Linux内核读文件过程

http://www.ilinuxkernel.com

目　录Table of Contents

图目录 List of Figures

1 概述

我们对系统调用read（）非常熟悉，也常听说“零拷贝”。在看Linux内核源码时，有很多人会有一些困惑；比如读文件的整个流程是怎样的？内核是如何Cache已经读取的文件数据？驱动从磁盘上读取的数据是否会直接写到用户的缓冲区中？内核是在哪个地方分配空间来存储将要读取的数据？是在哪个地方将当前进程阻塞，直至读取数据结束？“零拷贝”是如何实现的？

本文以Redhat Enterprise Linux 6 Update 3内核版本2.6.32-279.el6.x86_64为例，分析从用户进程通过read（）读取文件，直至数据返回给用户的整个流程。

在我们分析源码之前，仍要回顾一下内核中块设备操作的流程，如图1所示。

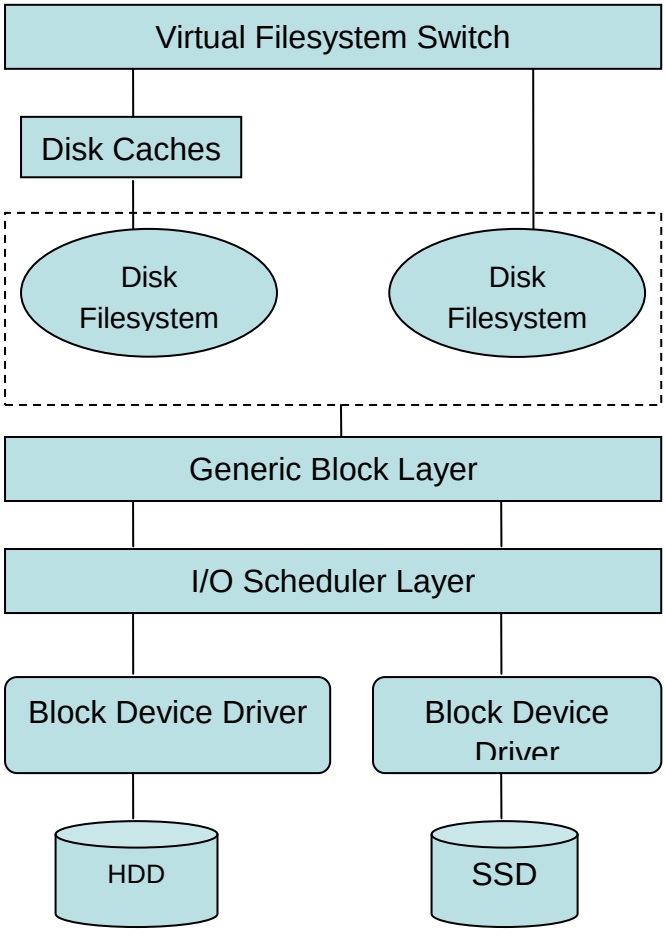

图1 内核中块设备操作流程

对于一个进程使用系统调用read（）读取磁盘上的文件。下面步骤是内核响应进程读请求的

步骤；

（1）系统调用read（）会触发相应的VFS（Virtual Filesystem Switch）函数，传递的参数有文件描述符和文件偏移量。

（2）VFS确定请求的数据是否已经在内存缓冲区中；若数据不在内存中，确定如何执行读操作。

（3）假设内核必须从块设备上读取数据，这样内核就必须确定数据在物理设备上的位置。这由映射层（Mapping Layer）来完成。

（4）此时内核通过通用块设备层（Generic Block Layer）在块设备上执行读操作，启动I/O操作，传输请求的数据。

（5）在通用块设备层之下是I/O调度层（I/O Scheduler Layer），根据内核的调度策略，对等待的I/O等待队列排序。

（6）最后，块设备驱动（Block Device Driver）通过向磁盘控制器发送相应的命令，执行真正的数据传输。

对于（1）、（2）两个步骤，在《Linux虚拟文件系统》中，我们讨论了VFS（Virtual Filesystem Switch）主要数据结构和操作，结合相关系统调用（如sys_read（）、sys_write（）等）的源码，我们不难理解VFS层相关的操作和实现。而对于第（3）步中的Mapping Layer需要结合具体的文件系统解释，我们暂时不考虑。

在《Linux通用块设备层》中，我们分析了第（4）步。上层的I/O请求发送到通用块设备层（Generic Block Layer）后，就会将请求继续传送到I/O调度层（I/O Scheduler Layer）。

在《Linux 内核I/O调度层》中，我们分析了第（5）步。管理块设备的请求队列，它决定队列中的请求排序以及何时派发请求块设备。《Linux块设备驱动》分析了块设备驱动是如何处理I/O请求及与硬件进行数据交互过程。

本文分析read（）系统调用的整个过程，包括VFS、通用块设备层、I/O调度层、块设备驱动等内容。大家可以随时参照图4 cfq_insert_request（）函数调用栈，图5 mpt_put_msg_frame（）函数调用栈，这样可以明确函数调用关系和层次。

2 虚拟文件系统与ext4文件系统层

在分析内核源码之前，回顾一下read（）函数原型。

ssize_t read(unsigned int fd, char * buf, size_t count)

read（）函数是从打开的文件中读取数据。如read（）成功，则返回读到的字节数。如已到达文件的尾端，则返回0。如果失败，则返回－1。有多种情况可使实际读到的字节数少于要求读字节数：

- 读普通文件时，在读到要求字节数之前已到达了文件尾端。例如，若在到达文件尾端之前还有50个字节，而要求读200个字节，则read返回50，下一次再调用read时，它将返回0 (文件尾端)。
- 当从终端设备读时，通常一次最多读一行。
- 当从网络读时，网络中的缓冲机构可能造成返回值小于所要求读的字节数。
- 某些面向记录的设备，例如磁带，一次最多返回一个记录。

读操作从文件的当前位移量处开始，在成功返回之前，该位移量增加实际读得的字节数。

2.1 sys_read（）

sys_read（）是系统调用read（）的实现，源码在文件fs/read_write.c中。

```
00372: SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf,
                       size_t, count)
00373: {
00374:     struct file *file;
00375:     ssize_t ret = -EBADF;
00376:     int fput_needed;
00377:
00378:     file = fget_light( fd, &fput_needed);
00379:     if (file) {
00380:         loff_t pos = file_pos_read(file);
00381:         ret = vfs_read(file, buf , count, &pos);
00382:         file_pos_write(file,    pos);
00383:         fput_light(file, fput_needed);
00384:     }
00385:
00386:     return ret;
00387: }
00388:
```

在sys_read（）函数中，实现主体是vfs_read（），也就是read（）函数的实现主体。file_pos_read（）和file_pos_write（）的功能单一，分别是读取/更新当前文件的读写位置；file_pos_write（）是根据读文件结果，更新文件读写位置。这两个函数源码均在fs/read_write.c中。

fget_light（）返回打开文件的file结构。

若files_struct为多进程/线程共享（fput_neede=1），fput_light（）就调用fput（）减少共享计数。

2.1.1 fget_light（）和fput_light（）

1. fget_light（）

在4.6版本中，变成__fget_light和__fput_light

fget_light（）根据打开文件号fd找到该已打开文件的file结构，源码在fs/file_table.c中。我们知道在read文件前，需要先调用open（）打开文件，打开文件后，就会有相应的文件描述符fd及file对象。

```
00291: /*
00292:  * Lightweight file lookup - no refcnt increment if fd table isn't shared.
00293:  * You can use this only if it is guranteed that the current task already
00294:  * holds a refcnt to that file. That check has to be done at fget() only
00295:  * and a flag is returned to be passed to the corresponding fput_light().
00296:  * There must not be a cloning between an fget_light/fput_light pair.
00297:  */
00298: struct file *fget_light(unsigned int  fd, int *fput_needed)
00299: {
00300:     struct file *file;
00301:     struct files_struct *files = current->files;
00302:
00303:     *fput_needed = 0;
00304:     if (likely((atomic_read(&files->count) == 1))) {
00305:         file = fcheck_files(files,  fd);
00306:     } else {
00307:         rcu_read_lock();
00308:         file = fcheck_files(files,  fd);
00309:         if (file) {
00310:             if (atomic_long_inc_not_zero(&file->f_count))
00311:             *fput_needed = 1;
00312:             else
00313:             /* Didn't get the reference, someone's freed */
00314:               file = NULL;
00315:         }
00316:         rcu_read_unlock();
00317:     }
00318:
00319:     return file;
00320: } « end fget_light »
00321:
```

对于fget_light（）中参数fput_needed，在该函数注释中已说明；若该进程fd表没有共享，则不用增加计数。而判断fd表是否被共享，是通过读取file->f_count的值，若该值不等于1，则

表明fd表是共享的（count是至少是1）。进程fd表共享的话，就将fput_needed的值设为1（311行），且此时已通过atomic_long_inc_not_zero（）增加了文件对象引用计数。

找到已打开文件的file结构非常简单，通过fcheck_files（）函数获取，源码在include/linux/fdtable.h中。对于files_struct和file数据结构，可以参考《Linux虚拟文件系统》章节。files->fd[fd]即是已打开文件的file结构。

