Linux内核写文件过程

http://www.ilinuxkernel.com

目　录Table of Contents

图目录 List of Figures

1 概述

用户进程通过系统调用write（）往磁盘上写数据，但write（）执行结束后，数据是否立即写到磁盘上？内核读文件数据时，使用到了“提前读”；写数据时，则使用了“延迟写”，即write（）执行结束后，数据并没有立即立即将请求放入块设备驱动请求队列，然后写到硬盘上。

本文不考虑以O_DIRECT或O_SYNC方式打开文件的情形，我们仅考虑异步I/O。同步方式写的I/O流程相对简单。

本文以Redhat Enterprise Linux 6 Update 3内核版本2.6.32-279.el6.x86_64为例，分析内核写文件过程。

在分析具体源码之前，我们回顾一下Linux内核块设备I/O流程，如图1所示。

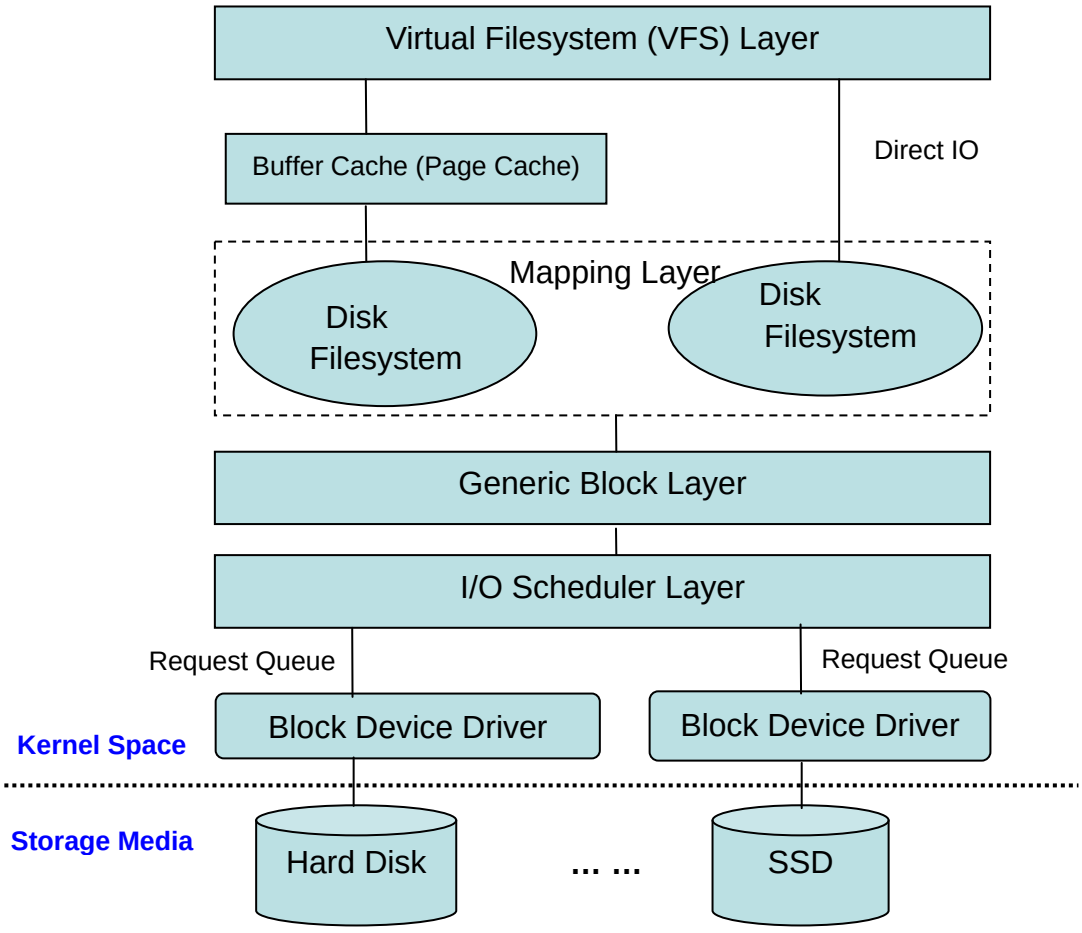

图1 Linux内核块设备I/O流程

为了简化块设备驱动分析，我们以Ramdisk块设备为基础，分析Linux内核写数据过程。本文实例将/dev/ram1格式化为ext4文件系统，然后读写文件。下面是写文件内核栈信息，从该栈信息中，可以看出整个I/O写操作，所经历的流程。

```
Pid: 4318, comm: cp Tainted: G          ---------- HT 2.6.32279.debug #19
Call Trace:
 [<ffffffff8135ce59>] ? brd_make_request+0x439/0x510
 [<ffffffff814fd503>] ? printk+0x41/0x46
 [<ffffffff81256f64>] ? generic_make_request+0x2c4/0x5b0
 [<ffffffff8125730c>] ? submit_bio+0xbc/0x160
 [<ffffffff811acd46>] ? submit_bh+0xf6/0x150
 [<ffffffffa0109773>] ? ext4_mb_init_cache+0x883/0x9f0 [ext4]
 [<ffffffff8112b560>] ? __lru_cache_add+0x40/0x90
 [<ffffffffa01099fe>] ? ext4_mb_init_group+0x11e/0x210 [ext4]
 [<ffffffffa0109bbd>] ? ext4_mb_good_group+0xcd/0x110 [ext4]
 [<ffffffffa010b35b>] ? ext4_mb_regular_allocator+0x19b/0x410 [ext4]
 [<ffffffff814feffe>] ? mutex_lock+0x1e/0x50
 [<ffffffffa010d22d>] ? ext4_mb_new_blocks+0x38d/0x560 [ext4]
 [<ffffffffa0100ace>] ? ext4_ext_find_extent+0x2be/0x320 [ext4]
 [<ffffffffa0103b83>] ? ext4_ext_get_blocks+0x1113/0x1a10 [ext4]
 [<ffffffff810097cc>] ? __switch_to+0x1ac/0x320
 [<ffffffff81136959>] ? zone_statistics+0x99/0xc0
 [<ffffffff811259f1>] ? get_page_from_freelist+0x3d1/0x820
 [<ffffffffa00dfd39>] ? ext4_get_blocks+0xf9/0x2a0 [ext4]
 [<ffffffffa00e07ad>] ? ext4_get_block+0xbd/0x120 [ext4]
 [<ffffffff811afe1b>] ? __block_prepare_write+0x1db/0x570
 [<ffffffffa00e06f0>] ? ext4_get_block+0x0/0x120 [ext4]
 [<ffffffff811b048c>] ? block_write_begin_newtrunc+0x5c/0xd0
 [<ffffffff811b0893>] ? block_write_begin+0x43/0x90
 [<ffffffffa00e06f0>] ? ext4_get_block+0x0/0x120 [ext4]
 [<ffffffffa00e4896>] ? ext4_write_begin+0x226/0x2d0 [ext4]
 [<ffffffffa00e06f0>] ? ext4_get_block+0x0/0x120 [ext4]
 [<ffffffffa00e4ac8>] ? ext4_da_write_begin+0x188/0x200 [ext4]
 [<ffffffffa00ab63f>] ? jbd2_journal_dirty_metadata+0xff/0x150 [jbd2]
 [<ffffffff814ff6f6>] ? down_read+0x16/0x30
 [<ffffffff81114ab3>] ? generic_file_buffered_write+0x123/0x2e0
 [<ffffffff810724c7>] ? current_fs_time+0x27/0x30
 [<ffffffff81116450>] ? __generic_file_aio_write+0x250/0x480
 [<ffffffff8113f1c7>] ? handle_pte_fault+0xf7/0xb50
 [<ffffffff811166ef>] ? generic_file_aio_write+0x6f/0xe0
 [<ffffffffa00d9131>] ? ext4_file_write+0x61/0x1e0 [ext4]
 [<ffffffff8117ad6a>] ? do_sync_write+0xfa/0x140
 [<ffffffff810920d0>] ? autoremove_wake_function+0x0/0x40
 [<ffffffff810edfc2>] ? ring_buffer_lock_reserve+0xa2/0x160
```

