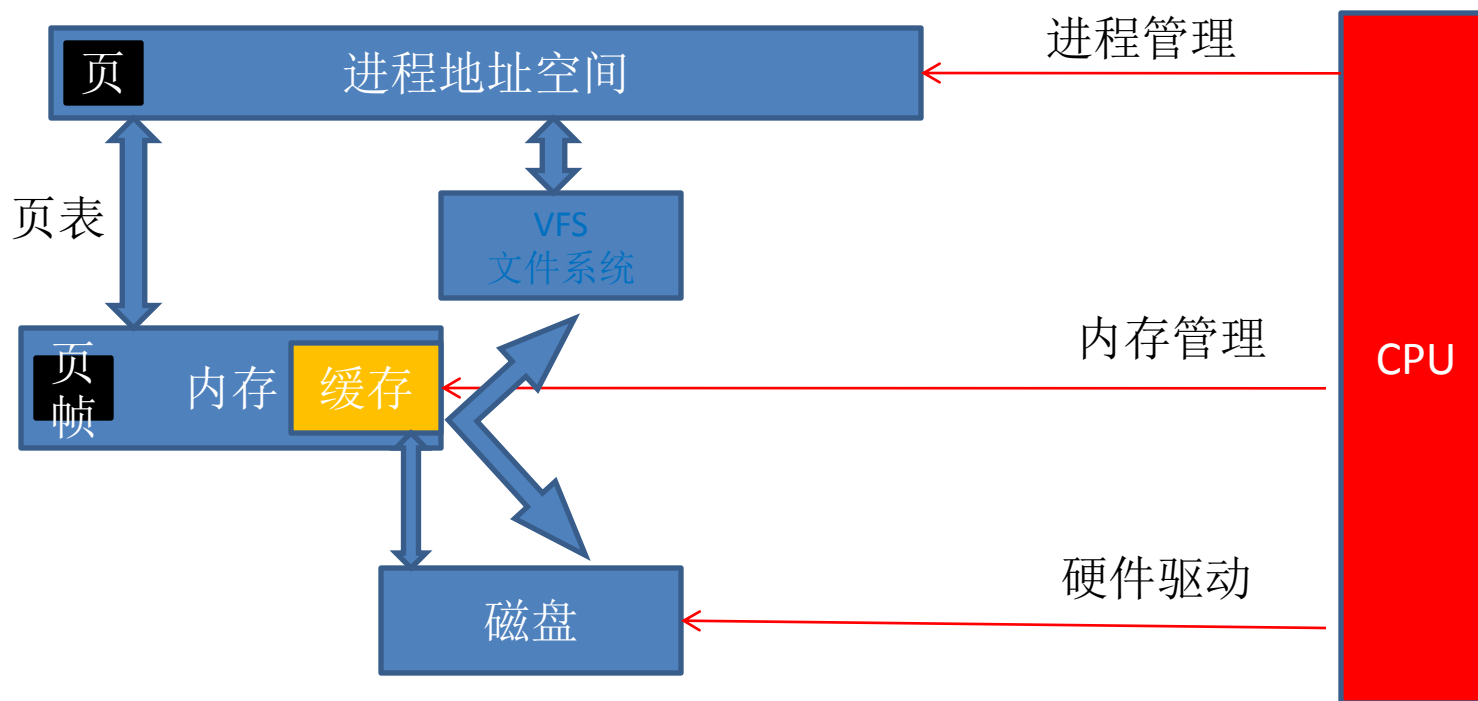
用户空间的文件read

- 不带缓冲的read操作([POSIX.1](#)): 利用open返回的文件描述符 [read\(STDIN, buf, BUFSIZE\)](#)
- 带缓冲的fread操作([ISO C](#)): 使用fopen返回的FILE对象指针fread(buf, 1, 10, fp)

问题提出

- 如何将用户空间读取文件操作抽象表示？
- 磁盘数据存储结构？
- 内核如何构建用于向磁盘驱动提交的读请求？
- 如何利用缓存提高系统读取文件的速度？

内核基本模块

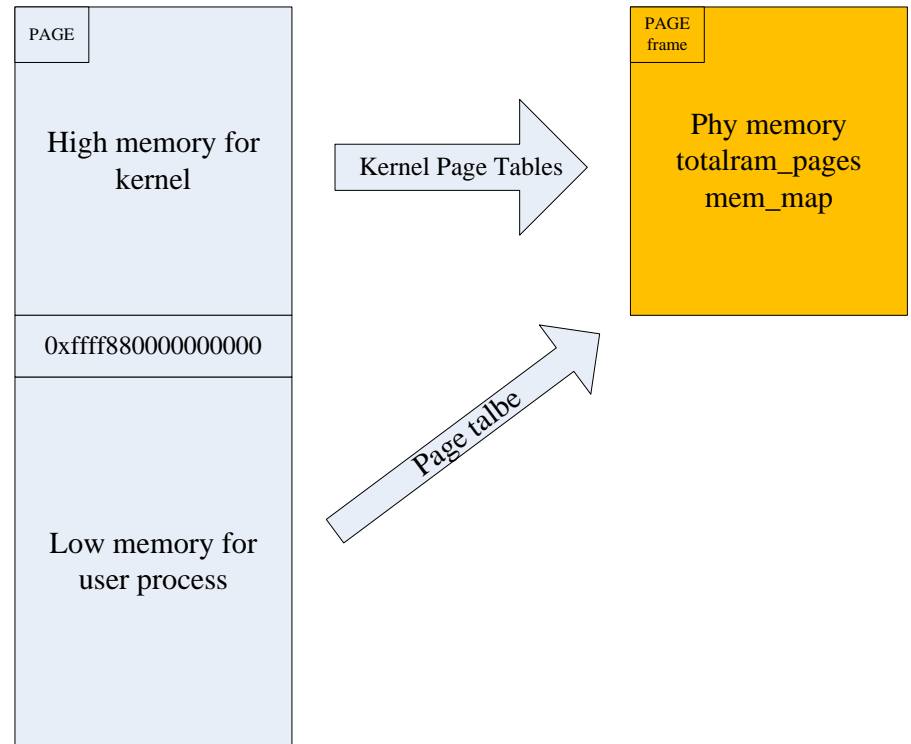


基本概念

- 读进程内存空间分配
- VFS
- 文件系统文件存放： ext2/ext3
- 磁盘高速缓存： 页高速缓冲（即磁盘中的数据以页的方式缓冲在RAM中）
- 块设备驱动
- IO调度