2. fput_light（）

若进程fd表是共享的，则fget_light（）函数中将fput_needed的值设为1，且增加了file结构的文件对象引用计数；读取文件数据结构后，就通过fput（）来递减文件对象引用计数，该inline函数源码在include/linux/file.h中。

```
00025: static inline void fput_light(struct file *file, int fput_needed)
00026: {
00027:     if (unlikely(fput_needed))
00028:         fput(file);
00029: }
00030:
```

2.2 vfs_read（）

sys_read（）是系统调用read（）的实现，实现主体是vfs_read（），也就是read（）函数的实现主体。源码在文件fs/read_write.c中。

该函数的的4个参数；file是已打开文件的file结构，该结构是由fd而来；buf是用户空间的缓冲区，就是内核将读取到的数据拷贝到这里；count是将要读取的字节数；pos是文件当前读写位置。

```
00277: ssize_t vfs_read(struct file *file, char __user *buf, size_t count,
                        loff_t *pos)
00278: {
00279:     ssize_t ret;
00280:
00281:     if (!(file->f_mode & FMODE_READ))
00282:         return -EBADF;
00283:     if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
00284:         return -EINVAL;
00285:     if (unlikely(! access_ok(VERIFY_WRITE, buf, count)))
00286:         return -EFAULT;
00287:
00288:     ret = rw_verify_area(READ, file, pos, count);
00289:     if (ret >= 0) {
00290:         count = ret;
00291:         if (file->f_op->read)
```

```
00292:              ret = file->f_op->read(file,   buf, count, pos);
00293:          else
00294:              ret = do_sync_read(file,   buf, count, pos);
00295:          if (ret > 0) {
00296:              fsnotify_access(file->f_path.dentry);
00297:              add_rchar(current,   ret);
00298:          }
00299:          inc_syscr(current);
00300:      }
00301:
00302:      return ret;
00303: } « end vfs_read »
00304:
```

在后面我们会看到vfs_read（）函数调用的是do_sync_read（），可以理解为vfs_read（）就是对do_sync_read（）函数的封装，在调用do_sync_read（）函数之前进行一些合法性检查。

281行是检查文件的访问模式，若不允许读，则返回-EBADF。在执行真正的读操作之前，文件操作表file->f_op不能为空，并且文件操作表中read或aio_read方法必须要有一个有实现；否则就返回-EINVAL（283行）。

access_ok（）函数检查用户态的buf是否有效（285行），VERIFY_WRITE表示用户buf是否可写。VERIFY_WRITE的优先级高于VERIFY_READ，因为用户的buf可写的话，必然可读。

rw_verify_area（）检查要访问的文件部分是否有冲突的强制锁，通过inode结构lock当前要操作的区域。文件系统是否允许使用强制锁，可以在mount时指定，如果mount的时候指定了MS_MANDLOCK，则允许使用强制锁。

通过上面合法性检查后就是读操作本身了。可想而知，不同的文件系统有不同的读操作，具体的文件系统通过file_operations结构提供用于读操作的函数指针。如果读操作的返回值大于0，说明读数据成功，且ret值为读取的字节数。则调用fsnotify_access（）通知文件被读取，add_rchar（）增加当前进程读取的字节数（295～298行）。

inc_syscr（）为增加当前进程read系统调用的次数。

```
00215: int rw_verify_area(int read_write, struct file *file, loff_t *ppos,
size_t count)
00216: {
00217:      struct inode *inode;
00218:      loff_t pos;
00219:      int retval = -EINVAL;
00220:
00221:      inode = file->f_path.dentry->d_inode;
00222:      if (unlikely((ssize_t)  count < 0))
00223:          return retval;
00224:      pos = *ppos;
00225:      if (unlikely((pos  < 0) || (loff_t) (pos + count) < 0))
00226:          return retval;
```

```
00227:
00228:     if (unlikely(inode->i_flock && mandatory_lock(inode))) {
00229:         retval = locks_mandatory_area(
00230:             read_write == READ ? FLOCK_VERIFY_READ  :
                                FLOCK_VERIFY_WRITE,
00231:             inode, file, pos, count);
00232:         if (retval < 0)
00233:             return retval;
00234:     }
00235:     retval = security_file_permission(file,
00236:                 read_write == READ ? MAY_READ : MAY_WRITE);
00237:     if (retval)
00238:         return retval;
00239:     return count > MAX_RW_COUNT ? MAX_RW_COUNT : count;
00240: } « end rw_verify_area »
00241:
```

这里我们是以ext4文件系统为例，来看一下ext4文件系统的file_operations结构，其定义在fs/ext4/file.c文件中。可以看到ext4文件系统中file->f_op->read的实现方法为do_sync_read()。

```
00234: const struct file_operations ext4_file_operations = {
00235:     .llseek         = ext4_llseek,
00236:     .read           = do_sync_read,
00237:     .write          = do_sync_write,
00238:     .aio_read       = generic_file_aio_read,
00239:     .aio_write      = ext4_file_write,
00240:     .unlocked_ioctl = ext4_ioctl,
00241: #ifdef CONFIG_COMPAT
00242:     .compat_ioctl   = ext4_compat_ioctl,
00243: #endif
00244:     .mmap           = ext4_file_mmap,
00245:     .open           = ext4_file_open,
00246:     .release = ext4_release_file,
00247:     .fsync          = ext4_sync_file,
00248:     .splice_read    = generic_file_splice_read,
00249:     .splice_write   = generic_file_splice_write,
00250: };
```

2.3 do_sync_read（）

2.3.1 异步I/O

Linux中最常用的输入/输出（I/O）模型是同步 I/O。在这个模型中，当请求发出之后，应用程序就会阻塞，直到请求满足为止。同步I/O场景下，调用应用程序在等待 I/O 请求完成时不需要使用任何CPU执行。但是在某些情况中，I/O 请求可能需要与其他进程产生交叠。可移植操作系统接口（POSIX）异步 I/O应用程序接口就提供了这种功能。

异步I/O背后的基本思想是允许进程发起很多 I/O 操作，而不用阻塞或等待任何操作完成。稍后或在接收到 I/O 操作完成的通知时，进程就可以检索 I/O 操作的结果。异步 I/O 允许用户空间来初始化操作而不必等待它们的完成； 因此， 一个应用程序可以在它的 I/O 在进行中时做其他的处理。一个复杂的、高性能的应用程序还可使用异步 I/O 来使多个操作在同一

个时间进行。

实现异步I/O操作的file_operations方法，都使用kiocb控制块（I/O Control Block）。其定义在include/linux/aio.h文件中。

```
00087: struct kiocb {
00088:     struct list_head    ki_run_list;
00089:     unsigned long       ki_flags;
00090:     int                 ki_users;
00091:     unsigned            ki_key;         /* id of this request */
00092:
00093:     struct file         *ki_filp;
00094:     struct kioctx       *ki_ctx;    /* may be NULL for sync ops */
00095:     int                 (*ki_cancel)(struct   kiocb *, struct io_event *);
00096:     ssize_t             (*ki_retry)(struct  kiocb *);
00097:     void                (*ki_dtor)(struct  kiocb *);
00098:
00099:     union {
00100:         void __user         *user;
00101:         struct task_struct  *tsk;
00102:     } ki_obj;
00103:
00104:     __u64               ki_user_data;     /* user's data for completion */
00105:     wait_queue_t        ki_wait;
00106:     loff_t              ki_pos;
00107:
00108:     void                *private;
00109:     /* State that we remember to be able to restart/retry  */
00110:     unsigned short      ki_opcode;
00111:     size_t              ki_nbytes;      /* copy of iocb->aio_nbytes */
00112:     char                __user *ki_buf;       /* remaining iocb->aio_buf */
00113:     size_t              ki_left;      /* remaining bytes */
00114:     struct iovec        ki_inline_vec;     /* inline vector */
00115:     struct iovec        *ki_iovec;
00116:     unsigned long       ki_nr_segs;
00117:     unsigned long       ki_cur_seg;
00118:
00119:     struct list_head    ki_list;      /* the aio core uses this
00120:                          * for cancellation */
00121:
00122:     /*
00123:      * If the aio_resfd field of the userspace iocb is not zero,
00124:      * this is the underlying eventfd context to deliver events to.
00125:      */
00126:     struct eventfd_ctx *ki_eventfd;
00127: } « end kiocb » ;
```