[<ffffffff810d69e2>] ? audit_syscall_entry+0x272/0x2a0
[<ffffffff81213136>] ? security_file_permission+0x16/0x20
[<ffffffff8117b068>] ? vfs_write+0xb8/0x1a0
[<ffffffff8117ba81>] ? sys_write+0x51/0x90
[<ffffffff8100b308>] ? tracesys+0xd9/0xde

2 虚拟文件系统与ext4文件系统层

分析源码之前，我们先看write（）函数原型

#include <unistd.h>

*ssize_t write(int fd, const void *buf, size_t count);*

2.1 sys_write（）

write（）系统调用在内核的实现为sys_write（），定义在文件include/linux/syscalls.h。

```
00635: asmlinkage long sys_write(unsigned int fd, const char __user
00636:                 *buf,    size_t count);
```

sys_write（）的实现定义在fs/read_write.c。

```
00389: SYSCALL_DEFINE3(write, unsigned int, fd, const char
00390:         __user *buf,    size_t, count)
00391: {
00392:         struct file *file;
00393:         ssize_t ret = -EBADF;
00394:         int fput_needed;
00395:
00396:     file = fget_light(fd, &fput_needed);
00397:     if (file) {
00398:         loff_t pos = file_pos_read(file);
00399:         ret = vfs_write(file, buf , count, &pos);
00400:         file_pos_write(file, pos);
00401:         fput_light(file, fput_needed);
00402:     }
00403:
00404:     return ret;
00405: }
```

396行的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write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。

若进程fd表是共享的，则fget_light（）函数中将fput_needed的值设为1；且通过get_file（）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_light（）来递减文件对象引用计数，该inline函数源码在include/linux/file.h中。

file_pos_read（）和file_pos_write（）的功能单一，一个是读取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。

```
00362: static inline loff_t file_pos_read(struct file *file)
00363: {
00364:     return file->f_pos;
00365: }
00366:
00367: static inline void file_pos_write(struct file *file, loff_t pos)
00368: {
00369:     file->f_pos = pos;
00370: }
```

sys_write（）实现主体是函数vfs_write（）。

2.2 vfs_write（）

vfs_write（）的实现也在文件fs/read_write.c中。

```
00332: ssize_t vfs_write(struct file *file, const char __user *buf,
00332:                   size_t count, loff_t *pos)
00333: {
00334:     ssize_t ret;
00335:
00336:     if (!(file->f_mode & FMODE_WRITE))
00337:         return -EBADF;
00338:     if (! file->f_op || (! file->f_op->write &&
                                    !file->f_op->aio_write))
00339:         return -EINVAL;
00340:     if (unlikely(!access_ok(VERIFY_READ, buf, count)))
00341:         return -EFAULT;
00342:
00343:     ret = rw_verify_area(WRITE, file, pos, count);
00344:     if (ret >= 0) {
00345:         count = ret;
00346:         if (file->f_op->write)
00347:             ret = file->f_op->write(file, buf, count, pos);
```

```
00348:          else
00349:              ret = do_sync_write(file, buf, count, pos);
00350:          if (ret > 0) {
00351:              fsnotify_modify(file->f_path.dentry);
00352:              add_wchar(current, ret);
00353:          }
00354:          inc_syscw(current);
00355:      }
00356:
00357:      return ret;
00358: } ? end vfs_write ?
00359:
00360: EXPORT_SYMBOL(vfs_write);
```

函数首先进行合法性检查。若传进来的文件模式不是写，或者文件对象没有文件操作方法、或没有write方法或没有aio_write方法，都将直接返回错误，而不进行数据操作（336～341行）。

rw_verify_area（）（343行）检查文件从当前位置f_pos开始的count个字节是否对写操作加上了“强制锁”，这是通过调用函数完成的，其代码在fs.h中。通过了对强制锁的检查后，还要通过security_file_permission（）检查文件是否可读，这种情形极少用，我们就不予以关注。

通过合法性检查后，就调用具体文件系统定义了file_operations中的write方法。对于ext4文件系统，file_operations方法定义在fs/ext4/file.c中。从定义中，可知write方法实现函数为do_sync_write（）。注意，虽然这里write方法为do_sync_write（），并不意味着是同步写。是否执行同步写，在后面的函数中确定。

```
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_write（）

do_sync_write（）也在文件fs/read_write.c中。

```
00307: ssize_t do_sync_write(struct file *filp, const char __user
00307:     *buf, size_t len, loff_t *ppos)
00308: {
00309:         struct iovec iov = { .iov_base = (void __user *)buf, .iov_len =
                                    len };
00310:         struct kiocb kiocb;
00311:         ssize_t ret;
00312:
00313:         init_sync_kiocb(&kiocb, filp);
00314:         kiocb.ki_pos = *ppos;
00315:         kiocb.ki_left = len;
00316:
00317:         for (;;) {
00318:             ret = filp->f_op->aio_write(&kiocb, &iov, 1, kiocb.ki_pos);
00319:             if (ret != -EIOCBRETRY)
00320:                 break;
00321:             wait_on_retry_sync_kiocb(&kiocb);
00322:         }
00323:
00324:         if (-EIOCBQUEUED == ret)
00325:             ret = wait_on_sync_kiocb(&kiocb);
00326:         *ppos = kiocb.ki_pos;
00327:         return ret;
00328: } ?   end do_sync_write ?
00329:
```

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。

异步 I/O 的实现是可选的，大部分设备会从这个能力中受益。块和网络驱动在整个时间是完全异步的，因此只有字符驱动对于明确的异步 I/O 支持是候选的。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。

定义了一个临时变量iov，该临时变量记录了用户空间缓冲区地址buf和所要写的字节数len，请大家留意用户空间的缓冲区地址buf是保存在iov中的。

初始化异步I/O数据结构后（313～315行），就调用file_operations中的aio_write方法。ext4文件系统对该方法实现函数是ext4_file_write（）（见ext4_file_operations结构体定义）。

2.4 ext4_file_write（）

2.4.1 Ext4文件系统extent特性

Ext2/3等老Linux文件系统使用间接块映射模式(block mapping)，文件的每一个块都要被记录下来，这使得对大文件的操作（如删除）效率低下。Ext4引入Extents这一概念来代替ext2/3使用的传统的块映射(block mapping)方式。“extent”是一个大的连续的物理块区域，当块大小为4KB时，ext4中的一个extent最大可以映射128MB的连续物理存储空间。

例如，Ext3 采用间接块映射，当操作大文件时，效率极其低下。比如一个 100MB 大小的文件，在 Ext3 中要建立 25,600 个数据块（每个数据块大小为 4KB）的映射表。而 Ext4 引入了现代文件系统中流行的 extents 概念，每个 extent 为一组连续的数据块，上述文件则表示为“该文件数据保存在接下来的 25,600 个数据块中”，提高了不少效率。

Ext4文件系统支持新的Extents模式，也支持传统的块映射模式。

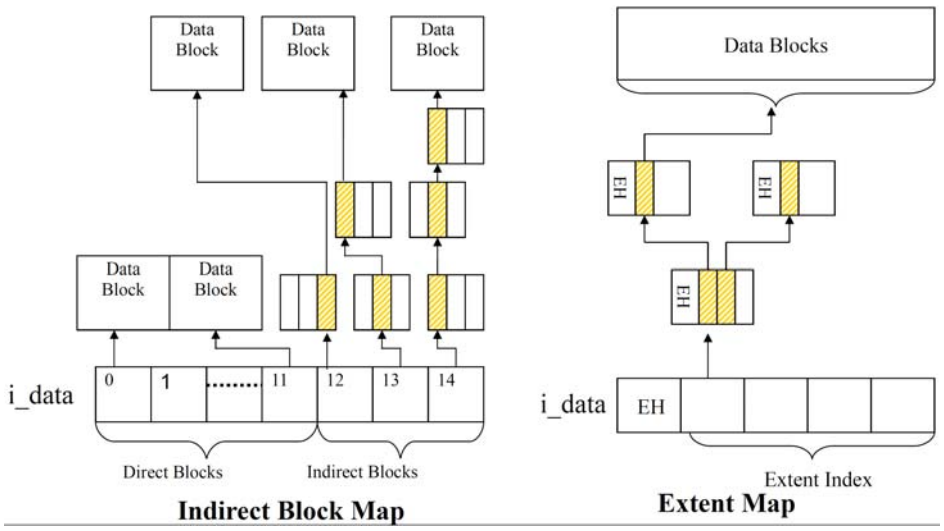

图2 Ext4文件系统间接块映射与Extent模式

Extent模式主要数据结构包括ext4_extent, ext4_extent_idx, ext4_extent_header，均定义在文件fs/ext4/ext4_extents.h文件中。