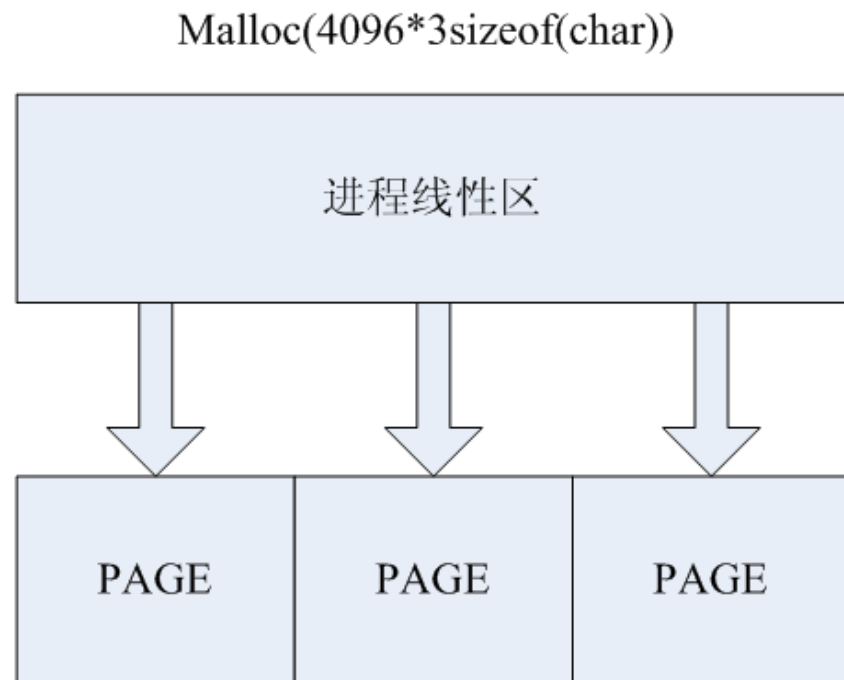
内存寻址

- 线性地址空间被分为低地址和高地址两个部分，分别被用于用户进程（当进程切换时，这部分地址空间的内容也随着改变了）和内核（这部分地址空间的内容总是一样的）。



进程地址空间

进程描述符中的[mm_struct](#)描述了进程地址空间的使用情况，进程使用的所有线性地址都是以线性区（页为单位的）[vm_area_struct](#)表示的，



Page & page frame

- 线性地址空间组织成页，这样可以提高访问效率，页内部地址映射为连续的RAM地址。页即表示一组线性地址也表示这组地址所存放的内容
- 物理地址被组织为页帧，由内核的内存管理模块负责管理，一个页表示以数据可以存储在任何地方（RAM或者磁盘——虚拟内存）。

请求调页vs.写时复制

- 访问的线性地址没有对应的物理地址存在，则向内存管理模块请求一个页帧（物理页），更新页表项，这个过程叫做请求调页。
- 写时复制备用来提高进程创建速度，父、子进程试图写共享页帧时产生异常，需要为进程分配新的页帧。

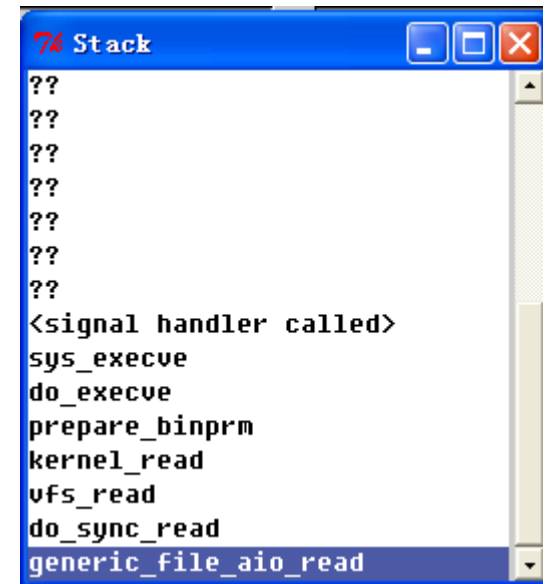
用户空间地址

- Malloc为用户申请线性区
- 调用Glibc封装的系统调用read——
kernel_read

```
Buf = malloc(BUFSIZE);  
read(fd, buf, BUFSIZE);
```

read内核态执行

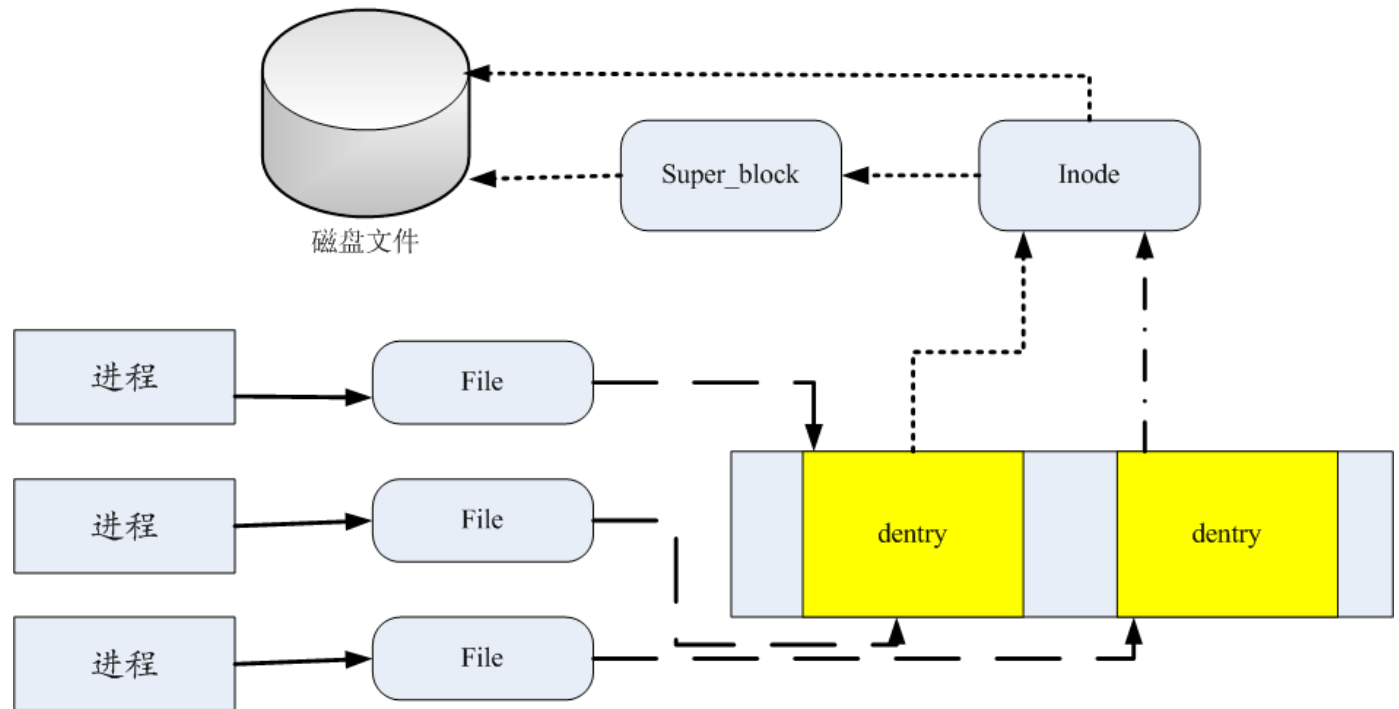
- 内核态执行read操作后的内核函数堆栈如图所示，
vfs_read后调用的是具体文件系统的read函数，但是大部分文件系统的read函数都是generic_file_aio_read。aio表示异步IO。



```
Stack
??
??
??
??
??
??
??
<signal handler called>
sys_execve
do_execve
prepare_binprm
kernel_read
vfs_read
do_sync_read
generic_file_aio_read
```