结构体struct kiocb中主要成员变量：

ki_run_list：将要重试的I/O操作链表；

ki_flags：kiocb描述符的标志；

ki_users：kiocb描述符的使用计数；

ki_key：异常I/O操作的标志，若值设为KIOCB_SYNC_KEY，则表示为同步I/O；

ki_filp：指向与即将进行I/O操作关联的file对象；

ki_ctx：指向当前操作的异步I/O环境描述符（context descriptor）；

ki_cancel：取消异步I/O时所调用的方法；

ki_retry：重试异步I/O操作所调用的方法；

ki_dtor：销毁kiocb描述符的方法；

ki_list：指向异步I/O环境中，活动的、即将进行的I/O操作链表；

ki_obj：对于同步操作，它指向发起I/O的进程描述符；对于异步操作，它指向用户空间的iocb数据结构；

ki_user_data：返回给用户进程的值；

ki_pos：当前文件读写位置；

ki_opcode：操作类型（read、write或者sync）；

ki_nbytes：将要传输的字节数；

ki_buf：用户空间缓存区当前位置；

ki_left：还未完成传输的字节数；

ki_wait：异步I/O操作的等待队列；

private：文件系统层使用。

kiocb结构的意义在于使能异步操作。若驱动能够初始化这个操作（或者简单地，将它排队到它能够被执行时），它必须做两件事情：

（1）记录它需要知道的关于这个操作的所有东西， 并且返回 -EIOCBQUEUED 给调用者。

（2）记录操作信息包括安排对用户空间缓冲的存取；当在调用进程的上下文运行时，一旦返回，将不能再来存取缓冲。

2.3.2 do_sync_read（）

do_sync_read（）函数的实现主体在文件fs/read_write.c中。

```
00252: ssize_t do_sync_read(struct file *filp, char __user *buf, size_t len,
loff_t *ppos)
00253: {
00254:     struct iovec iov = { .iov_base = buf, .iov_len = len };
00255:     struct kiocb kiocb;
00256:     ssize_t ret;
00257:
00258:     init_sync_kiocb(&kiocb, filp);
```

```
00259:     kiocb.ki_pos = *ppos;
00260:     kiocb.ki_left = len;
00261:
00262:     for (;;) {
00263:         ret = filp->f_op->aio_read(&kiocb, &iov, 1, kiocb.ki_pos);
00264:         if (ret != -EIOCBRETRY)
00265:             break;
00266:         wait_on_retry_sync_kiocb(&kiocb);
00267:     }
00268:
00269:     if (-EIOCBQUEUED == ret)
00270:         ret = wait_on_sync_kiocb(&kiocb);
00271:     *ppos = kiocb.ki_pos;
00272:     return ret;
00273: } « end do_sync_read »
00274:
00275: EXPORT_SYMBOL(do_sync_read);
00276:
```

函数中有个临时变量kiocb，它是用来跟踪记录即将进行I/O操作的完成状态。通过宏init_sync_kiocb（）来初始化kiocb，设置对象为同步操作；就是将ki_key设为KIOCB_SYNC_KEY，ki_filp设为filp，而将将ki_obj.task的值设为current。

宏init_sync_kiocb（）的实现在文件include/linux/aio.h中。

```
00129: #define is_sync_kiocb(iocb)     ((iocb)->ki_key == KIOCB_SYNC_KEY)
00130: #define init_sync_kiocb(x, filp)            \
00131:     do {                                    \
00132:         struct task_struct *tsk = current;   \
00133:         (x)->ki_flags = 0;                  \
00134:         (x)->ki_users = 1;                  \
00135:         (x)->ki_key = KIOCB_SYNC_KEY;       \
00136:         (x)->ki_filp = (filp);              \
00137:         (x)->ki_ctx = NULL;                 \
00138:         (x)->ki_cancel = NULL;              \
00139:         (x)->ki_retry = NULL;               \
00140:         (x)->ki_dtor = NULL;                \
00141:         (x)->ki_obj.tsk = tsk;              \
00142:         (x)->ki_user_data = 0;              \
00143:         init_wait((&(x)->ki_wait));         \
00144:     } while (0)
00145:
```

269～270行表示异步I/O操作还没完成，就调用wait_on_sync_kiocb(&kiocb)等待I/O操作完成。

ext4文件系统中file->f_op->read的实现方法为generic_file_aio_read（）（见ext4_file_operations定义）。

2.4 generic_file_aio_read（）

generic_file_aio_read（）函数实现在文件mm/filemap.c中。该函数是read（）系统调用执行主体，且所有文件系统共用。该函数实现同步和异步的读操作。

函数有4个参数，iocb和iov在前面一节已作介绍，nr_segs的值事实上为1（do_sync_read（）参数传递），ppos是用来保存文件当前读写位置。

```
01332: /**
01333: * generic_file_aio_read - generic filesystem read routine
01334: * @iocb:     kernel I/O control block
01335: * @iov: io vector request
01336: * @nr_segs:      number of segments in the iovec
01337: * @pos:      current file position
01338: *
01339: * This is the "read()" routine for all filesystems
01340: * that can use the page cache directly.
01341: */
01342: ssize_t
01343: generic_file_aio_read(struct kiocb *iocb, const struct iovec
01344:             *iov, unsigned long  nr_segs, loff_t  pos)
01345: {
01346:     struct file *filp = iocb->ki_filp;
01347:     ssize_t retval;
01348:     unsigned long seg = 0;
01349:     size_t count;
01350:     loff_t *ppos = &iocb->ki_pos;
01351:
01352:     count = 0;
01353:     retval = generic_segment_checks(iov, &nr_segs, &count, VERIFY_WRITE);
01354:     if (retval)
01355:         return retval;
01356:
```

1353行合法性检查。generic_segment_checks（）用来验证iovec描述的用户空间缓冲区是有效的。因为读文件的起始地址和大小都是由sys_read（）传递过来，在使用它们之前，必须要先进行合法性检查。若参数无效，则返回-EFAULT。

```
01357:     /* coalesce the iovecs and go direct-to-BIO for O_DIRECT */
01358:     if (filp->f_flags & O_DIRECT) {
01359:         loff_t size;
01360:         struct address_space *mapping;
01361:         struct inode *inode;
01362:
01363:         mapping = filp->f_mapping;
01364:         inode = mapping->host;
01365:         if (!count)
01366:             goto ↓out; /* skip atime */
01367:         size = i_size_read(inode);
01368:         if (pos < size) {
01369:             retval = filemap_write_and_wait_range(mapping, pos,
01370:                     pos + iov_length(iov, nr_segs) - 1);
01371:             if (!retval) {
01372:                 retval = mapping->a_ops->direct_IO(READ, iocb,
01373:                         iov, pos, nr_segs);
01374:             }
```

```
01375:                 if (retval > 0) {
01376:                     *ppos = pos + retval;
01377:                     count -= retval;
01378:                 }
01379:
01380:                 /*
01381:                  * Btrfs can have a short DIO read if we encounter
01382:                  * compressed extents, so if there was an error, or if
01383:                  * we've already read everything we wanted to, or if
01384:                  * there was a short read because we hit EOF, go ahead
01385:                  * and return.  Otherwise fallthrough to buffered io for
01386:                  * the rest of the read.
01387:                  */
01388:                 if (retval < 0 || !count || *ppos >= size) {
01389:                     file_accessed(filp);
01390:                     goto ↓out;
01391:                 }
01392:             } « end if pos<size »
01393:         } « end if filp- >f_flags&O_DIRECT »
01394:
```

当执行direct IO读操作时，则进入直读模式，也就是跳过页缓存（1358～1393行）。不过这里我们暂时不关注direct IO，而是关注最常用的情形：异步读取基于页面缓存的文件。也就是会执行1395～1427行之间的代码。

```
01395:     count = retval;
01396:     for (seg = 0; seg < nr_segs; seg++) {
01397:         read_descriptor_t desc;
01398:         loff_t offset = 0;
01399:
01400:         /*
01401:          * If we did a short DIO read we need to skip the section of the
01402:          * iov that we've already read data into.
01403:          */
01404:         if (count) {
01405:             if (count > iov[seg].iov_len) {
01406:                 count -= iov[seg].iov_len;
01407:                 continue;
01408:             }
01409:             offset = count;
01410:             count = 0;
01411:         }
01412:
01413:         desc.written = 0;
01414:         desc.arg.buf = iov[seg].iov_base + offset;
01415:         desc.count = iov[seg].iov_len - offset;
01416:         if (desc.count == 0)
01417:             continue;
01418:         desc.error = 0;
01419:         do_generic_file_read(filp, ppos, &desc, file_read_actor);
01420:         retval += desc.written;
01421:         if (desc.error) {
01422:             retval = retval ?: desc.error;
01423:             break;
01424:         }
01425:         if (desc.count > 0)
01426:             break;
01427:     } « end for seg=0;seg<nr_segs;seg++ »
01428: out:
01429:     return retval;
01430: } « end generic_file_aio_read »
```

```
01431: EXPORT_SYMBOL(generic_file_aio_read);
01432:
```

1397、1413～1418行是定义一个read_descriptor_t类型的临时变量desc，该临时变量保存将要进行读操作的状态，该状态和用户空间缓冲区有关。这里再提出iov[seg].iov_base的值，就是sys_read(unsigned int fd, char __user * buf, size_t count)中的第2个参数的值；iov[seg].iov_len值是要读取的字节数count。

