```
ssize_t
vfs_read(struct *file, char __user *buf,
                 size_t count, loff_t *pos)
{
      ...
  filp = filp->f_op->read(filp, buf, count, pos);
      ...
}

struct file {
      ...
      struct file_operations *f_op;
      ...
};

struct file_operations ext2_file_operations = {
      ...
      .read = generic_file_read,
      .write = generic_file_write,
      ...
};
```

generic_file_read()主要解决文件同步操作和异步操作的问题。

```
ssize_t
generic_file_read(struct file *file, char __user *buf,
                            size_t count, loff_t *ppos)
{
  struct iovet local_iov = {
                .iov_base = buf,
                .iov_len = count
                };

  ...
  ret = __generic_file_aio_read(&kiocb, &local_iov, 1, ppos);
  ...
}
```

_generic_file_aio_read()函数读文件操作，输入参数iov包括用户传入的用户地址和希望读入的字节数。

```
ssize_t
__generic_file_aio_read(struct kiocb *iocb,
                        const struct iovec *iov,
                        unsigned long nr_segs, looff_t *ppos)
{

}
```

```
void do_generic_mapping_read(struct address_space *mapping,
                             struct filp_ra_state *_ra,
                             struct file *filp,
                             loff_t *ppos,
                             read_descriptor_t *desc,
                             read_actor_t actor)
{
  struct file_ra_state ra = *_ra;
  ...
  /*PAGE_CACHE_SHIFT = 12 (page size = 4KB)*/
  index = *ppos >> PAGE_CACHE_SHIFT;
  next_index = index;
  prev = ra.prev_page;
  last_index = (*ppos + desc->count + PAGE_CACHE_SIZE - 1)
                                      >> PAGE_CACHE_SHIFT;
  /*PAGE_MASK = 0x fffff 000*/
  offset = *ppos & ~PAGE_MASK;

  isize = i_size_read(inode);

  end_index = (i_size - 1) >> PAGE_CACHE_SHIFT;

  for(;;)
  {
    struct page *page;
    unsigned long nr, ret;

    nr = PAGE_CACHE_SIZE;
    if(index >= end_index)
    {
      if(index > end_index)
         goto out;
      nr = ((isize - 1) & ~PAGE_CACHE_MASK) + 1;
      if(nr < offset)
         goto out;
    }
  nr = nr - offset;

  cond_resched();
  if(index == next_index)
            next_index = page_cache_readahead(mapping, &ra,
                             filp, index, last_index - index);

  find_page:
  page_ok:
  ...
  no_cached_page:
  readpage:

}
```

当进行第一次循环的时候，调用page_cache_readahead()进入文件预读。然后进入检查page cache是否存在我们需要的页面。如果页面根本不存在，进入no_cached_page 的分支去申请一个页面，然后再去读这个页面。
no_cached_page 函数申请一个页面，然后将页面插入 page cache。如果成功，则进入readpage分支开始从硬盘读入数据。 readpage函数真正执行从硬盘读入一个页面的数据，以下是涉及的数据结构和具体函数

```
no_page_cached:
   if(!cached_page)
   {
     cached_page = page_cache_alloc_cold(mapping);
     if(!cached_page)
     {
       desc->error = -ENOMEM;
       goto out;
     }
   }
   error =  add_to_page_cache_lru(cached_page, mapping,
                                  index, GFP_KERNEL);
   ...
   page = cached_page;
   cached_page = NULL:
   goto readpage;
```

```
...
struct address_space *mapping;
...
readpage:
   error = mapping->a_ops->readpage(filp, page);
```

```
 struct address_space {
      struct inode     *host;
      ...
      struct address_space_operations *a_ops;
      ...
};
  /*fs.h*/
struct address_space_operations {
  ...
  int (*writepage)(struct page *,
                   struct writeback_control *wbc);
  int (*readpage)(struct file *filp,struct page *);
  ...
}
```

从银盘读取数据是readpage 函数实现的。针对ext2文件系统提供的读页面函数是ext2_readpage()。 do_mpage_readpage()将度请求转换为一个bio结构，如果bio有效，则提交bio给底层去执行读操作。

```
static int ext2_readpage(struct file *file, struct page *page)
{
  return mpage_readpage(page, ext2_get_block);
}/*fs/ext2/inode.c*/
/**/
int mpage_readpage(struct page *page, get_block_t get_block)
{
  struct bio *bio = NULL;
  unsigned long first_logical_block = 0;
  sector_t last_bl
  ...
  bio = do_mpage_readpage(bio, page, 1, &last_block_in_bio,
                      &map_bh, &first_logical_block, get_block);
  if(bio)
  {
     mpage_bio_submit(READ, bio);
  }
}
```

do_mpage_readpage()调用get_block()函数，该函数是文件系统提供的映射函数，可以根据逻辑块号获得硬盘的物理块号。

```
static struct bio *do_mpage_readpage(struct bio *bio,
              struct page *page, unsigned nr_pages,
              sector_t *last_block_bio,
              struct buffer_head *map_bh,
              unsigned long *first_logic_block,
              get_block_t get_block)
{
  sector_t block_in_file;
  /* page->index - the offset of the file in the memory address
                   unit is page
     PAGE_CACHE_SHIFT - the bit count of page size (12 - 4KB)
     blkbit- the bit count of block size(10 - 1KB)
     block_in_file is the first block number of the file
  */
  block_in_file = (sector_t)page->index << (PAGE_CACHE_SIZE - blkbits);
  get_block(inode, block_in_file, map_bh, 0);
}

static int get_block(struct inode *inode, sector_t iblock,
                   struct buffer_head *bh_result, int creat)
{
   unsigned max_blocks = bh_result->b_size >> inode->i_blkbits;
   int ret = ext2_get_blocks(inode, iblock, max_blocks,
                                  bh_result, create);
         ...
}

static int ext2_get_blocks(struct inode *inode,
             sector_t iblock, unsigned long maxblocks,
             struct buffer_head *bh_result, int create)
{
  ...
  ext2_block_to_path(inode,iblock,offset,&block_to_boundary);
  ...
}

static int ext2_block_to_path(struct inode *inode, long i_block
                                  int offset[4],int *boundary)
{
  ...
  int depth;
  /*EXT2_NDIR_BLOCK = 12*/
  const long direct_blocks = EXT2_NDIR_BLOCKS;

  depth = ext2_block_to_path(inode, iblock, offset,
                                  &block_to_boundary);
  ...
}
```

从文件内的偏移量f导出相应数据快的逻辑块号需要以下两个步骤:

- 从偏移量f导出文件的块号，即在偏移量f处的字符所在的块索引
- 把文件的块号转化为相应的逻辑块号

导出文件的第f个字符所在的文件块号是相当容易的，只要用f除以文件系统块的大小，并取整即可。 例如，假设块的大小为4KB，如果f小于4096，那么这个字符就在文件的第一个数据块中，其中文件 的块号为0。如果f等于或者大于4096而小于8192，则这个字符就在文件块号为1的数据块中，以此 类推。
由于Ext2文件的数据块在磁盘上不必是相邻的，因此吧文件的块号转化为相应的逻辑块号并不是直接 了当的。

关于head_buffer 中的b_blocknr 存放逻辑号，即块在磁盘或分区中的编号。现在磁盘讲它们的结构呈现为一个简单的视图，一个B个扇区大小的逻辑块的序列，编号为0,1,...,B-1.磁盘中有一个小的硬件/固件控制设备，称为磁盘控制器，维护这逻辑块号和实际物理磁盘扇区之间的映射关系。