VFS

- Super_block
- Inode
- Dentry
- File

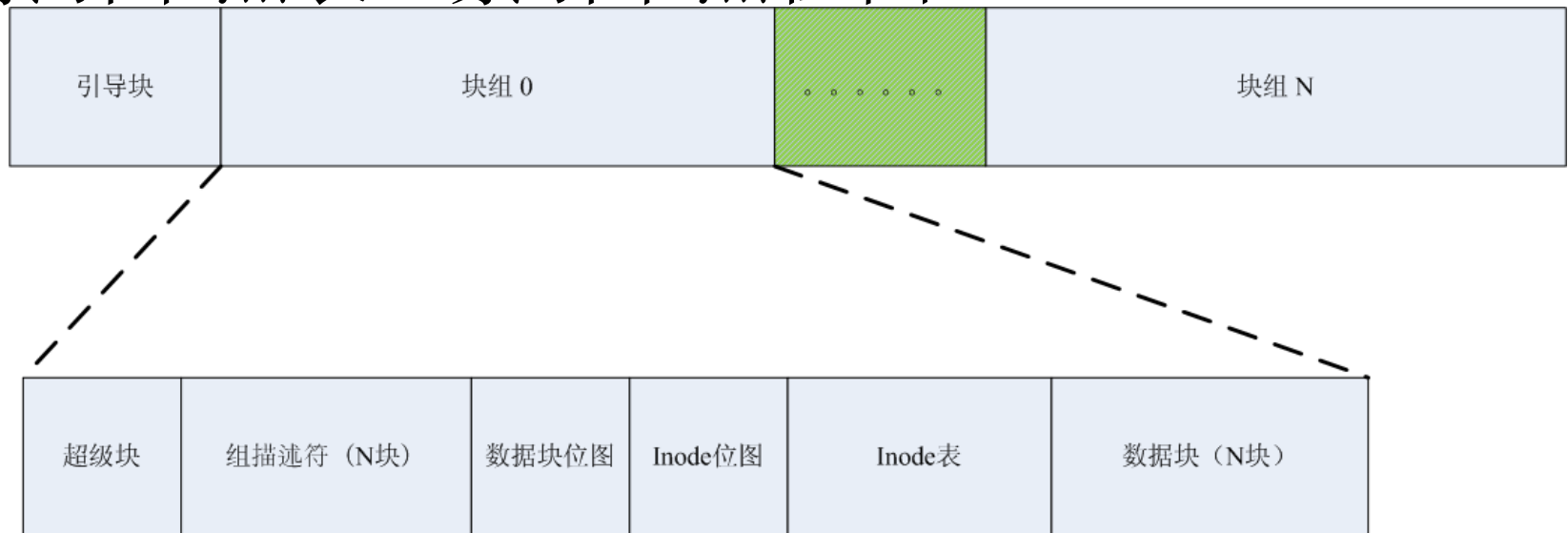


VFS

- **Super_block:** 包含文件系统的全局信息（块大小等）
- **Inode:** 文件的信息（大小，引用次数等）
- **Dentry**和**File**由内核负责构造，磁盘上不存
在相关数据结构