初始化临时变量desc后，就调用函数do_generic_file_read（），传递的参数包括文件对象指针filp，文件偏移量指针ppos，desc地址和函数file_read_actor（）地址。

generic_file_aio_read（）函数返回值是拷贝到用户缓冲区的字节数，这个值就是read_descriptor_t结构体中written的值。

1. read_descriptor_t数据结构

read_descriptor_t数据结构定义在文件include/linux/fs.h中。

```
00578: /*
00579: * "descriptor" for what we're up to with a read.
00580: * This allows us to use the same read code yet
00581: * have multiple different users of the data that
00582: * we read from a file.
00583: *
00584: * The simplest case just copies the data to user
00585: * mode.
00586: */
00587: typedef struct {
00588:     size_t written;
00589:     size_t count;
00590:     union {
00591:         char __user *buf;
00592:         void *data;
00593:     } arg;
00594:     int error;
00595: } read_descriptor_t;
00596:
```

各成员变量含义如下：

written：已拷贝到用户缓冲区的字节数；

count：待传输的字节数；

arg.buf：用户缓冲区的当前位置；

error：读操作的错误码（0表示没有错误）。

2.5 do_generic_file_read（）

do_generic_file_read（）函数的实现在mm/filemap.c文件中，主要功能是从磁盘上读取请求的页面，然后将数据拷贝到用户空间缓冲区中。注释也作了说明，该函数是通用文件读例程，真正执行读操作，是通过mapping->a_ops->readpage（）来完成。

这个函数非常长，我们分段阅读。

```
01025: /**
01026: * do_generic_file_read - generic file read routine
01027: * @filp:the file to read
01028: * @ppos:    current file position
01029: * @desc:    read_descriptor
01030: * @actor:   read method
01031: *
01032: * This is a generic file read routine, and uses the
01033: * mapping->a_ops->readpage() function for the actual low-level stuff.
01034: *
01035: * This is really ugly. But the goto's actually try to clarify some
01036: * of the logic when it comes to error handling etc.
01037: */
01038: static void do_generic_file_read(struct file *filp, loff_t *ppos,
01039:         read_descriptor_t *desc, read_actor_t actor)
01040: {
01041:     struct address_space *mapping = filp->f_mapping;
01042:     struct inode *inode = mapping->host;
01043:     struct file_ra_state *ra = &filp->f_ra;
01044:     pgoff_t index;
01045:     pgoff_t last_index;
01046:     pgoff_t prev_index;
01047:     unsigned long offset;      /* offset into pagecache page */
01048:     unsigned int prev_offset;
01049:     int error;
01050:
01051:     index = *ppos >> PAGE_CACHE_SHIFT;
01052:     prev_index = ra->prev_pos >> PAGE_CACHE_SHIFT;
01053:     prev_offset = ra->prev_pos & (PAGE_CACHE_SIZE-1);
01054:     last_index = (*ppos + desc->count + PAGE_CACHE_SIZE-1) >>
                                                    PAGE_CACHE_SHIFT;
01055:     offset = *ppos & ~PAGE_CACHE_MASK;
01056:
```

. 获取文件address_space对象：mapping = filep->f_mapping（1041行）。

通过1042行，获取address_space数据结构的拥有者，也就是inode对象，它的地址保存在address_space对象中的host成员变量中；inode对象拥有用来存储读取数据的页面。

获取预读状态ra：ra = &filp->f_ra，记录预读状态（1043行）。

内核从磁盘上读取文件数据，实际上是以页面大小（4096字节）为单位的；即使用户进程只读1字节，内核每次仍然会至少读取4096字节（若文件本身小于4K怎么办？）

宏PAGE_CACHE_SHIFT的值为12，PAGE_CACHE_SIZE的大小就是一个页面大小（4096字节），PAGE_CACHE_MASK的值为~(PAGE_SIZE-1)。

注1：这里只考虑页面大小为4K，不考虑页面大小为4M的情形。

将文件理解成分为多个页面；由前面的解释，我们就不难理解，1051～1054行计算将要读取的数据起始位置index和结束位置last_index所在文件中的页偏移。要读取数据第1字节不一定是和页面对齐的，所以1055行用临时变量offset记录读取的第一个字节在文件页面内的偏移量。

接下来开始一个for(; ;)循环处理这些页。

```
01057:      for (;;) {
01058:          struct page *page;
01059:          pgoff_t end_index;
01060:          loff_t isize;
01061:          unsigned long nr, ret;
01062:
01063:          cond_resched();
```

1063行检查当前进程TIF_NEED_RESCHED标志，确定是否需要调度；若设置了该标志，则调度schedule（）进行进程调度。

```
01064: find_page:
01065:          page = find_get_page(mapping, index);
01066:          if (!page) {
01067:              page_cache_sync_readahead(mapping,
01068:                      ra, filp,
01069:                      index, last_index - index);
01070:              page = find_get_page(mapping, index);
01071:              if (unlikely(page == NULL))
01072:                  goto ↓no_cached_page;
01073:          }
```

用户读取文件数据，使用之后，内核不会立即将数据丢弃；而是将数据缓存在内核里。进程再次请求该数据时，就会直接返回给用户，不必再从磁盘上读取；这样可以大大提高效率。

find_get_page（）就是查找请求的数据是否已经在内核缓冲区中。如果返回值为NULL，则说明此页不在高速缓存中，那么它将执行以下步骤：

（1）page_cache_sync_readahead（），这个函数会在cache未命中时被调用，它将提交预读请求，修改ra参数，当然也会相应的去预读一些页。

（2）再次去高速缓存中查找（上一步的预取可能已经将其读到cache中），若果没有找到，那么说明所请求的数据确实不在cache中，就跳到no_cached_page（1070～1072行）。

```
01074:          if (PageReadahead(page)) {
01075:              page_cache_async_readahead(mapping,
01076:                      ra, filp, page,
01077:                      index, last_index - index);
01078:          }
01079:          if (!PageUptodate(page)) {
01080:              if (inode->i_blkbits == PAGE_CACHE_SHIFT ||
01081:                      !mapping->a_ops->is_partially_uptodate)
```

```
01082:                goto ↓page_not_up_to_date;
01083:            if (!trylock_page(page))
01084:                goto ↓page_not_up_to_date;
01085:            /* Did it get truncated before we got the lock? */
01086:            if (!page->mapping)
01087:                goto ↓page_not_up_to_date_locked;
01088:            if (!mapping->a_ops->is_partially_uptodate(page,
01089:                                    desc, offset))
01090:                goto ↓page_not_up_to_date_locked;
01091:            unlock_page(page);
01092:        }
```

执行到1074行，说明所请求数据已在高速缓存中，那么此时判断它是否为readahead页（1074行），如果是，则发起异步预读请求page_cache_async_readahead（），这个函数在page有PG_readahead字段时才被调用，意味着已经用光了足够的readahead window中的page，我们需要添加更多的page到预读窗口。

接着判断此页的PG_uptodate位，如果被置位，则说明页中所存数据是最新的，因此无需从磁盘读数据，跳到1093行page_ok处。

mapping->a_ops->is_partially_uptodate（）检查页面中的buffer是否都处于update状态。当文件或者设备映射到内存中时，它们的inode结构就会和address_space相关联。当页面属于一个文件时，page->mapping就会指向这个地址空间。如果这个页面是匿名的且映射开启，则address_space就是swapper_space，swapper_space是管理交换地址空间的。

若页面不是最新的，说明页中的数据无效（1080～1091行）：

（1）若mapping->a_ops->is_partially_uptodate是NULL，或者trylock_page(page)是NULL，那么跳到page_not_up_to_date（1080～1084行）。

（2）若文件页面还没建立映射或者页面中的buffer不都是update状态，则跳转到page_not_up_to_date_locked（1086～1090行）。

（3）前面没有跳转的话，说明页面是PG_uptodate状态，通过unlock_page（）解锁页面。

```
01093: page_ok:
01094:        /*
01095:         * i_size must be checked after we know the page is Uptodate.
01096:         *
01097:         * Checking i_size after the check allows us to calculate
01098:         * the correct value for "nr", which means the zero-filled
01099:         * part of the page is not copied back to userspace (unless
01100:         * another truncate extends the file - this is desired though).
01101:         */
01102:
01103:        isize = i_size_read(inode);
01104:        end_index = (isize - 1) >> PAGE_CACHE_SHIFT;
```

```
01105:          if (unlikely(! isize || index > end_index)) {
01106:              page_cache_release(page);
01107:              goto ↓out;
01108:          }
01109:
01110:      /* nr is the maximum number of bytes to copy from this page */
01111:      nr = PAGE_CACHE_SIZE;
01112:          if (index == end_index) {
01113:              nr = ((isize - 1) & ~PAGE_CACHE_MASK)    + 1;
01114:              if (nr <= offset) {
01115:                  page_cache_release(page);
01116:                  goto ↓out;
01117:              }
01118:          }
01119:          nr = nr - offset;
01120:
01121:          /* If users can be writing to this page using arbitrary
01122:           * virtual addresses, take care about potential aliasing
01123:           * before reading the page on the kernel side.
01124:           */
01125:          if (mapping_writably_mapped(mapping))
01126:              flush_dcache_page(page);
01127:
01128:          /*
01129:           * When a sequential read accesses a page several times,
01130:           * only mark it as accessed the first time.
01131:           */
01132:          if (prev_index != index || offset != prev_offset)
01133:              mark_page_accessed(page);
01134:          prev_index = index;
01135:
01136:          /*
01137:           * Ok, we have the page, and it's up-to-date, so
01138:           * now we can copy it to user space...
01139:           *
01140:           * The actor routine returns how many bytes were actually used..
01141:           * NOTE! This may not be the same as how much of a user buffer
01142:           * we filled up (we may be padding etc), so we can only update
01143:           * "pos" here (the actor routine has to update the user buffer
01144:           * pointers and the remaining count).
01145:           */
01146:          ret = actor(desc, page, offset, nr);
01147:          offset += ret;
01148:          index += offset >> PAGE_CACHE_SHIFT;
01149:          offset &= ~PAGE_CACHE_MASK;
01150:          prev_offset = offset;
01151:
01152:          page_cache_release(page);
01153:          if (ret == nr && desc->count)
01154:              continue;
01155:          goto ↓out;
01156:
```