```
00068: /*
00069:  * This is the extent on-disk structure.
00070:  * It's used at the bottom of the tree.
00071:  */
00072: struct ext4_extent {
00073:     __le32 ee_block; /* first logical block extent covers */
00074:     __le16 ee_len;       /* number of blocks covered by extent */
00075:     __le16 ee_start_hi;  /* high 16 bits of physical block */
00076:     __le32 ee_start_lo;  /* low 32 bits of physical block */
00077: };
00078:
```

```
00078:
00079: /*
00080:  * This is index on-disk structure.
00081:  * It's used at all the levels except the bottom.
00082:  */
00083: struct ext4_extent_idx {
00084:     __le32 ei_block; /* index covers logical blocks from 'block' */
00085:     __le32 ei_leaf_lo;    /* pointer to the physical block of the next *
00086:                 * level. leaf or next index could be there */
00087:     __le16 ei_leaf_hi;    /* high 16 bits of physical block */
00088:     __u16   ei_unused;
00089: };
00090:
00091: /*
00092:  * Each block (leaves and indexes), even inode-stored has header.
00093:  */
00094: struct ext4_extent_header {
00095:     __le16 eh_magic;     /* probably will support different formats */
00096:     __le16 eh_entries;   /* number of valid entries */
00097:     __le16 eh_max;       /* capacity of store in entries */
00098:     __le16 eh_depth;     /* has tree real underlying blocks? */
00099:     __le32 eh_generation;     /* generation of the tree */
00100: };
```

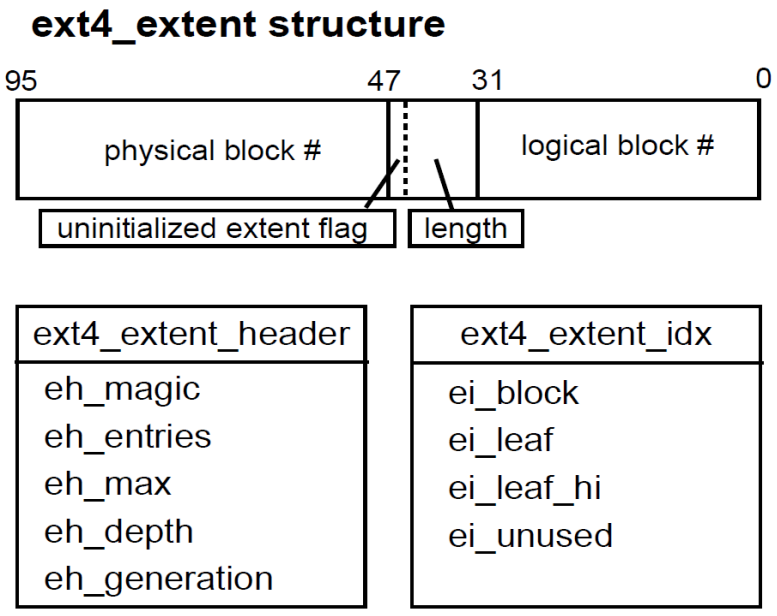

图3 ext4_extent、ext4_extent_idx、ext4_extent_header数据结构

ext4_extent, ext4_extent_idx, ext4_extent_header三个数据结构关系，如下图所示。

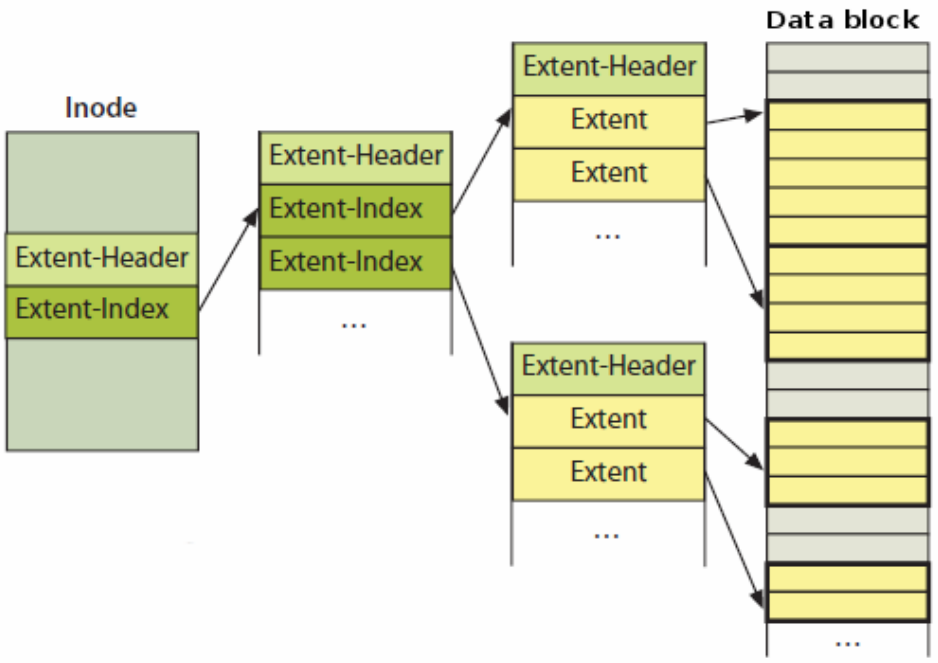

图4 ext4 extent树

2.4.2 ext4_file_write（）函数分析

ext4_file_write（）函数的源码在文件fs/ext4/file.c中。

```
00091: static ssize_t
00092: ext4_file_write(struct kiocb *iocb, const struct iovec *iov,
00093:         unsigned long nr_segs, loff_t pos)
00094: {
00095:     struct inode *inode = iocb->ki_filp->f_path.dentry->d_inode;
00096:     int unaligned_aio = 0;
00097:     ssize_t ret;
00098:
00099:     /*
00100:      * If we have encountered a bitmap-format file, the size limit
00101:      * is smaller than s_maxbytes, which is for extent-mapped files.
00102:      */
00103:
00104:     if (!(ext4_test_inode_flag(inode, EXT4_INODE_EXTENTS))) {
00105:         struct ext4_sb_info *sbi = EXT4_SB(inode->i_sb);
00106:         size_t length = iov_length(iov, nr_segs);
00107:
00108:         if ((pos > sbi->s_bitmap_maxbytes ||
00109:           (pos == sbi->s_bitmap_maxbytes && length > 0)))
00110:             return -EFBIG;
00111:
00112:         if (pos + length > sbi->s_bitmap_maxbytes) {
00113:             nr_segs = iov_shorten((struct iovec *)iov, nr_segs,
00114:                     sbi->s_bitmap_maxbytes - pos);
```