Ext2/ext3磁盘数据结构

- 超级块 组描述符 数据块 数据块位图
- 索引节点表 索引节点位图

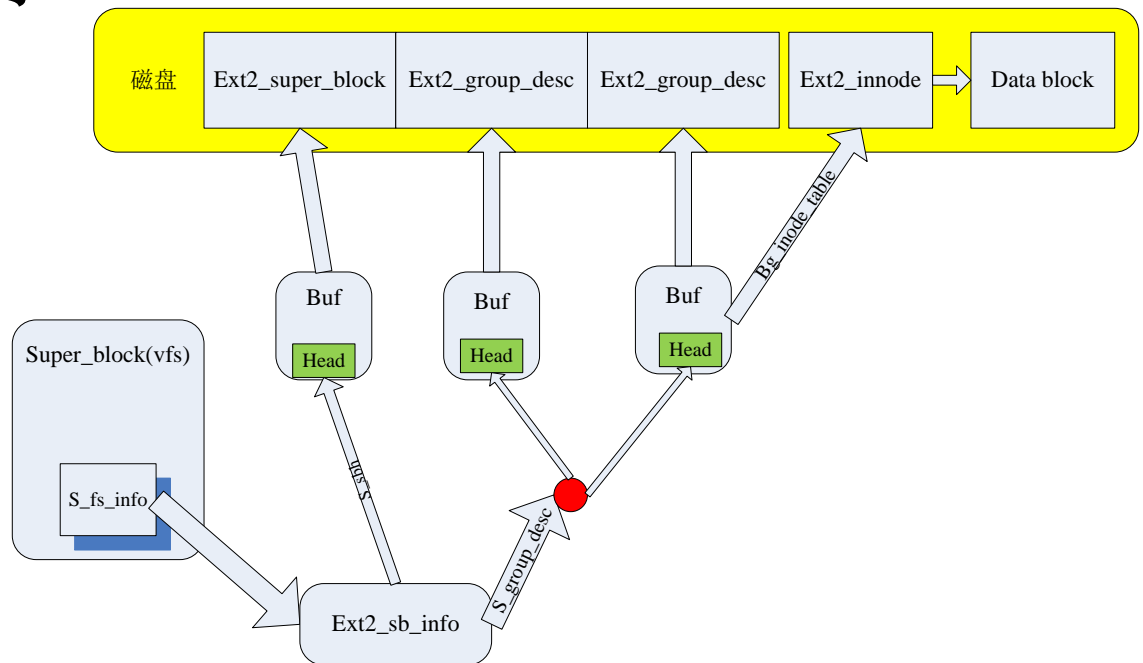


Ext2/ext3磁盘数据结构

- 超级块和组描述符在每个数据块组中都有相同的映像，通过超级块可以知道当前文件系统的信息，如索引节点的总数
- Ext2的inode大小相同，[ext2_inode](#)为128bytes，其中包括了一个描述逻辑上的一个数据组成单元所具有的性质，如修改时间，所有者ID，硬链接技术，指向数据库的指针（i_block），文件访问控制链表（ACL），文件大小等。通过VFS中[inode](#)的标号i_ino找到磁inode数据后就可以找到数据所分布的block了。

Ext2/ext3内存数据结构

- 使用磁盘数据结构初始化文件系统内存数据结构
- 以超级块为例：



struct ext2_super_block 和 struct ext2_sb_info:

- ext2_super_block是ext2文件系统超级块的磁盘数据结构，包括了表征磁盘数据分配的全局变量，如：inode总数，block的大小，inode数据结构的大小，文件系统大小（以块为单位），每个块组（group）包含的块数，每个块组包含的inode个数等。而ext2_sb_info是ext2超级块在内存中的数据结构，由于缓冲块的存在，这个结构不仅仅包括了ext2_super_block的大部分字段，还包括了指向内核磁盘缓冲区的超级块缓冲的头部。

- 通过ext2_sb_info可以直接诸如有多少个块组这样的信息，每个块组都有一个组描述符与之对应：[ext2_group_desc](#)，通过这个数据结构我们可以得到当前组所对应的inode table所在块的块号、块位图和inode位图。

- 以上所有的超级块和组描述符的数据都在内核安装文件系统时调用[ext2_fill_super](#)函数初始化并将ext2_sb_info的指针赋值给VFS中的超级块[super_block](#)的s_fs_info。这里面，每一个组描述符对应一个缓冲区头部，通过VFS的s_fs_info->ext2_sb_info->s_group_dest就可以获得组描述符进而获得inode table和inode bitmap、block bitmap。

文件系统操作

- 超级块的操作：用来对inode和超级块进行操作，如分配超级块、读取超级块等等
- Inode操作：包括目录、链接等操作，主要是修改inode属性
- 文件操作：文件打开、读、写操作，需要注意的是，文件操作很多都是不同文件系统通用的操作，如read操作使用generic_file_aio_read操作，只不过是在文件内部数据寻址的时候需要使用到具体文件系统的操作（文件中的偏移量转换为磁盘中的扇区号）。
- 如何从逻辑块号得到扇区号： $\text{blocks} \ll (\text{blkbits} - 9)$ 。也就是逻辑块号其实就是物理扇区（512-9）的倍数。

VFS & Ext2/ext3

- Ext2向VFS注册
- VFS使用ext2中的数据初始化超级块等数据结构
- 以inode为例：

VFS inode -> ext2_inode_info-> ext2_inode
EXT2_I宏可以索引VFS inode到ext2_inode_info
ext2_inode_info的vfs_inode字段指向VFS中的inode

找到磁盘文件数据

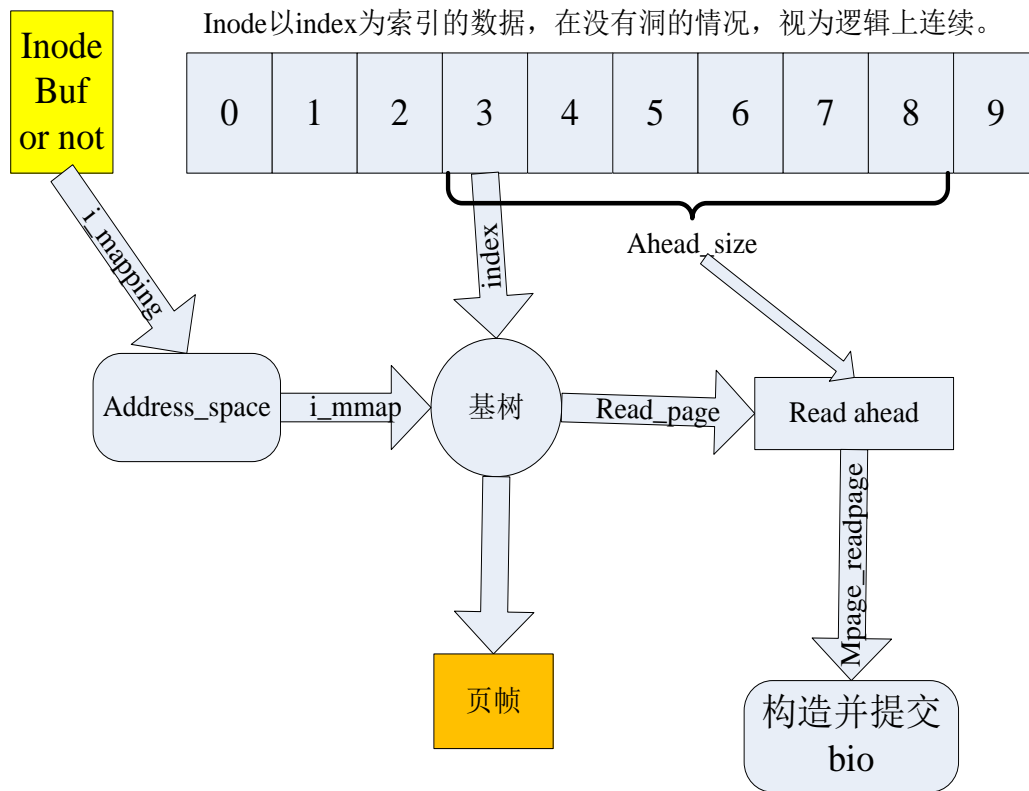
- 通过open操作找到文件对象
- 通过dentry找到inode
- 通过inode找到数据块索引