1093～1155行之间的代码总是会被执行，即使页面不在Cache中，从磁盘读取数据，之后也会跳转page_ok处（1216行）。

首先检查文件大小，也就是index * 4096是否超过i_size，如果超过，则减少page的计数器，并且跳转到out处（1103～1108行）。设置nr大小，nr是从此页可读取的最大字节数，就是向用户缓冲区拷贝的字节数。之后就跳转到page_ok标号处。通常的页是4KB，但是对于文件的最后一页，可能不足4KB，所以要进行调整（1111～1119行）。

如果用户使用不正常的虚拟地址写这个页，要flush_dcache_page（），将dcache相应的page里的数据写到memory里去，以保证dcache内的数据与memory内的数据的一致性。但在x86架构中，flush_dcache_page（）的实现为空，不做任何操作（1125～1126行）。

1132～1133行调用make_page_accessed（）设置页面的PG_active和PG_referenced标志，表示页面正在被使用且不能被调换（swap）出去。若同样的页面被do_generic_file_read（）多次读，这一步仅在第一次读时执行（即offset为0）。

1146行就是将已读取到的数据拷贝到用户空间的缓冲区中，调用的是file_read_actor（）函数，稍后我们再仔细分析这个函数。数据拷贝到用户空间的缓冲区后，更新局部变量offset和index的值（1148～1149行），递减页面计数（1152行）。若read_descriptor_t描述符中的count值不为0，说明还有其他文件数据需要读，for循环不退出，重新从1058行开始执行，进行下一轮读。

```
01157: page_not_up_to_date:
01158:         /* Get exclusive access to the page ... */
01159:         error = lock_page_killable(page);
01160:         if (unlikely(error))
01161:             goto ↓readpage_error;
01162:
01163: page_not_up_to_date_locked:
01164:         /* Did it get truncated before we got the lock? */
01165:         if (!page->mapping) {
01166:             unlock_page(page);
01167:             page_cache_release(page);
01168:             continue;
01169:         }
01170:
01171:         /* Did somebody else fill it already? */
01172:         if (PageUptodate(page)) {
01173:             unlock_page(page);
01174:             goto ↑page_ok;
01175:         }
01176:
```

page_not_up_to_date：

lock_page_killable（），给page加锁，这个过程可以被中断的，如果加锁失败，则跳到readpage_error，否则跳转到page_not_up_to_date_locked。

page_not_up_to_date_locked：

（1）如果在获取锁之前就此页已经被truncated，释放锁，减少页的计数，结束本次循环，进入下一次循环。

（2）否则，它没有被truncated，如果它已经被填充（已从磁盘上读取），那么解锁，并且跳转到page_ok。

```
01177: readpage:
01178:     /*
01179:     * A previous I/O error may have been due to temporary
01180:          * failures, eg. multipath errors.
01181:          * PG_error will be set again if readpage fails.
01182:          */
01183:         ClearPageError(page);
01184:         /* Start the actual read. The read will unlock the page. */
01185:         error = mapping->a_ops->readpage(filp, page);
01186:
01187:         if (unlikely(error)) {
01188:             if (error == AOP_TRUNCATED_PAGE) {
01189:                 page_cache_release(page);
01190:                 goto ↑find_page;
01191:             }
01192:             goto ↓readpage_error;
01193:         }
01194:
01195:         if (!PageUptodate(page)) {
01196:             error = lock_page_killable(page);
01197:             if (unlikely(error))
01198:                 goto ↓readpage_error;
01199:             if (!PageUptodate(page)) {
01200:                 if (page->mapping == NULL) {
01201:                     /*
01202:                      * invalidate_inode_pages got it
01203:                      */
01204:                     unlock_page(page);
01205:                     page_cache_release(page);
01206:                     goto ↑find_page;
01207:                 }
01208:                 unlock_page(page);
01209:                 shrink_readahead_size_eio(filp, ra);
01210:                 error = -EIO;
01211:                 goto ↓readpage_error;
01212:             }
01213:             unlock_page(page);
01214:         } « end if ! PageUptodate(page) »
01215:
01216:         goto ↑page_ok;
01217:
01218: readpage_error:
01219:         /* UHHUH! A synchronous read error occurred. Report it */
01220:         desc->error = error;
01221:         page_cache_release(page);
01222:         goto ↓out;
01223:
01224: no_cached_page:
01225:         /*
01226:          * Ok, it wasn't cached, so we need to create a new
01227:          * page..
01228:          */
01229:         page = page_cache_alloc_cold(mapping);
01230:         if (!page) {
01231:             desc->error = -ENOMEM;
01232:             goto ↓out;
01233:         }
01234:         error = add_to_page_cache_lru(page, mapping,
01235:                         index, GFP_KERNEL);
01236:         if (error) {
01237:             page_cache_release(page);
01238:             if (error == -EEXIST)
01239:                 goto ↑find_page;
01240:             desc->error = error;
```

```
01241:                goto ↓out;
01242:            }
01243:            goto ↑readpage;
01244:        } « end for ;; »
01245:
01246: out:
01247:        ra->prev_pos = prev_index;
01248:        ra->prev_pos <<= PAGE_CACHE_SHIFT;
01249:        ra->prev_pos |= prev_offset;
01250:
01251:        *ppos = ((loff_t)index << PAGE_CACHE_SHIFT)    + offset;
01252:        file_accessed(filp);
01253: } « end do_generic_file_read »
01254:
```

1177行～1216行就是真正执行从磁盘读数据的过程。其实真正执行从磁盘读数据的操作仅在1185行，就是调用文件的address_space对象readpage（）方法，对于ext4文件系统来说，readpage方法实现都是函数ext4_readpage（），这个函数仅是对mpage_readpage（）简单封装。从函数名就可以看出，该函数一次只读一个页面，显然对于内核来说这种情形是很少有的，大部分是一次读取多个页面，也就是预读（一次读多个页面）。对于do_generic_file_read（）来说，执行最多的路径是执行1067行的page_cache_sync_readahead（）和1075行page_cache_async_readahead（）。

mapping->a_ops->readpage（），如果读取失败，error返回值不为0，再进一步判断error是AOP_TRUNCATED_PAGE时，则减少计数，跳到find_page，如果不是此error，那么跳到readpage_error（1187～1193行）。

若页面没有被设置PG_uptodate标志位，即页中的数据不是最新的，那么给页加锁，如果加锁失败，则跳到readpage_error。加锁成功则再次判断他的PG_uptodate位，如果仍然没有被置位，再判断page的mapping是不是为空，若为空，则解锁并且减少page计数，然后跳到find_page。如果mapping不为空，则调整ra，并跳到readpage_error（1195～1214行）。

经常以上判断检查后，说明页中的数据是最新的，那么跳到page_ok（1216行）。

下面是几个标号分支处理说明：

readpage_error（1218～1222行）：

到此步骤，说明同步read出错，报告这个错误，并且跳出循环，即跳到out。

no_cached_page（1224～1243行）：

分配一个新页：page_cache_alloc_cold()，然后添加到cache的lru上：add_to_page_cache_lru()，跳到readpage。

这里，我们就可以回答本文概述中的一个问题：“内核是在哪个地方分配空间来存储将要读取的数据？”。答案是，在do_generic_mapping_read（）函数中的1229～1233行，此时内存缓存中没有要请求的数据。这里提醒一下，read（）的一个参数buf，其缓冲区是在用户空间的，大小与页面大小无关。而1229～1233行分配的空间是在内核中，且大小是以页面大小为单位。

out（1246～1252行）：

所请求或者可提供的字节已经被读取，更新filp->f_ra提前读数据结构（1247～1249行）。然后更新*ppos的值（1251行），也就是更新当前文件读写位置；编程中经常使用read（）、write（）的话，不难理解ppos的值。最后调用file_accessed（）来更新文件inode节点中i_atime的时间为当前时间，且将inode标记为脏。至此，do_generic_file_read（）函数执行完毕。那么有个问题，现在用户请求的数据是否已经拷贝到用户缓冲区buf中？