```
00115:          }
00116:      } else if (unlikely((iocb->ki_filp->f_flags & O_DIRECT) &&
00117:                      !is_sync_kiocb(iocb)))
00118:          unaligned_aio = ext4_unaligned_aio(inode, iov, nr_segs,
                                                   pos);
00119:
```

函数首先检查文件是否为Ext4的extent模式，若为传统的块映射方式，先检查文件是否过大（104～110行）。若当前文件位置加上待写的数据长度，大小若超过最大文件限制，则要做相应的调整（112～115行），最终文件大小不能超过sbi->s_bitmap_maxbytes。

若不是extent模式，但写方法为O_DIRECT，就要判断I/O是否为Block对齐（116～118行）。

```
00120:      /* Unaligned direct AIO must be serialized; see comment above */
00121:      if (unaligned_aio) {
00122:          static unsigned long unaligned_warn_time;
00123:
00124:          /* Warn about this once per day */
00125:          if (printk_timed_ratelimit(&unaligned_warn_time,
                                           60*60*24*HZ))
00126:              ext4_msg(inode->i_sb, KERN_WARNING,
00127:                   "Unaligned AIO/DIO on inode %ld by %s; "
00128:                   "performance will be poor.",
00129:                   inode->i_ino, current->comm);
00130:
00131:          mutex_lock(&EXT4_I(inode)->i_aio_mutex);
00132:          ext4_aiodio_wait(inode);
00133:      }
00134:
```

若待写的I/O不是块对齐的，就必须串行写，不能并行写（121～133行）。这种情形很少见，这里不作深入分析。

```
00135:      ret = generic_file_aio_write(iocb, iov, nr_segs, pos);
00136:
00137:      if (unaligned_aio)
00138:          mutex_unlock(&EXT4_I(inode)->i_aio_mutex);
00139:
00140:      return ret;
00141: } ?   end ext4_file_write ?
```

generic_file_aio_write（）（135行）就是ext4_file_write（）的主体执行语句。若I/O不是块对齐，写操作完成后，还要对i_aio_mutex解锁（137～138行）。

2.5 generic_file_aio_write（）

generic_file_aio_write（）的源码在文件mm/filemap.c中。

```
02544:
02545: /**
02546:  * generic_file_aio_write - write data to a file
02547:  * @iocb:   IO state structure
02548:  * @iov:    vector with data to write
02549:  * @nr_segs:   number of segments in the vector
02550:  * @pos:    position in file where to write
02551:  *
02552:  * This is a wrapper around ____generic_file_aio_write() to be used by most
02553:  * filesystems. It takes care of syncing the file in case of O_SYNC file
02554:  * and acquires i_mutex as needed.
02555:  */
02556: ssize_t generic_file_aio_write(struct kiocb *iocb, const
02557:         struct iovec *iov, unsigned long nr_segs, loff_t pos)
02558: {
02559:     struct file *file = iocb->ki_filp;
02560:     struct inode *inode = file->f_mapping->host;
02561:     ssize_t ret;
02562:
02563:     BUG_ON(iocb->ki_pos != pos);
02564:
02565:     mutex_lock(&inode->i_mutex);
02566:     ret = __generic_file_aio_write(iocb, iov, nr_segs,
                            &iocb->ki_pos);
02567:     mutex_unlock(&inode->i_mutex);
02568:
02569:     if (ret > 0 || ret == -EIOCBQUEUED) {
02570:         ssize_t err;
02571:
02572:         err = generic_write_sync(file, pos, ret);
02573:         if (err < 0 && ret > 0)
02574:             ret = err;
02575:     }
02576:     return ret;
02577: } ?   end generic_file_aio_write ?
02578: EXPORT_SYMBOL(generic_file_aio_write);
02579:
```

该函数实际上是对__generic_file_aio_write（）函数的封装。

在do_sync_write（）中已经将当前文件写的起始位置记录在iocb->ki_pos，为了防止多个进程同时写文件改了起始位置，再次检查起始位置是否发生了变化，若变化了，说明这次写操作，存在Bug（2563行）。

接下来执行主体函数__generic_file_aio_write（），执行写操作前要加锁（2565行），

完成后解锁（2567行）。若写操作成功，就返回写完成的字节数，返回值大于0；写操作出现错误，就返回相应的错误码。如果返回了 -EIOCBQUEUED，说明这个 iocb 已经进入了workqueue，但未完成（2569行），接下来就要调用generic_write_sync（）将数据刷新到硬盘上（2569～2575行）。

2.6 __generic_file_aio_write（）

写文件的主要过程都在__generic_file_aio_write（）函数中完成（文件mm/filemap.c）。

```
02426: /**
02427:  * ____generic_file_aio_write -  write data to a file
02428:  * @iocb:   IO state structure (file, offset, etc.)
02429:  * @iov:    vector with data to write
02430:  * @nr_segs:   number of segments in the vector
02431:  * @ppos:  position where to write
02432:  *
02433:  * This function does all the work needed for actually writing data to a
02434:  * file. It does all basic checks, removes SUID from the file, updates
02435:  * modification times and calls proper subroutines depending on whether
02436:  * we do direct IO or a standard buffered write.
02437:  *
02438:  * It expects i_mutex to be grabbed unless we work on a block device or
02439:  * similar object which does not need locking at all.
02440:  *
02441:  * This function does *not* take care of syncing data in case of O_SYNC
02442:  * write. A caller has to handle it. This is mainly due to the fact that we
02443:  * want to avoid syncing under i_mutex.
02444:  */
02445: ssize_t ___generic_file_aio_write(struct kiocb *iocb,
02445:     const struct iovec *iov ,
02446:                 unsigned long nr_segs, loff_t *ppos)
02447: {
02448:     struct file *file = iocb->ki_filp;
02449:     struct address_space * mapping = file->f_mapping;
02450:     size_t ocount;      / * original count */
02451:     size_t count;       / * after file limit checks */
02452:     struct inode    *inode = mapping->host;
02453:     loff_t      pos;
02454:     ssize_t     written;
02455:     ssize_t     err;
02456:
02457:     ocount = 0;
02458:     err = generic_segment_checks(iov, &nr_segs, &ocount,
                        VERIFY_READ);
02459:     if (err)
02460:         return err;
02461:
02462:     count = ocount;
```

```
02463:      pos = *ppos;
02464:
```

先调用generic_segment_checks（）来检查iovec中给出的用户缓冲区有效性（2458行）。通过检查后，更新检查后的实际可写入数据大小（大多数情况下不变，只有待写的数据超出文件大小限制，count值才会变化）。

```
02465:      vfs_check_frozen(inode->i_sb, SB_FREEZE_WRITE);
02466:
02467:      /* We can write back this queue in page reclaim */
02468:      current->backing_dev_info = mapping->backing_dev_info;
02469:      written = 0;
02470:
02471:      err = generic_write_checks(file, &pos, &count,
                        S_ISBLK(inode->i_mode));
02472:      if (err)
02473:          goto ↓out;
02474:
02475:      if (count == 0)
02476:          goto ↓out;
02477:
02478:      err = file_remove_suid(file);
02479:      if (err)
02480:          goto ↓out;
02481:
02482:      file_update_time(file);
02483:
```

vfs_check_frozen（）和Snapshot功能有关，我们可以忽略它。

设置current->backing_dev_info的值为file->f_mapping->backing_dev_info（2468行）。设置该值的目的是为允许当前进程将file->f_mapping拥有的脏页面回写到磁盘上，即使相应的请求队列拥塞，也是如此。

generic_write_checks（）来检查对该文件的是否有相应的写权限，这个和系统中是否对文件大小有限制有关（2471行）。

将文件的suid标志清0，而且如果是可执行文件的话，就将sgid标志也清0（2478行）。既然写文件，那么文件就会被修改（或创建），修改文件的时间是要记录在inode中的，并且将inode标记为脏（回写到磁盘上），file_update_time（）更新文件的访问时间（2482行）。