磁盘数据高速缓冲——页缓冲

- 为了了解读文件的过程，需要知道页高速缓冲的一些性质，缓存的目的就是减少IO次数，提高系统性能，页高速缓冲作为一种磁盘高速缓冲主要包括了对磁盘上的文件的
 - 普通数据内容的缓冲；
 - 磁盘块设备的元数据的缓冲（如超级块、inode、目录等）；

页缓存

- 以页为单位的文件视图
- 以inode作为文件的唯一标识
- Address_space对象
- 预读
- 直接将磁盘数据读取到页帧中



预读

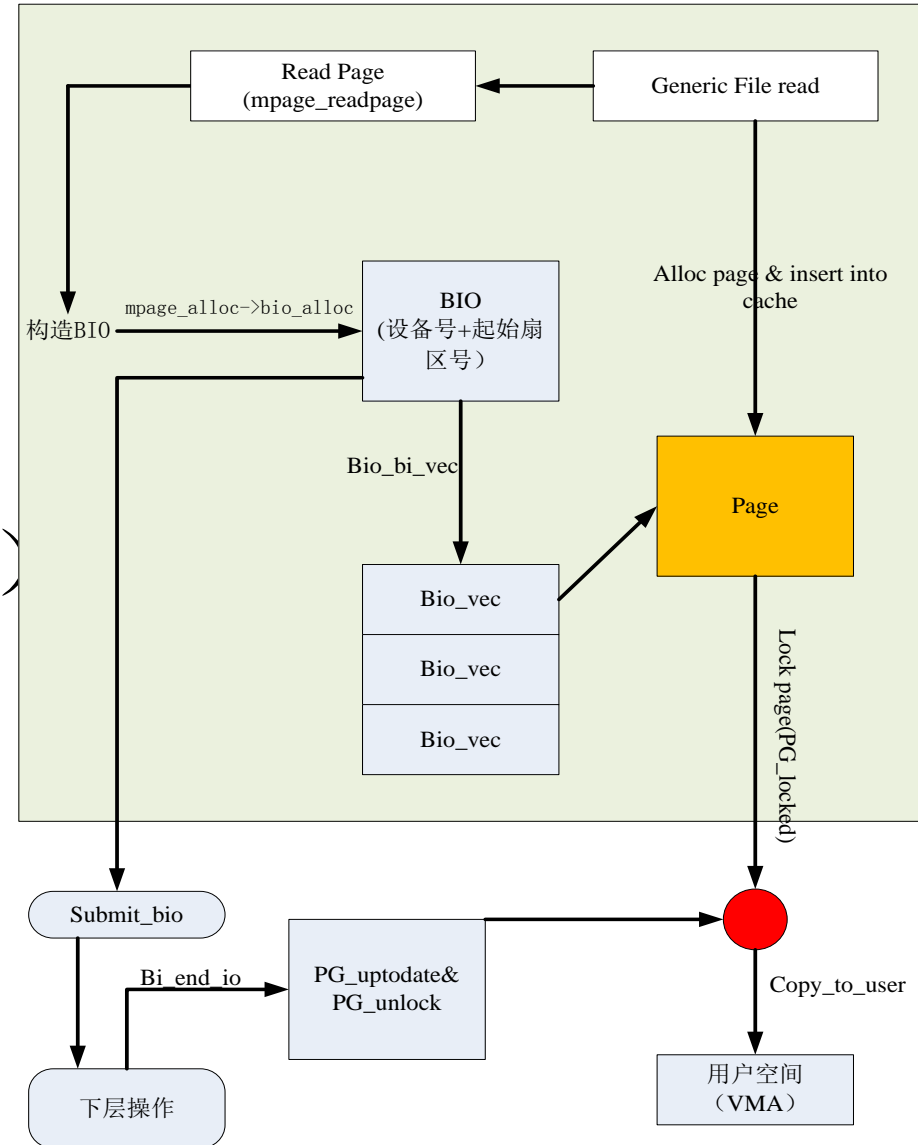
- 按需预读（由读磁盘数据页请求触发）：
 - 同步预读：发生对不在缓冲中的页的读取
 - 异步预读：异步设定读取窗口时，读到设定窗口页时进行预取

Generic_file_aio_read

- 当前磁盘页是否在页高速缓冲中
- 是否需要预读
 - 预读量
 - 读取页 `ext3_mpage_readpages(mpage_readpages)`
- 阻塞读进程，等到磁盘IO完成之后从页缓冲中拷贝到用户空间

Mpage_readpages

- Get_block
- 根据用户请求（磁盘文件页、页中偏移等）构造IO请求
- 构造BIO传往通用块层（submit_bio）



阻塞read用户

- 请求页帧后设置PG_locked，然后执行lock操作讲自己锁死在这个锁上，待请求的磁盘数据已经传送到页帧上面时，bi_end_io唤醒睡在页的PG_locked上面的进程。
- PG_updated表示当前页帧内的数据是否是最新的。

通用块层（构造通用的块IO请求）

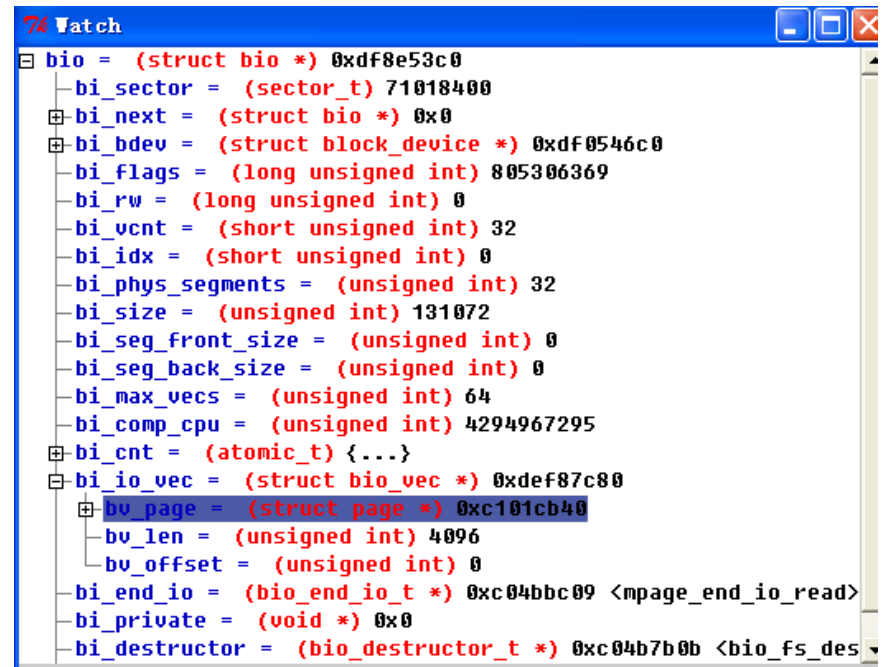
- 表示一次磁盘连续IO操作的数据结构为**bio**，**bio**在通用块层构造，构造好之后传递给I/O调度层，进而传递给块驱动层来处理。磁盘操作的最小单位是扇区，所以**bio**中使用起始扇区号和长度来表示这次IO操作（读or写）所涉及的物理区域。