2.5.1 address_space->readpage（）方法

源码均在fs/ext4/inode.c。

```
04103: static const struct address_space_operations ext4_ordered_aops   = {
04104:     .readpage        = ext4_readpage,
04105:     .readpages       = ext4_readpages,
04106:     .writepage       = ext4_writepage,
04107:     .sync_page       = block_sync_page,
04108:     .write_begin          = ext4_write_begin,
04109:     .write_end       = ext4_ordered_write_end,
04110:     .bmap            = ext4_bmap,
04111:     .invalidatepage       = ext4_invalidatepage,
04112:     .releasepage          = ext4_releasepage,
04113:     .direct_IO       = ext4_direct_IO,
04114:     .migratepage          = buffer_migrate_page,
04115:     .is_partially_uptodate     = block_is_partially_uptodate,
04116:     .error_remove_page     = generic_error_remove_page,
04117: };
04118:
```

这里我们列出系统默认操作方式ext4_ordered_aops，readpage实现为ext4_readpage（），仅是对mpage_readpage（）函数作进一步封装。后面我们将详细分析mpage_readpage（）的实现。

ext4_get_block（）函数的主要功能：基于给定的inode，查找文件的逻辑块iblock，并将其与bh进行映射。如果没有查找到，且参数create为1，则执行块分配操作为文件申请新的block，然后再与bh建立映射。

```
03550: static int ext4_readpage(struct file *file, struct page *page)
03551: {
```

```
03552:      return mpage_readpage(page, ext4_get_block);
03553: }
03554:
```

2.5.2 file_read_actor（）

函数file_read_actor（）源码出mm/filemap.c。

该函数在do_generic_file_read（）1146行被调用，是用来将内核缓冲区中的数据拷贝到用户缓冲区中。

```
01255: int file_read_actor(read_descriptor_t *desc, struct page *page,
01256:             unsigned long offset, unsigned long size)
01257: {
01258:     char *kaddr;
01259:     unsigned long left, count = desc->count;
01260:
01261:     if (size > count)
01262:         size = count;
01263:
01264:     /*
01265:      * Faults on the destination of a read are common, so do it before
01266:      * taking the kmap.
01267:      */
01268:     if (!fault_in_pages_writeable(desc->arg.buf, size)) {
01269:         kaddr = kmap_atomic(page, KM_USER0);
01270:         left = __copy_to_user_inatomic(desc->arg.buf,
01271:                         kaddr + offset, size);
01272:         kunmap_atomic(kaddr, KM_USER0);
01273:         if (left == 0)
01274:             goto ↓success;
01275:     }
01276:
01277:     /* Do it the slow way */
01278:     kaddr = kmap(page);
01279:     left = __copy_to_user(desc->arg.buf, kaddr + offset, size);
01280:     kunmap(page);
01281:
01282:     if (left) {
01283:         size -= left;
01284:         desc->error = -EFAULT;
01285:     }
01286: success:
01287:     desc->count = count - size;
01288:     desc->written += size;
01289:     desc->arg.buf += size;
01290:     return size;
01291: } « end file_read_actor »
```

执行步骤如下：

（1）若空间处于高端内存，则调用kmap_atomic（）建立内核持久映射（详情见《Linux高端内存的管理》）；

（2）调用__copy_to_user_inatomic（）将数据拷贝到用户空间；注意此时进程可能会被阻塞，因为访问用户空间时发生页面错误；

（3）调用kunmap（）解除地址映射；

（4）更新count、written和arg.buf的值。count表示待读取的字节数，written是已拷贝到用户控件的字节数，arg.buf是用户空间的缓冲区地址。

2.6 do_mpage_readpage（）

前面提到Linux内核读文件通常采取预读机制，不是每次只读一个页面。在do_generic_file_read（）函数中，执行最多的是page_cache_sync_readahead（）和page_cache_async_readahead（），很少直接调用mapping->a_ops->readpage（）。我们会在另外的章节中单独介绍Linux内核预读和Cache机制，在预读机制中，会调用mapping->a_ops->readpages（）一次读取多个页面。

这里我们先分析执行较少时的情形，调用a_ops->readpage（）一次读取一个页面。一次读取多个页面，在Linux内核预读机制章节分析。

do_mpage_readpage（）也在文件fs/mpage.c中，试图读取文件中的一个page大小的数据。理想的情况是这个page大小的数据在磁盘上都是连续的，然后只需要提交一个bio请求就可以获取所有的数据。

do_mpage_readpage（）大部分工作在检查page上所有的物理块是否连续，检查的方法就是调用文件系统提供的get_block（）函数；如果不连续，就调用block_read_full_page （）以buffer 缓冲区的形式来逐个块获取数据。

do_mpage_readpage分析page对应的物理块映射关系进行不同处理：

（1）调用get_block（）函数检查page中所有的物理块是否连续；

（2）若磁盘上物理块不连续，就调用mpage_bio_submit（）读取。

（3）若磁盘上物理块连续。如果没有bio，则新分配一个bio。

（4）物理块连续，但是和上一个page的物理block不连续。把上一次的bio给提交了，然后重新分配bio。

（5）所请求的页面有hole，也有数据，就block_read_full_page（）逐块读取。

（6）所请求的页面全是hole，将page清零，不用读了，直接返回。

```
00157: /*
00158:  * This is the worker routine which does all the work of mapping the disk
00159:  * blocks and constructs largest possible bios, submits them for IO if the
00160:  * blocks are not contiguous on the disk.
00161:  *
00162:  * We pass a buffer_head back and forth and use its buffer_mapped() flag to
00163:  * represent the validity of its disk mapping and to decide when to do the next
00164:  * get_block() call.
```

```
00165: */
00166: static struct bio *
00167: do_mpage_readpage(struct bio *bio, struct page *page,
unsigned nr_pages,
00168:         sector_t *last_block_in_bio, struct buffer_head *map_bh,
00169:         unsigned long *first_logical_block, get_block_t get_block)
00170: {
00171:     struct inode *inode = page->mapping->host;
00172:     const unsigned blkbits = inode->i_blkbits;
00173:     const unsigned blocks_per_page = PAGE_CACHE_SIZE >> blkbits;
00174:     const unsigned blocksize = 1 << blkbits;
00175:     sector_t block_in_file;
00176:     sector_t last_block;
00177:     sector_t last_block_in_file;
00178:     sector_t blocks[MAX_BUF_PER_PAGE];
00179:     unsigned page_block;
00180:     unsigned first_hole = blocks_per_page;
00181:     struct block_device *bdev = NULL;
00182:     int length;
00183:     int fully_mapped = 1;
00184:     unsigned nblocks;
00185:     unsigned relative_block;
00186:
00187:     if (page_has_buffers(page))
00188:         goto ↓confused;
00189:
00190:     block_in_file = (sector_t)page->index << (PAGE_CACHE_SHIFT - blkbits);
00191:     last_block = block_in_file + nr_pages * blocks_per_page;
00192:     last_block_in_file = (i_size_read(inode) + blocksize - 1) >> blkbits;
00193:     if (last_block > last_block_in_file)
00194:         last_block = last_block_in_file;
00195:     page_block = 0;
00196:
```

首先检查页面PG_private标志（187行）；若该标志已设置，则表明：（1）它是缓冲页面（即该页面与一个buffer head链表关联），且已经从磁盘上读取到内存中；（2）页面中的块在磁盘上不是相邻的；这样的话，就跳转到318行，一次读取页面的一个块。

获取块大小blocksize（保存在page->mapping->host->inode->i_blkbits），并且计算访问这个页面所需的两个值：页面中的块数（173行）和页面的第一个块（即页内的第一个块相对于文件起始位置的索引）。

block_in_file：page中的第一个block number。

last_block：page中最后一个block 大小。

last_block_in_file：文件最后一个block大小。

last_block最终值为本次page读操作的最后一个block大小。

```
00197:     /*
00198:      * Map blocks using the result from the previous get_blocks call first.
00199:      */
00200:     nblocks = map_bh->b_size >> blkbits;
00201:     if (buffer_mapped(map_bh) && block_in_file > *first_logical_block &&
00202:             block_in_file < (*first_logical_block + nblocks)) {
00203:         unsigned map_offset = block_in_file - *first_logical_block;
```

```
00204:          unsigned last = nblocks - map_offset;
00205:
00206:          for (relative_block = 0; ; relative_block++) {
00207:              if (relative_block == last) {
00208:                  clear_buffer_mapped(map_bh);
00209:                  break;
00210:              }
00211:              if (page_block == blocks_per_page)
00212:                  break;
00213:              blocks[page_block] = map_bh->b_blocknr   + map_offset +
00214:                              relative_block;
00215:              page_block++;
00216:              block_in_file++;
00217:          }
00218:          bdev = map_bh->b_bdev;
00219:      }
00220:
```