```
02484:      /* coalesce the iovecs and go direct-to-BIO for O_DIRECT */
02485:      if (unlikely(file->f_flags & O_DIRECT)) {
02486:          loff_t endbyte;
```

```
02487:          ssize_t written_buffered;
02488:
02489:          written = generic_file_direct_write(iocb, iov, &nr_segs,
02490:                                      pos,ppos, count, ocount);
02491:          if (written < 0 || written == count)
02492:              goto ↓out;
02493:          /*
02494:           * direct-io write to a hole: fall through to buffered I/O
02495:           * for completing the rest of the request.
02496:           */
02497:          pos += written;
02498:          count -= written;
02499:          written_buffered = generic_file_buffered_write(iocb,
02500:                          iov,nr_segs, pos, ppos, count,
02501:                          written);
02502:          /*
02503:           * If generic_file_buffered_write() retuned a synchronous error
02504:           * then we want to return the number of bytes which were
02505:           * direct-written, or the error code if that was zero.  Note
02506:           * that this differs from normal direct-io semantics, which
02507:           * will return -EFOO even if some bytes were written.
02508:           */
02509:          if (written_buffered < 0) {
02510:              err = written_buffered;
02511:              goto ↓out;
02512:          }
02513:
02514:          /*
02515:           * We need to ensure that the page cache pages are written to
02516:           * disk and invalidated to preserve the expected O_DIRECT
02517:           * semantics.
02518:           */
02519:          endbyte = pos + written_buffered - written - 1;
02520:          err = do_sync_mapping_range(file->f_mapping, pos,
02521:                      endbyte,SYNC_FILE_RANGE_WAIT_BEFORE|
02522:                              SYNC_FILE_RANGE_WRITE|
02523:                              SYNC_FILE_RANGE_WAIT_AFTER);
02524:          if (err == 0) {
02525:              written = written_buffered;
02526:              invalidate_mapping_pages(mapping,
02527:                              pos >> PAGE_CACHE_SHIFT,
02528:                              endbyte >> PAGE_CACHE_SHIFT);
02529:          } else {
02530:              /*
02531:               * We don't know how much we wrote, so just return
02532:               * the number of bytes which were direct-written
02533:               */
02534:          }
02535:      } else {
02536:          written = generic_file_buffered_write(iocb, iov,
02537:                      nr_segs,pos, ppos, count, written);
02538:      }
02539:  out:
```

```
02540:      current->backing_dev_info = NULL;
02541:      return written ? written : err;
02542: } ?   end ____generic_file_aio_write ?
02543: EXPORT_SYMBOL(__generic_file_aio_write);
```

若写方式为Direct IO（不经过页面Cache），那么就会执行2485～2535行。本节先考虑写操作经过页面Cache的情形，我们将单独分析Linux内核Cache机制。

前面的工作都是一些合法性检查、记录文件改变、修改时间。而写文件的主要工作是调用函数generic_file_buffered_write（）来完成。

2.7 generic_file_buffered_write（）

定义新的结构体struct iov_iter，将写文件的信息，如位置、写数据大小、数据所在位置做进一步封装（2405行），然后调用generic_perform_write（）执行写操作。

```
02395: ssize_t
02396: generic_file_buffered_write(struct kiocb *iocb,
const struct iovec *iov,
02397:          unsigned long nr_segs, loff_t pos, loff_t *ppos,
02398:          size_t count, ssize_t written)
02399: {
02400:      struct file *file = iocb->ki_filp;
02401:      struct address_space *mapping = file->f_mapping;
02402:      ssize_t status;
02403:      struct iov_iter i;
02404:
02405:      iov_iter_init(&i, iov, nr_segs, count, written);
02406:      status = generic_perform_write(file, &i, pos);
02407:
02408:      if (likely(status >= 0)) {
02409:          written += status;
02410:          *ppos = pos + status;
02411:      }
02412:
02413:      /*
02414:       * If we get here for O_DIRECT writes then we must have fallen through
02415:       * to buffered writes (block instantiation inside i_size).   So we sync
02416:       * the file data here, to try to honour O_DIRECT expectations.
02417:       */
02418:      if (unlikely(file->f_flags & O_DIRECT) && written)
02419:          status = filemap_write_and_wait_range(mapping,
02420:                      pos, pos + written - 1);
02421:
02422:      return written ? written : status;
02423: } ?   end generic_file_buffered_write ?
02424: EXPORT_SYMBOL(generic_file_buffered_write);
```

02425:

通常情况下，写文件都会成功，generic_perform_write（）返回后，更新已完成写入的数据量（2409行）和文件当前位置（2410行）。

若文件打开模式为O_DIRECT，于是就要调用filemap_write_and_wait_range（），将文件内 [pos, pos+written-1] 范围内的数据同步到磁盘上。

2.8 generic_perform_write（）

2.8.1 ext4文件系统address_space_operations

ext4文件系统相对于ext3，增加了一种address_space_operations方法ext4_da_aops，da的含义就是delay allocation。这些方法定义均在文件fs/ext4/inode.c文件中。

```
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:
04119: static const struct address_space_operations
              ext4_writeback_aops = {
04120:     .readpage       = ext4_readpage,
04121:     .readpages      = ext4_readpages,
04122:     .writepage      = ext4_writepage,
04123:     .sync_page      = block_sync_page,
04124:     .write_begin        = ext4_write_begin,
04125:     .write_end      = ext4_writeback_write_end,
04126:     .bmap           = ext4_bmap,
04127:     .invalidatepage     = ext4_invalidatepage,
04128:     .releasepage        = ext4_releasepage,
04129:     .direct_IO      = ext4_direct_IO,
04130:     .migratepage        = buffer_migrate_page,
04131:     .is_partially_uptodate  = block_is_partially_uptodate,
04132:     .error_remove_page  = generic_error_remove_page,
```

```
04133: };
04134:
04135: static const struct address_space_operations
           ext4_journalled_aops = {
04136:     .readpage       = ext4_readpage,
04137:     .readpages      = ext4_readpages,
04138:     .writepage      = ext4_writepage,
04139:     .sync_page      = block_sync_page,
04140:     .write_begin        = ext4_write_begin,
04141:     .write_end      = ext4_journalled_write_end,
04142:     .set_page_dirty         = ext4_journalled_set_page_dirty,
04143:     .bmap           = ext4_bmap,
04144:     .invalidatepage     = ext4_invalidatepage,
04145:     .releasepage        = ext4_releasepage,
04146:     .is_partially_uptodate  = block_is_partially_uptodate,
04147:     .error_remove_page  = generic_error_remove_page,
04148: };
04149:
04150: static const struct address_space_operations
         ext4_da_aops = {
04151:     .readpage       = ext4_readpage,
04152:     .readpages      = ext4_readpages,
04153:     .writepage      = ext4_writepage,
04154:     .writepages     = ext4_da_writepages,
04155:     .sync_page      = block_sync_page,
04156:     .write_begin        = ext4_da_write_begin,
04157:     .write_end      = ext4_da_write_end,
04158:     .bmap           = ext4_bmap,
04159:     .invalidatepage     = ext4_da_invalidatepage,
04160:     .releasepage        = ext4_releasepage,
04161:     .direct_IO      = ext4_direct_IO,
04162:     .migratepage        = buffer_migrate_page,
04163:     .is_partially_uptodate  = block_is_partially_uptodate,
04164:     .error_remove_page  = generic_error_remove_page,
04165: };
04166:
```

2.8.2 ext4文件系统delay allocation机制

延时分配(Delayed allocation)该技术也称为allocate-on-flush，可以提升文件系统的性能。只有buffer IO中每次写操作都会涉及的磁盘块分配过程推迟到数据回写时再进行，即数据将要被真正写入磁盘时，文件系统才为其分配块，这与其它文件系统在早期就分配好必要的块是不同的。另外，由于ext4的这种做法可以根据真实的文件大小做块分配决策，它还减少了碎片的产生。

通常在进行Buffer Write时，系统的实际操作仅仅是为这些数据在操作系统内分配内存页(page cache)并保存这些数据，等待用户调用fsync等操作强制刷新或者等待系统触发定

时回写过程。在数据拷贝到page cache这一过程中，系统会为这些数据在磁盘上分配对应的磁盘块。

而在使用delalloc（delay allocation）后，上面的流程会略有不同，在每次buffer Write时，数据会被保存到page cache中，但是系统并不会为这些数据分配相应的磁盘块，仅仅会查询是否有已经为这些数据分配过磁盘块，以便决定后面是否需要为这些数据分配磁盘块。在用户调用fsync或者系统触发回写过程时，系统会尝试为标记需要分配磁盘块的这些数据分配磁盘块。这样，文件系统可以为这些属于同一个文件的数据分配尽量连续的磁盘空间，从而优化后续文件的访问性能。

2.8.3 genric_perform_write（）

generic_perform_write（）函数实现也在文件mm/filemap.c。