通用块层（回忆概念）

- **块：**虽然物理读取磁盘是以扇区为单位的，但是VFS和文件系统组织磁盘内容是以**block**（块）为单位的，也就是上层发起的I/O操作是以块为单位的。
- **页：**页这个单位是磁盘数据缓冲的最小单元，也就是说页高速缓存缓存的是N个**block**的内容，比如超级块只有一个块，如果一页包含4块，那么一次读取不仅仅读取超级块内容还将读取这个块设备文件中超级块后面（组描述符）的3个**block**的内容。
- **段：**读取上来的数据的存放RAM区域称为段（**segment**）（以线性地址标示，内核空间）的连续地址空间中。

构造的BIO

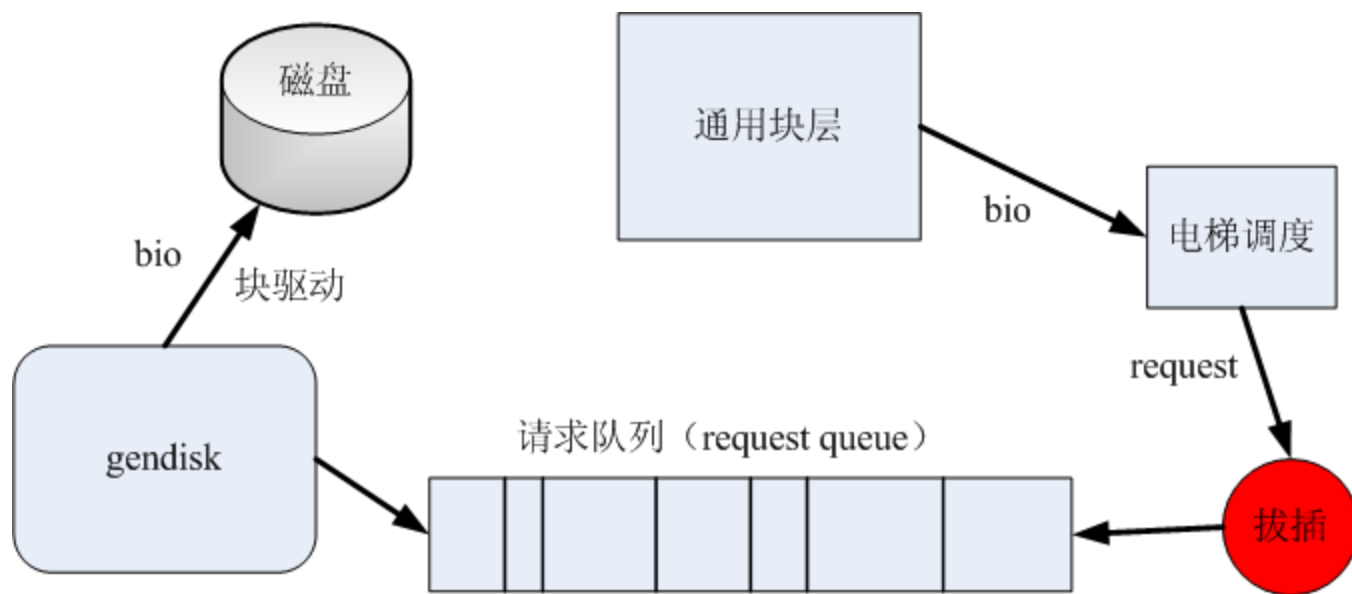
- bi_io_vec表示当前bio代表的磁盘数据需要读入的内存页：将数据放入到页：0xc101cb40 (0x101cb40)中
- bi_rw=0（读）
- bi_sector=71018400
- bi_end_io：读完成后的处理方案（因为读是硬件异步完成的，这个方法要负责更新页帧的一些标志位）



```
Watch
bio = (struct bio *) 0xdf8e53c0
  bi_sector = (sector_t) 71018400
  bi_next = (struct bio *) 0x0
  bi_bdev = (struct block_device *) 0xdf0546c0
  bi_flags = (long unsigned int) 805306369
  bi_rw = (long unsigned int) 0
  bi_vcnt = (short unsigned int) 32
  bi_idx = (short unsigned int) 0
  bi_phys_segments = (unsigned int) 32
  bi_size = (unsigned int) 131072
  bi_seg_front_size = (unsigned int) 0
  bi_seg_back_size = (unsigned int) 0
  bi_max_vecs = (unsigned int) 64
  bi_comp_cpu = (unsigned int) 4294967295
  bi_cnt = (atomic_t) {...}
  bi_io_vec = (struct bio_vec *) 0xdef87c80
    bv_page = (struct page *) 0xc101cb40
    bv_len = (unsigned int) 4096
    bv_offset = (unsigned int) 0
  bi_end_io = (bio_end_io_t *) 0xc04bbcb9 <mpage_end_io_read>
  bi_private = (void *) 0x0
  bi_destructor = (bio_destructor_t *) 0xc04b7b0b <bio_fs_des>
```

提交BIO——Generic_make_request

- Request: 连续的磁盘块组成的IO请求
- Gendisk: 磁盘（驱动模型）
- Request queue: 每个磁盘都有一个请求队列负责收集上层的IO请求



Bio & request

- 都表示连续磁盘块请求
- Bio表示可以使用一次磁盘传输完成（受到 `/sys/block/hda/queue/max_hw_sectors_kb` 影响）
- Request中可以包含扇区连续的多个bio，将多个bio合并到一个request由IO调度模块完成

I/O调度

- 默认的处理请求的程序 q->make_request_fn (默认为 make_request) 使用当前的gendisk的queue中设置的I/O调度算法进行调度，例如：

```
void elv_merged_request(struct request_queue *q, struct request
*rq, int type)
{
    struct elevator_queue *e = q->elevator;
    if (e->ops->elevator_merged_fn)
        e->ops->elevator_merged_fn(q, rq, type);
    if (type == ELEVATOR_BACK_MERGE)
        elv_rqhash_reposition(q, rq);
    q->last_merge = rq;
}
```