通常情况下，map_bh仅是临时变量，不会执行203～218行。这里忽略。

```
00221:      /*
00222:       * Then do more get_blocks calls until we are done with this page.
00223:       */
00224:      map_bh->b_page = page;
00225:      while (page_block < blocks_per_page) {
00226:          map_bh->b_state = 0;
00227:          map_bh->b_size = 0;
00228:
00229:          if (block_in_file < last_block) {
00230:              map_bh->b_size = (last_block-block_in_file) << blkbits;
00231:              if (get_block(inode,   block_in_file, map_bh, 0))
00232:                  goto ↓confused;
00233:              *first_logical_block   = block_in_file;
00234:          }
00235:
00236:          if (!buffer_mapped(map_bh)) {
00237:              fully_mapped = 0;
00238:              if (first_hole == blocks_per_page)
00239:                  first_hole = page_block;
00240:              page_block++;
00241:              block_in_file++;
00242:              continue;
00243:          }
00244:
00245:          /* some filesystems will copy data into the page during
00246:           * the get_block call, in which case we don't want to
00247:           * read it again.  map_buffer_to_page copies the data
00248:           * we just collected from get_block into the page's buffers
00249:           * so readpage doesn't have to repeat the get_block call
00250:           */
00251:          if (buffer_uptodate(map_bh)) {
00252:              map_buffer_to_page(page, map_bh, page_block);
00253:              goto ↓confused;
00254:          }
00255:
00256:          if (first_hole != blocks_per_page)
00257:              goto ↓confused;         /* hole -> non-hole */
00258:
00259:  /* Contiguous blocks? */
00260:  if (page_block && blocks[page_block-1] != map_bh->b_blocknr-1)
00261:              goto ↓confused;
00262:          nblocks = map_bh->b_size >> blkbits;
00263:          for (relative_block = 0; ; relative_block++) {
00264:              if (relative_block == nblocks) {
00265:                  clear_buffer_mapped(map_bh);
00266:                  break;
```

```
00267:                } else if (page_block == blocks_per_page)
00268:                    break;
00269:                blocks[page_block] = map_bh->b_blocknr+relative_block;
00270:                page_block++;
00271:                block_in_file++;
00272:            }
00273:            bdev = map_bh->b_bdev;
00274:        } « end while page_block<blocks_per... »
00275:
```

224～274行：

（1）对于页内的每一个块，调用文件系统相关的get_block函数计算逻辑块数，就是相对于磁盘或分区起始位置的索引。

（2）页面里所有块的逻辑块数存放在一个局部数组blocks[]中。

（3）检查前面几步中可能出现的异常条件。例如，有些块在磁盘上不相邻，或有些块落入文件空洞中，或某个块缓冲区已经由get_block函数填充。然跳转到confused标号处。

```
00276:      if (first_hole != blocks_per_page) {
00277:          zero_user_segment(page, first_hole << blkbits, PAGE_CACHE_SIZE);
00278:          if (first_hole == 0) {
00279:              SetPageUptodate(page);
00280:              unlock_page(page);
00281:              goto ↓out;
00282:          }
00283:      } else if (fully_mapped) {
00284:          SetPageMappedToDisk(page);
00285:      }
00286:
```

如果发现文件中有空洞，就将整个page清零，因为文件洞的区域物理层不会真的去磁盘上读取，必须在这里主动清零，否则文件洞区域内容可能随机。该页面也可能是文件的最后一部分数据，于是页面中的部分块在磁盘上没有对应的数据。这样的话，将这些块中都清零。否则设置页面描述符的PG_mappeddisk标志（284行）。

```
00287:      /*
00288:       * This page will go to BIO.    Do we need to send this BIO off first?
00289:       */
00290:      if (bio && (*last_block_in_bio    != blocks[0] - 1))
00291:          bio = mpage_bio_submit(READ, bio);
00292:
00293: alloc_new:
00294:      if (bio == NULL) {
00295:          bio = mpage_alloc(bdev,    blocks[0] << (blkbits - 9),
00296:                  min_t(int, nr_pages, bio_get_nr_vecs(bdev)),
00297:                  GFP_KERNEL);
00298:          if (bio == NULL)
00299:              goto ↓confused;
00300:      }
00301:
```

调用mpage_alloc（）分配一个新bio描述符（295行），仅包含一个段（segment）；并将

成员变量bi_bdev的值初始化为块设备描述符地址，将成员变量bi_sector初始化为页面中第一个块的逻辑块号。这两个值在前面已经计算。

```
00302:     length = first_hole << blkbits;
00303:     if (bio_add_page(bio,   page, length, 0) < length) {
00304:         bio = mpage_bio_submit(READ, bio);
00305:         goto ↑alloc_new;
00306:     }
00307:
00308:     relative_block = block_in_file - *first_logical_block;
00309:     nblocks = map_bh->b_size >> blkbits;
00310:     if ((buffer_boundary(map_bh) && relative_block == nblocks) ||
00311:       (first_hole != blocks_per_page))
00312:         bio = mpage_bio_submit(READ, bio);
00313:     else
00314:         *last_block_in_bio    = blocks[blocks_per_page   - 1];
00315: out:
00316:     return bio;
00317:
00318: confused:
00319:     if (bio)
00320:         bio = mpage_bio_submit(READ, bio);
00321:     if (!PageUptodate(page))
00322:         block_read_full_page(page, get_block);
00323:         else
00324:         unlock_page(page);
00325:     goto ↑out;
00326: } « end do_mpage_readpage »
00327:
```

若物理块上连续，则尝试页面合并成新的bio。304行是当前page和上一个page的物理block不连续，把上一次的bio给提交，然后重新分配bio。

若当前page和上一个page，物理上连续，合并成新的bio后，返回更新后的bio，等待下一次操作（316行）。

318～325行：若函数跳转到这里，页面中包含的块在磁盘上是不相邻的。若页面是最新的（设置了PG_update标志），调用unlock_page（）来对页面解锁，否则就调用block_read_full_page（）来一次读取页面上的一个块。

mpage_bio_submit（）函数，首先设置bio->bi_end_io的值为mpage_end_io_read（），该函数是bio结束方法。当I/O数据传输结束时，它就立即被执行。假设没有I/O错误，该函数主要设置页面描述符中的PG_uptodate标志，然后调用unlock_page（）来解锁页面，并唤醒在事件上等待的所有进程；最后调用bio_put（）销毁bio描述符。在调用submit_bio（），设置数据传输方向到bi_rw标志，更新per-CPU类型变量page_states以记录读扇区数；然后调用generic_make_request（）。

2.7 mpage_bio_submit（）

```
00084: static struct bio *mpage_bio_submit(int  rw, struct bio *bio)
00085: {
00086:     bio->bi_end_io = mpage_end_io_read;
00087:     if (rw == WRITE)
00088:         bio->bi_end_io = mpage_end_io_write;
00089:     submit_bio(rw, bio);
00090:     return NULL;
00091: }
00092:
```

mpage_bio_submit（）的源码非常简单，就是设置bio的bi_end_io方法，然后调用submit_bio（）。源码实现在fs/mpage.c中。

至此，我们分析了read（）在虚拟文件系统和文件系统中的流程，分析到了mpage_bio_submit（），该函数最后调用submit_bio（），此时进入了通用块设备层（Generic Block Layer），接下来会进入I/O调度层、驱动层，最后数据返回。

我们将会在单独的章节介绍通用块设备层、I/O调度层和驱动层。

3 读数据完成返回过程

3.1 读进程的阻塞和继续执行过程

当内核将读请求发到磁盘控制器驱动的请求队列中，显然将请求放入队列中后，用户进程请求的数据并没有立即读到内核空间和用户缓冲区中。将读请求放入队列中后，sys_read（）执行结束，并返回到用户空间？用户进程读数据时，是否会有阻塞过程？

将请求放入队列后，read（）系统调用并没有立即从内核空间返回到用户空间。因为请求的数据还没有读到内存中，就不能返回到用户空间，一旦返回到用户空间，用户进程就可以随时访问，可此时用户缓冲区中还没有请求的数据呢，又如何能访问？

将请求放到块设备（磁盘）的请求队列上后，用户进程就会阻塞，直到所请求的数据就绪为止。本小节就主要分析在sys_read（）执行过程中，在哪里阻塞，又是何时被唤醒继续执行的过程。

回顾前面分析的函数do_generic_file_read（）。

```
01038: static void  do_generic_file_read(struct file *filp, loff_t *ppos,
01039:         read_descriptor_t *desc, read_actor_t  actor)
01040: {
01041:     struct address_space *mapping = filp->f_mapping;
01042:     struct inode *inode = mapping->host;
01043:     struct file_ra_state *ra = &filp->f_ra;
```

```
... ...
01063:          cond_resched();
01064: find_page:
01065:          page = find_get_page(mapping, index);
01066:          if (!page) {
01067:              page_cache_sync_readahead(mapping,
01068:                      ra, filp,
01069:                      index, last_index - index);
01070:              page = find_get_page(mapping, index);
01071:              if (unlikely(page == NULL))
01072:                  goto ↓no_cached_page;
01073:          }
01074:          if (PageReadahead(page)) {
01075:              page_cache_async_readahead(mapping,
01076:                      ra, filp, page,
01077:                      index, last_index - index);
01078:                      }
01079:          if (!PageUptodate(page)) {
01080:              if (inode->i_blkbits == PAGE_CACHE_SHIFT ||
01081:                      !mapping->a_ops->is_partially_uptodate)
01082:                  goto ↓page_not_up_to_date;
01083:              if (!trylock_page(page))
01084:                  goto ↓page_not_up_to_date;
01085:              /* Did it get truncated before we got the lock? */
01086:              if (!page->mapping)
01087:                  goto ↓page_not_up_to_date_locked;
01088:              if (!mapping->a_ops->is_partially_uptodate(page,
01089:                                  desc, offset))
01090:                  goto ↓page_not_up_to_date_locked;
01091:              unlock_page(page);
01092:           }
... ...
```