```
02302: static ssize_t generic_perform_write(struct file *file,
02303:                 struct iov_iter *i, loff_t pos)
02304: {
02305:     struct address_space *mapping = file->f_mapping;
02306:     const struct address_space_operations *a_ops =
                       mapping->a_ops;
02307:     long status = 0;
02308:     ssize_t written = 0;
02309:     unsigned int flags = 0;
02310:
02311:     /*
02312:      * Copies from kernel address space cannot fail (NFSD is a big user).
02313:      */
02314:     if (segment_eq(get_fs(), KERNEL_DS))
02315:         flags |= AOP_FLAG_UNINTERRUPTIBLE;
02316:
```

若当前I/O操作是属于在内核中进行，显然是不能被中断的（用户态的I/O操作可以被中断），就要设置AOP_FLAG_UNINTERRUPTIBLE标志（2314～2315行）。

```
02317:     do {
02318:         struct page *page;
02319:         pgoff_t index;      /* Pagecache index for current page */
02320:         unsigned long offset;     /* Offset into pagecache page */
02321:         unsigned long bytes;   /* Bytes to write to page */
02322:         size_t copied;      /* Bytes copied from user */
02323:         void *fsdata;
02324:
```

```
02325:          offset = (pos & (PAGE_CACHE_SIZE - 1));
02326:          index = pos >> PAGE_CACHE_SHIFT;
02327:          bytes = min_t(unsigned long, PAGE_CACHE_SIZE - offset,
02328:                                  iov_iter_count(i));
```

index：当前pos位置在pagecache的索引（以页面大小为单位）。

offset：为在页面内的偏移。

bytes：要从用户空间拷贝的数据大小。

```
02329:
02330: again:
02331:          /*
02332:           * Bring in the user page that we will copy from _first_.
02333:           * Otherwise there's a nasty deadlock on copying from the
02334:           * same page as we're writing to, without it being marked
02335:           * up-to-date.
02336:           *
02337:           * Not only is this an optimisation, but it is also required
02338:           * to check that the address is actually valid, when atomic
02339:           * usercopies are used, below.
02340:           */
02341:          if (unlikely(iov_iter_fault_in_readable(i, bytes))) {
02342:                  status = -EFAULT;
02343:                  break;
02344:          }
02345:
02346:          status = a_ops->write_begin(file, mapping, pos, bytes, flags,
02347:                                  &page, &fsdata);
02348:          if (unlikely(status))
02349:                  break;
02350:
```

调用索引节点（file->f_mapping）中address_space对象的write_begin方法（2346行），write_begin方法会为该页分配和初始化缓冲区首部。稍后，我们会详细分析ext4文件系统实现的write_begin方法ext4_da_write_begin（）。

```
02351:          if (mapping_writably_mapped(mapping))
02352:                  flush_dcache_page(page);
02353:
```

mapping->i_mmap_writable记录VM_SHAREE共享映射数。若mapping_writably_mapped（）不等于0，则说明该页面被多个共享使用，调用flush_dcache_page()。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。

```
02354:          pagefault_disable();
02355:          copied = iov_iter_copy_from_user_atomic(page, i,
                            offset, bytes);
02356:          pagefault_enable();
02357:          flush_dcache_page(page);
02358:
02359:          mark_page_accessed(page);
```

用户写的文件数据在用户态，内核是不能直接使用用户地址空间进行数据传输的（访问用户态的地址，可能会产生页面中断），因此在将数据写到硬盘前，先将待写的数据从用户态拷到内核里面的空间。拷贝数据前，先关闭页面中断，拷贝完成后，使能页面中断，并标记页面被访问（2354～2359行）。

```
02360:          status = a_ops->write_end(file, mapping, pos, bytes, copied,
02361:                                    page, fsdata);
02362:          if (unlikely(status < 0))
02363:              break;
02364:          copied = status;
02365:
02366:          cond_resched();
02367:
```

将待写的数据拷贝到内核空间后，调用ext4文件系统的address_space_operations的write_end方法。前面看到ext4文件系统有4种模式：writeback、ordered、journalled和delay allocation。加载ext4分区时，默认方式为delay allocation。对应的write_end方法为ext4_da_write_end（）。

cond_resched（）检查当前进程的TIF_NEED_RESCHED标志，若该标志为设置，则调用schedule函数，调度一个新程序投入运行。

```
02368:          iov_iter_advance(i, copied);
02369:          if (unlikely(copied == 0)) {
02370:              /*
02371:               * If we were unable to copy any data at all, we must
02372:               * fall back to a single segment length write.
02373:               *
02374:               * If we didn't fallback here, we could livelock
02375:               * because not all segments in the iov can be copied at
02376:               * once without a pagefault.
02377:               */
```

```
02378:              bytes = min_t(unsigned long, PAGE_CACHE_SIZE -
02379:                              offset,iov_iter_single_seg_count(i));
02380:              goto ↑again;
02381:          }
```

当a_ops->write_end（）执行完成后，写数据操作完成了（注意，此时数据不一定真正写到磁盘上，因为大多数数据写为异步I/O）。接下来就要更新iov_iter结构体里的信息，包括文件的位置、写数据大小、数据所在位置（2368行）。若copied值为0，说明没能将数据从用户态拷贝到内核态，就要再次尝试写操作（2369～2380行）。

```
02382:          pos += copied;
02383:          written += copied;
02384:
02385:          balance_dirty_pages_ratelimited(mapping);
02386:          if (fatal_signal_pending(current)) {
02387:              status = -EINTR;
02388:              break;
02389:          }
02390:      } ?   end do ?  while (iov_iter_count(i));
02391:
02392:      return written ? written : status;
02393: } ?   end generic_perform_write  ?
02394:
```

2382～2383行更新文件位置pos和已完成写的数据大小。

调用balance_dirty_pages_ratelimited（）来检查页面Cache中的脏页比例是否超过一个阀值（通常为系统中页的40%）。若超过阀值，就调用writeback_inodes（）来刷新几十页到磁盘上（2385行）。

大家看到上面很多步骤，不免有疑问，generic_perform_write（）函数执行完后，数据被写到磁盘上去了吗？

在generic_perform_write（）函数执行完成后，我们应知道以下两点：

（1） 写数据已从用户空间拷贝到页面Cache中（内核空间）；

（2） 数据页面标记为脏；

（3） 数据还未写到磁盘上去，这就是“延迟写”技术。后面我们会分析何时、在哪里、怎样将数据写到磁盘上的。

1. ext4文件系统a_ops->begin_write（）方法ext4_da_write_begin（）

Linux文件系统VFS层将应用程序的写入数据分割成页面大小（默认为4KB）。对于每个页面，VFS会检查其是否已经为其创建了buffer_head结构，若没有创建，则为其创建buffer_head；若已经创建则检查每个buffer_head的状态，包括buffer_head是否已经与物理磁盘块建立映射等。这些就是a_ops->write_begin（）方法的主要功能。

a_ops->write_begin（）是由VFS提供的一个接口，具体文件系统负责实现该接口。ext4文件系统默认采用delay allocation方法，其实现的函数为ext4_da_write_begin（）。