请求队列的拔插

- `__make_request`发现请求队列空则阻塞请求队列并启动一个回复插入队列的定时器
- 定时器（默认3ms）到时则清楚队列的阻塞，回复队列中IO请求的处理
- 带停顿的电梯算法

一个简单的块驱动

- 注册磁盘[gendisk](#)，gendisk代表一个磁盘，包括设备的基本信息，以及诸如磁盘请求队列的queue。
 - Device.gd = alloc_disk(16);
 - add_disk(Device.gd);
- 初始化队列queue: blk_queue_make_request(Queue, sbd_make_request);

```
static int sbd_make_request(struct request_queue *q, struct bio *bio)
{
    bio_for_each_segment(bvec, bio, i) {
        unsigned int len = bvec->bv_len;
        err = sbd_do_bvec(sbd, bvec->bv_page, len,
            bvec->bv_offset, rw, sector);
        if (err)
            break;
        sector += len >> SECTOR_SHIFT;
    }
    bio_endio(bio, err);
    return 0;
}
```

完整的read步骤

- Vfs open操作返回一个打开的文件的文件描述符，这个文件描述符作为current->files->fd数组的索引可以得到打开文件的file对象。
- 系统调用sys_read得到file对象之后通过该对象的f_op->read操作读取数据，通常file->f_op->read操作为文件系统对应的read操作。文件的inode在读入内存中时，vfs的inode结构中存放了初始的文件操作地址（不同文件系统自行定义），vfs使用inode中存储的操作地址初始化file的f_op字段。Ext2的文件操作（ULL3表18-10）列出，read操作对应的为generic_file_read操作，这个操作是一个内核通用的读操作。

完整的read步骤

- 读取文件的generic_file_aio_read函数初始化读参数后调用do_generic_file_read，该函数以index和offset分别表示文件地址空间中的页索引和页索引内的偏移。检查需求的页是否在缓存中，启动一个一页一页读取的循环，每次循环中进行预读等处理。发生缺页是调用的操作是address space中的readpage或者readpages。
- 具体文件系统的读取页操作readpage使用mpage_readpage方法，进而调用do_mpage_readpage方法，在这个方法中通过get_block获得页中每一块的逻辑块号。使用尽可能大的bio向通用块层提交bio。每次提交的bio的size不会超过max_sectors。比如如果忽略掉缓冲区不同，采用扩大初始bio并提交bio到通用块层。

完整的read步骤

- 通用块层的mpage_bio_submit中的submit_bio调用generic_make_request向I/O调度层提交bio，内核为每个进程保存一个未处理完的bio的队列（防止generic_make_request的调用递归进行用满内核栈），然后提交队列中的每一个bio。
- generic_make_request函数需要调用具体块设备驱动的make_request函数，但是大部分都是使用内核通用函数__make_request来处理，这个函数使用I/O调度算法来处理bio，对每个bio或者插入到一个request里面。
- 块设备驱动的其他函数负责从请求队列q里面处理request。

用户空间读取文件操作抽象表示

- 读取的文件file: filep
- 读取的用户缓冲区: 进程线性区
- 读取的文件内容: 磁盘文件页
- 读取的操作: VFS f_op -
>generic_file_aio_read
- 读取的文件位置: index & count & offset
- 监视读取状态: kiocb

磁盘数据存储结构

- VFS数据结构
- Ext2磁盘数据结构和内存数据结构
- 磁盘数据内容的页高速缓冲 address_space

构建用于向磁盘驱动提交的读请求

- 通用块层的bio
- 基于调度算法的bio合并
- 基于磁盘的请求队列
- 具体磁盘驱动处理请求队列中的请求
- 将数据传送到页高速缓冲

利用缓存提高系统读取文件的速度

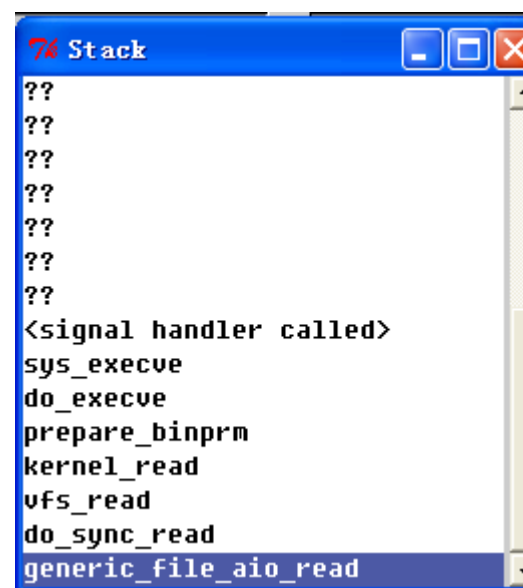
- 基于每个inode的address_space对象，缓冲每一个文件的页
- dentry和inode缓冲
- 预读

一次调试

- Linux 2.6.30.10
- vmware 7
- cygwin
- insight

- 使用test_io_simple进行读测试，开始调试时，我们发现，首先读入的是test_io_simple这个程序。
- 陷入系统调用后执行读的内核函数栈情况：
- 通用文件读写程序是大多数文件系统的读调用的程序：

```
1322 generic_file_aio_read(struct kiocb *iocb, const struct iovec *iov,  
1323                        unsigned long nr_segs, loff_t pos)  
- 1324 {  
■ 1325     struct file *filp = iocb->ki_filp;  
1326     ssize_t retval;  
1327     unsigned long seg;  
1328     size_t count;
```



filp

Filep为VFS的file结构，当前需要读取的文件是

test_io_simple

用户空间的需求：（读128字节）

```
▣ desc = (read_descriptor_t) {...}
  |
  | written = (size_t) 0
  | count = (size_t) 128
  |
  | arg = (union) {...}
  |
  | error = (int) 68
```

```
▣ filp = (struct file *) 0xde9093c0
  |
  | f_u = (union) {...}
  |
  | f_path = (struct path) {...}
  |
  | mnt = (struct vfsmount *) 0xdf40e8c0
  |
  | dentry = (struct dentry *) 0xc359a21c
  |
  | d_count = (atomic_t) {...}
  |
  | d_flags = (unsigned int) 0
  |
  | d_lock = (spinlock_t) {...}
  |
  | d_mounted = (int) 0
  |
  | d_inode = (struct inode *) 0xc359d4c8
  |
  | d_hash = (struct hlist_node) {...}
  |
  | d_parent = (struct dentry *) 0xc3502534
  |
  | d_name = (struct qstr) {...}
  |   |
  |   | hash = (unsigned int) 3288122613
  |   | len = (unsigned int) 14
  |   |
  |   | name = (const unsigned char *) 0xc359a278 "test_io_simple"
  |   |
  |   | d_lru = (struct list_head) {...}
  |   |
  |   | d_u = (union) {...}
  |   |
  |   | d_subdirs = (struct list_head) {...}
  |   |
  |   | d_alias = (struct list_head) {...}
  |   |
  |   | d_time = (long unsigned int) 0
  |   |
  |   | d_op = (const struct dentry_operations *) 0x0
  |   |
  |   | d_sb = (struct super_block *) 0xdf567600
  |   |
  |   | d_fscdata = (void *) 0x0
```