读文件数据大多数情况下都是通过预读的，也就是执行page_cache_sync_readahead（）或page_cache_async_readahead（）。该函数中完成了将数据请求放到块设备队列上去，然后该函数就执行结束了，但请求的数据并不会那么快已读到内存中。

执行预读后，通过find_get_page（）再次查找所请求的页面是否在缓冲区中，经过预读后，页面已经在缓冲区中。进而执行1079行，因为请求的数据还未就绪，那么页面得PG_update标志就没有被设置，就会跳转到page_not_up_to_date标号处。

```
01157: page_not_up_to_date:
01158:          /* Get exclusive access to the page ... */
01159:          error = lock_page_killable(page);
01160:          if (unlikely(error))
01161:              goto ↓readpage_error;
01162:
01163: page_not_up_to_date_locked:
01164:          /* Did it get truncated before we got the lock? */
01165:          if (!page->mapping) {
01166:              unlock_page(page);
01167:              page_cache_release(page);
01168:              continue;
01169:          }
01170:
01171:          /* Did somebody else fill it already? */
01172:          if (PageUptodate(page)) {
```

```
01173:              unlock_page(page);
01174:              goto ↑page_ok;
01175:          }
01176:
```

… …

进程的阻塞正是在lock_page_killable（）inline函数中，当请求的数据读到内存中后，lock_page_killable（）才会执行完毕，该函数定义在include/linux/pagemap.h中。

```
00331: /*
00332:  * lock_page_killable is like lock_page but can be interrupted by fatal
00333:  * signals.  It returns 0 if it locked the page and -EINTR if it was
00334:  * killed while waiting.
00335:  */
00336: static inline int lock_page_killable(struct page *page)
00337: {
00338:     might_sleep();
00339:     if (!trylock_page(page))
00340:         return __lock_page_killable(page);
00341:     return 0;
00342: }
00343:
```

__lock_page_killable（）的实现在mm/filemap.c中。

```
00633: int __lock_page_killable(struct page *page)
00634: {
00635:     DEFINE_WAIT_BIT(wait,   &page->flags, PG_locked);
00636:
00637:     return __wait_on_bit_lock(page_waitqueue(page), &wait,
00638:                 sync_page_killable, TASK_KILLABLE);
00639: }
00640: EXPORT_SYMBOL_GPL(__lock_page_killable);
```

当数据还未就绪时，调用__wait_on_bit_lock（）等待解除PG_locked和设置PG_update。

后面我们会分析内核在何时、何处解除page锁标志和设置PG_update标志。

Call Trace:

<ffffffff8014b005>{__lock_page+125}
<ffffffff8014ab2a>{page_wake_function+0}
<ffffffff8014ab2a>{page_wake_function+0}
<ffffffff8014b0ee>{find_get_page+65}
<ffffffff8014b5e2>{do_generic_mapping_read+500}
<ffffffff8014b7f6>{file_read_actor+0}
<ffffffff8014d405>{__generic_file_aio_read+382}
<ffffffff8014d5fa>{generic_file_aio_read+48}
<ffffffff80166fad>{do_sync_read+178}
<ffffffff80127e04>{autoremove_wake_function+0}
<ffffffff801670a8>{vfs_read+207}
<ffffffff80167304>{sys_read+69}
<ffffffff80110202>{system_call+126}

图2 __lock_page（）函数调用栈

3.2 读数据返回过程

这里我们就不详细分析磁盘控制器驱动是如何从请求队列中摘掉请求，然后读取数据的。现在从数据完全都到内存缓冲区中开始分析。

当数据读到内存中后，磁盘控制器就会向CPU发一个中断，然后就会执行中断处理程序。为了简化我们的分析，下图给出了读数据完成后，中断处理的函数栈。

```
#############Wake up task name:md5sum, pid:3832, in_iowait:1, state:130, flags:402000 ###########
Pid: 0, comm: swapper Tainted: G                ---------- HT 2.6.32279.debug #43
Call Trace:
 <IRQ>  [<ffffffff81060167>] ? try_to_wake_up+0x2f7/0x480
 [<ffffffff810137f3>] ? native_sched_clock+0x13/0x80
 [<ffffffff81060302>] ? default_wake_function+0x12/0x20
 [<ffffffff81092186>] ? autoremove_wake_function+0x16/0x40
 [<ffffffff810921eb>] ? wake_bit_function+0x3b/0x50
 [<ffffffff8104e309>] ? __wake_up_common+0x59/0x90
 [<ffffffff810533e8>] ? __wake_up+0x48/0x70
 [<ffffffff81092121>] ? __wake_up_bit+0x31/0x40
 [<ffffffff81114277>] ? unlock_page+0x27/0x30
 [<ffffffff811b6dc9>] ? mpage_end_io_read+0x39/0x90
 [<ffffffff811b138d>] ? bio_endio+0x1d/0x40
 [<ffffffff81254d7b>] ? req_bio_endio+0x9b/0xe0
 [<ffffffff81256917>] ? blk_update_request+0x107/0x490
 [<ffffffff81256cc7>] ? blk_update_bidi_request+0x27/0xa0
 [<ffffffff8125814f>] ? blk_end_bidi_request+0x2f/0x80
 [<ffffffff812581f0>] ? blk_end_request+0x10/0x20
 [<ffffffff8136c22f>] ? scsi_io_completion+0xaf/0x6c0
 [<ffffffff81363332>] ? scsi_finish_command+0xc2/0x130
 [<ffffffff8136c9a5>] ? scsi_softirq_done+0x145/0x170
 [<ffffffff8125d825>] ? blk_done_softirq+0x85/0xa0
 [<ffffffff81073f61>] ? __do_softirq+0xc1/0x1e0
 [<ffffffff810db936>] ? handle_IRQ_event+0xf6/0x170
 [<ffffffff8100c24c>] ? call_softirq+0x1c/0x30
 [<ffffffff8100de85>] ? do_softirq+0x65/0xa0
 [<ffffffff81073d45>] ? irq_exit+0x85/0x90
 [<ffffffff81505d15>] ? do_IRQ+0x75/0xf0
 [<ffffffff8100ba53>] ? ret_from_intr+0x0/0x11
 <EOI>  [<ffffffff812f7a0a>] ? acpi_idle_enter_c1+0xa3/0xc1
 [<ffffffff812f79e9>] ? acpi_idle_enter_c1+0x82/0xc1
 [<ffffffff81407847>] ? cpuidle_idle_call+0xa7/0x140
 [<ffffffff81009e06>] ? cpu_idle+0xb6/0x110
 [<ffffffff814f6eff>] ? start_secondary+0x22a/0x26d
```

图3 读数据返回内核栈

对于scsi_end_io_completion（）、__end_that_request_first（）函数，可以参考Linux块设备驱动章节。这里我们就关系mpage_end_io_read（）函数，这是让lock_page_killable（）解锁的重要之处，源码在fs/mpage.c中。

```
00030: /*
00031:  * I/O completion handler for multipage BIOs.
00032:  *
00033:  * The mpage code never puts partial pages into a BIO (except for end-of-file).
00034:  * If a page does not map to a contiguous run of blocks then it simply falls
00035:  * back to block_read_full_page().
00036:  *
00037:  * Why is this?  If a page's completion depends on a number of different BIOs
00038:  * which can complete in any order (or at the same time) then determining the
00039:  * status of that page is hard.  See end_buffer_async_read() for the details.
00040:  * There is no point in duplicating all that complexity.
00041:  */
00042: static void mpage_end_io_read(struct bio *bio, int err)
00043: {
00044:     const int uptodate = test_bit(BIO_UPTODATE, &bio->bi_flags);
00045:     struct bio_vec *bvec = bio->bi_io_vec + bio->bi_vcnt - 1;
00046:
00047:     do {
00048:         struct page *page = bvec->bv_page;
00049:
00050:         if (--bvec >= bio->bi_io_vec)
00051:             prefetchw(&bvec->bv_page->flags);
00052:
00053:         if (uptodate) {
00054:             SetPageUptodate(page);
00055:         } else {
00056:             ClearPageUptodate(page);
00057:             SetPageError(page);
00058:         }
00059:         unlock_page(page);
00060:     } while (bvec >= bio->bi_io_vec);
00061:     bio_put(bio);
00062: } « end mpage_end_io_read »
00063:
```

函数先检查bio标志，是否设置了BIO_UPTODATE（44行）；若设置了该标志，说明bio请求被完全处理（一个bio请求可能多次执行才能完成），且请求的数据完全被读到内存中。

若数据完全就绪（56行），此时就设置page的PG_update标志，然后解锁页面（59行）。当读进程再次被调度执行时，就不会再被阻塞在lock_page_killable（）函数中了。

对于读取的数据如何拷贝到用户缓冲区中，do_generic_file_read（）函数中已作了分析。