```
03265: static int ext4_da_write_begin(struct file *file, struct
03265:                 address_space *  mapping,
03266:                 loff_t pos, unsigned len, unsigned flags,
03267:                 struct page **pagep, void **fsdata)
03268: {
03269:     int ret, retries = 0;
03270:     struct page *page;
03271:     pgoff_t index;
03272:     unsigned from, to;
03273:     struct inode *inode = mapping->host;
03274:     handle_t *handle;
03275:
03276:     index = pos >> PAGE_CACHE_SHIFT;
03277:     from = pos & (PAGE_CACHE_SIZE - 1);
03278:     to = from + len;
03279:
03280:     if (ext4_nonda_switch(inode->i_sb)) {
03281:         *fsdata = (void *)FALL_BACK_TO_NONDELALLOC;
03282:         return ext4_write_begin(file, mapping, pos,
03283:                     len, flags, pagep, fsdata);
03284:     }
```

采用延迟分配，也要考虑当前存储空间空闲块情况。若所剩余空闲块低于一定比例，就不能再采用delay allocation，切换到非延迟分配模式写数据（3280～3282行）。

```
03285:     *fsdata = (void *)0;
03286:     trace_ext4_da_write_begin(inode, pos, len, flags);
03287: retry:
03288:     /*
03289:      * With delayed allocation, we don't log the i_disksize update
03290:      * if there is delayed block allocation. But we still need
03291:      * to journalling the i_disksize update if writes to the end
03292:      * of file which has an already mapped buffer.
03293:      */
03294:     handle = ext4_journal_start(inode, 1);
03295:     if (IS_ERR(handle)) {
03296:         ret = PTR_ERR(handle);
```

```
03297:          goto ↓out;
03298:      }
03299:      /* We cannot recurse into the filesystem as the transaction is already
03300:       * started */
03301:      flags |= AOP_FLAG_NOFS;
03302:
03303:      page = grab_cache_page_write_begin(mapping, index,
                                               flags);
03304:      if (!page) {
03305:          ext4_journal_stop(handle);
03306:          ret = -ENOMEM;
03307:          goto ↓out;
03308:      }
```

打开ext4文件系统日志模式（3294行），设置标志位为AOP_FLAG_NOFS（3301行）。AOP_FLAG_NOFS标志告诉其他helper代码进行内存分配时，要清除GFP_FS标志。

调用grab_cache_page_write_begin（）在pagecache中查找页面（3303行）。如果找到了该页，则增加计数并设置PG_locked标志。如果该页不在页面Cache中，则分配一个新页，并调用add_to_page_cache_lru（），将该页插入页面Cache中，这个函数也会增加页面引用计数，并设置PG_locked标志。若grab_cache_page_write_begin（）没能成功返回页面，说明系统没有空闲内存了，就没法继续写数据到硬盘，失败返回（3304～3308行）。

```
03309:      *pagep = page;
03310:
03311:      ret = block_write_begin(file, mapping, pos, len, flags,
03312:                  pagep, fsdata,ext4_da_get_block_prep);
03313:      if (ret < 0) {
03314:          unlock_page(page);
03315:          ext4_journal_stop(handle);
03316:          page_cache_release(page);
03317:          /*
03318:           * block_write_begin may have instantiated a few blocks
03319:           * outside i_size.  Trim these off again. Don't need
03320:           * i_size_read because we hold i_mutex.
03321:           */
03322:          if (pos + len > inode->i_size)
03323:              ext4_truncate_failed_write(inode);
03324:      }
03325:
03326:      if (ret == -ENOSPC &&ext4_should_retry_alloc(inode->i_sb,
                        &retries))
03327:          goto ↑retry;
03328: out:
03329:      return ret;
03330: } ?   end ext4_da_write_begin ?
```

03331:

block_write_begin（）会检查页面所有的buffer_head的状态，如buffer_head是否已经建立映射等，对于没有建立映射的buffer_head，需要调用函数ext4_da_get_block_prep（）将其与物理磁盘块建立映射关系，并将block设置标志位BH_New、BH_Mapped、BH_Delay（表示该块在写入的时候再进行延迟分配）（3311行）。

在block_write_begin（）函数中，主要是对缓冲页和缓冲区头的操作。两者之间的关系如下图所示。

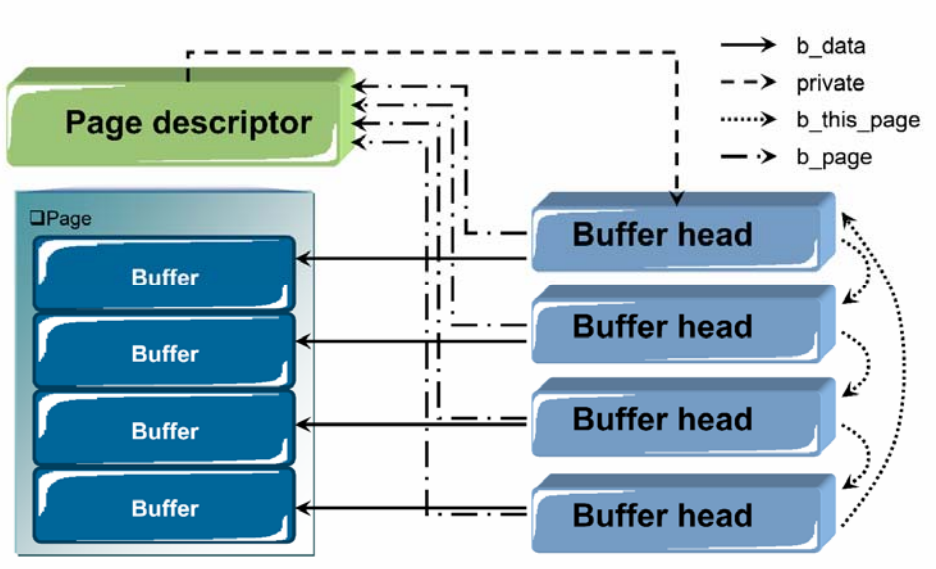

图5 缓冲页与缓冲区头的关系

2. ext4文件系统a_ops->write_end（）方法ext4_da_write_end（）

ext4_da_write_begin（）函数实现在文件fs/ext4/inode.c中。

```
03355: static int ext4_da_write_end(struct file *file,
03356:                 struct address_space *mapping,
03357:                 loff_t pos, unsigned len, unsigned copied,
03358:                 struct page *page, void *fsdata)
03359: {
03360:     struct inode *inode = mapping->host;
03361:     int ret = 0, ret2;
03362:     handle_t *handle = ext4_journal_current_handle();
03363:     loff_t new_i_size;
03364:     unsigned long start, end;
03365:     int write_mode = (int)(unsigned long)fsdata;
03366:
03367:     if (write_mode == FALL_BACK_TO_NONDELALLOC) {
03368:         switch (ext4_inode_journal_mode(inode)) {
```

```
03369:          case EXT4_INODE_ORDER_DATA_MODE:
03370:              return ext4_ordered_write_end(file, mapping,
03371:                          pos,len, copied, page, fsdata);
03372:          case EXT4_INODE_WRITEBACK_DATA_MODE:
03373:              return ext4_writeback_write_end(file, mapping,
03374:                          pos,len, copied, page, fsdata);
03375:          default:
03376:              BUG();
03377:          }
03378:      }
03379:
```

ext4_da_write_begin（）函数中，若硬盘上的空闲块低于一定比例，就不能继续使用delay allocation模式，并设置写数据模式为FALL_BACK_TO_NONDELALLOC。3367~3377行就是处理回退到传统写数据模式，包括ordered和writeback模式。