- 查找该文件内容是否已经在address_space(mapping)所对应的基树中（即有没有缓冲可用）：

```

1051             cond_resched();
1052 find_page:
- 1053             page = find_get_page(mapping, index);
- 1054             if (!page) {
- 1055                 page_cache_sync_readahead(mapping,
1056                                     ra, filp,

```

- 如果已经有页缓冲对应该文件页内容则直接拷贝到用户空间中（actor函数）：

```

1128             * pos here (the actor routine has to up
1129             * pointers and the remaining count).
1130             */
- 1131             ret = actor(desc, page, offset, nr);
- 1132             offset += ret;
- 1133             index += offset >> PAGE_CACHE_SHIFT;
- 1134             offset &= ~PAGE_CACHE_MASK;

```

- 拷贝完后释放掉该页引用计数：

```

1136             -
- 1137             page_cache_release(page);
- 1138             if (ret == nr && desc->count)
1139                 continue;

```

- 当页不在页缓冲中的时候，需要readahead来读取（预读）磁盘页：
（同步预读）

```

1052 find_page:
- 1053         page = find_get_page(mapping, index);
- 1054         if (!page) {
- 1055             page_cache_sync_readahead(mapping,
1056                                     ra, filp,
1057                                     index, last_index - index);
- 1058             page = find_get_page(mapping, index);
- 1059             if (unlikely(page == NULL))
1060                 goto no_cached_page;
1061         }

```

- 按需预读的同步预读算法：

```

446
447         /* do read-ahead */
- 448         ondemand_readahead(mapping, ra, filp, false, offset, req_size);
- 449     }

```

- 请求slab分配好页后，往页中读取磁盘数据：

```

193         */
- 194         if (ret)
195             read_pages(mapping, filp, &page_pool, ret);
- 196         BUG_ON(!list_empty(&page_pool));

```


- 调用address_space中指定的读取页的方法（具体到文件系统）：

```
112         int ret;  
113  
114         if (mapping->a_ops->readpages) {  
115             ret = mapping->a_ops->readpages(filp, mapping, pages, nr_pages);  
116             /* Clean up the remaining pages */  
117             put_pages_list(pages);  
...
```

- 我们可以看一下当前的读取的一些变量的值：

```
⊞ mapping = (struct address_space *) 0xc3552d58  
⊞ filp = (struct file *) 0xdeef7540  
   offset = (long unsigned int) 1592  
   nr_to_read = (long unsigned int) 32  
   lookahead_size = (long unsigned int) 32  
⊞ inode = (struct inode *) <value optimized out>  
⊞ page = (struct page *) 0xc126c6c0  
   end_index = (long unsigned int) 4351  
⊞ page_pool = (struct list_head) {...}  
   page_idx = (int) <value optimized out>  
   ret = (int) 32  
   isize = (loff_t) <value optimized out>
```

- 大部分文件系统实现的readpages都是使用mpage_readpages方法，这个方法又调用了do_mpage_readpage反复扩大提交的bio的大小，并利用具体文件系统的get_block方法得到读取当前文件页内的全部的块所对应的磁盘逻辑块的块号：如果能够扩大提交的bio就延缓提交bio，否则提交。

```
388         if (!add_to_page_cache_tru(page, mapping,  
389                                     page->index, GFP_KERNEL)) {  
390             bio = do_mpage_readpage(bio, page,  
391                                     nr_pages - page_idx,  
392                                     &last_block_in_bio, &map_bh,  
393                                     &first_logical_block,  
394                                     get_block);
```

```

159 * This is the worker routine which does all the work of mapping the disk
160 * blocks and constructs largest possible bios, submits them for IO if the
161 * blocks are not contiguous on the disk.
162 *
163 * We pass a buffer_head back and forth and use its buffer_mapped() flag to
164 * represent the validity of its disk mapping and to decide when to do the next
165 * get_block() call.
166 */
167 static struct bio *
168 do_mpage_readpage(struct bio *bio, struct page *page, unsigned nr_pages,
169                  sector_t *last_block_in_bio, struct buffer_head *map_bh,
170                  unsigned long *first_logical_block, get_block_t get_block)
171 {
172     struct inode *inode = page->mapping->host;
173     const unsigned blkbits = inode->i_blkbits;
174     const unsigned blocks_per_page = PAGE_SIZE >> blkbits;

```

- 提交bio:

```

1541 void submit_bio(int rw, struct bio *bio)
- 1542 {
- 1543     int count = bio_sectors(bio);
1544
- 1545     bio->bi_rw |= rw;
1546
1547     /*

```

- 提交bio之前更新统计信息（iostate即利用这个信息）：

```

1550     /*
1551     if (bio_has_data(bio)) {
- 1552         if (rw & WRITE) {
1553             count_vm_events(PGPGOUT, count);
1554         } else {
1555             task_io_account_read(bio->bi_size);
1556             count_vm_events(PGPGIN, count);
1557         }
1558     }

```

- 提交bio即调用通用块层的函数提交bio到具体设备的磁盘队列上去：

```

1489  */
1490 void generic_make_request(struct bio *bio)
1491 {
1492     if (current->bio_tail) {
1493         /* make_request is active */
1494         *(current->bio_tail) = bio;
1495         bio->bi_next = NULL;
1496         current->bio_tail = &bio->bi_next

```

- 调用具体队列中的方法处理提交的bio生产request并将request放到请求队列中：make_request_fn默认使用__make_request函数来处理。

```

1468     }
1469
1470     ret = q->make_request_fn(q, bio);
1471     } while (ret);
1472
1473     return;

```