```
03380:      trace_ext4_da_write_end(inode, pos, len, copied);
03381:      start = pos & (PAGE_CACHE_SIZE - 1);
03382:      end = start + copied - 1;
03383:
03384:      /*
03385:       * generic_write_end() will run mark_inode_dirty() if i_size
03386:       * changes.  So let's piggyback the i_disksize mark_inode_dirty
03387:       * into that.
03388:       */
03389:
03390:      new_i_size = pos + copied;
03391:      if (new_i_size > EXT4_I(inode)->i_disksize) {
03392:          if (ext4_da_should_update_i_disksize(page, end)) {
03393:              down_write(&EXT4_I(inode)->i_data_sem);
03394:              if (new_i_size > EXT4_I(inode)->i_disksize) {
03395:                  /*
03396:                   * Updating i_disksize when extending file
03397:                   * without needing block allocation
03398:                   */
03399:                  if (ext4_should_order_data(inode))
03400:                      ret = ext4_jbd2_file_inode(handle,
03401:                                      inode);
03402:
03403:                  EXT4_I(inode)->i_disksize = new_i_size;
03404:              }
03405:              up_write(&EXT4_I(inode)->i_data_sem);
03406:              /* We need to mark inode dirty even if
03407:               * new_i_size is less that inode- >i_size
03408:               * bu greater than i_disksize.(hint delalloc)
03409:               */
03410:              ext4_mark_inode_dirty(handle, inode);
03411:          } ?   end if ext4_da_should_update... ?
03412:      } ?   end if new_i_size>EXT4_I(ino... ?
```

在ext4_inode_info结构体中，成员变量i_disksize记录磁盘上inode大小。i_disksize的值就是文件实际使用硬盘上的块大小。若当前文件位置pos加上待写的数据量copied超过i_disksize（3391行），再进一步检查delay allocation机制下是否需要更新i_disksize（3392行）。更新i_disksize时，还要检查是否需要分配块（3399行），最后将inode设置为dirty。

```
03413:      ret2 = generic_write_end(file, mapping, pos, len, copied,
03414:                               page, fsdata);
03415:      copied = ret2;
03416:      if (ret2 < 0)
03417:          ret = ret2;
03418:      ret2 = ext4_journal_stop(handle);
03419:      if (!ret)
03420:          ret = ret2;
03421:
03422:      return ret ? ret : copied;
03423: } ?   end ext4_da_write_end ?
03424:
```

接下来调用generic_write_end（），并关闭ext4文件系统的journal。

2.8.4 genric_write_end（）

generic_write_end（）函数在文件fs/buffer.c中。

```
02084: int generic_write_end(struct file *file, struct
                address_space *mapping,
02085:          loff_t pos, unsigned len, unsigned copied,
02086:          struct page *page, void *fsdata)
02087: {
02088:      struct inode *inode = mapping->host;
02089:      int i_size_changed = 0;
02090:
02091:      copied = block_write_end(file, mapping, pos, len, copied,
                            page, fsdata);
02092:
02093:      /*
02094:       * No need to use i_size_read() here, the i_size
02095:       * cannot change under us because we hold i_mutex.
02096:       *
02097:       * But it's important to update i_size while still holding page lock:
02098:       * page writeout could otherwise come in and zero beyond i_size.
02099:       */
02100:      if (pos+copied > inode->i_size) {
02101:          i_size_write(inode, pos+copied);
02102:          i_size_changed = 1;
02103:      }
```

```
02104:
02105:      unlock_page(page);
02106:      page_cache_release(page);
02107:
02108:      /*
02109:       * Don't mark the inode dirty under page lock. First, it unnecessarily
02110:       * makes the holding time of page lock longer. Second, it forces lock
02111:       * ordering of page lock and transaction start for journaling
02112:       * filesystems.
02113:       */
02114:      if (i_size_changed)
02115:          mark_inode_dirty(inode);
02116:
02117:      return copied;
02118: } ?   end generic_write_end ?
02119:  EXPORT_SYMBOL(generic_write_end);
02120:
```

调用block_write_end（）提交写请求（2091行），然后设置page的dirty标记（2114～2115行）。

2.8.5 block_write_end（）

block_write_end（）函数的实现也在文件fs/buffer.c中。

```
02048: int block_write_end(struct file *file, struct address_space
02049:              *mapping, loff_t pos, unsigned len, unsigned copied,
02050:              struct page *page, void *fsdata)
02051: {
02052:      struct inode *inode = mapping->host;
02053:      unsigned start;
02054:
02055:      start = pos & (PAGE_CACHE_SIZE - 1);
02056:
02057:      if (unlikely(copied < len)) {
02058:          /*
02059:           * The buffers that were written will now be uptodate, so we
02060:           * don't have to worry about a readpage reading them and
02061:           * overwriting a partial write. However if we have encountered
02062:           * a short write and only partially written into a buffer, it
02063:           * will not be marked uptodate, so a readpage might come in and
02064:           * destroy our partial write.
02065:           *
02066:           * Do the simplest thing, and just treat any short write to a
02067:           * non uptodate page as a zero-length write, and force the
02068:           * caller to redo the whole thing.
02069:           */
02070:          if (!PageUptodate(page))
02071:              copied = 0;
02072:
02073:          page_zero_new_buffers(page, start+copied, start+len);
```

```
02074:      }
02075:      flush_dcache_page(page);
02076:
02077:      /* This could be a short (even 0-length) commit */
02078:      __block_commit_write(inode, page, start, start+copied);
02079:
02080:      return copied;
02081: } ?   end block_write_end ?
02082: EXPORT_SYMBOL(block_write_end);
02083:
```

block_write_end（）主要是对__block_commit_write（）函数的封装。

__block_commit_write（）为page中的每一个buffer_head结构设置BH_Dirty标记。

```
01926: static int ___block_commit_write(struct inode *inode,
01927:      struct page *page, unsigned from, unsigned to)
01928: {
01929:      unsigned block_start, block_end;
01930:      int partial = 0;
01931:      unsigned blocksize;
01932:      struct buffer_head *bh, *head;
01933:
01934:      blocksize = 1 << inode->i_blkbits;
01935:
01936:      for(bh = head = page_buffers(page), block_start = 0;
01937:          bh != head || !block_start;
01938:          block_start=block_end, bh = bh->b_this_page) {
01939:          block_end = block_start + blocksize;
01940:          if (block_end <= from || block_start >= to) {
01941:              if (!buffer_uptodate(bh))
01942:                  partial = 1;
01943:          } else {
01944:              set_buffer_uptodate(bh);
01945:              mark_buffer_dirty(bh);
01946:          }
01947:          clear_buffer_new(bh);
01948:      }
01949:
01950:      /*
01951:       * If this is a partial write which happened to make all buffers
01952:       * uptodate then we can optimize away a bogus readpage() for
01953:       * the next read(). Here we 'discover' whether the page went
01954:       * uptodate as a result of this (potentially partial) write.
01955:       */
01956:      if (!partial)
01957:          SetPageUptodate(page);
01958:      return 0;
01959: } ?   end ____block_commit_write ?
01960:
```

__block_commit_write（）执行步骤如下：

（1）对页内受写操作影响的所有缓冲区，设置缓冲区头BH_Uptodate和BH_Dirty标志；

（2）将相应的inode标记为脏；

（3）若页中所有的缓冲区都是up-to-date，则设置页面PG_uptodate标志；

（4） 设置页面PG_dirty标志，并将基树中对应的页标志为脏。

至此，write（）调用就要返回了。那么write（）调用，数据被写到磁盘上去了吗？到现在为止，我们还没看到将数据写到存储上的动作，仅是将待写的数据已用户空间拷贝到页面Cache中（内核空间），然后数据页面标记为脏，但数据数据还未写到磁盘上去。这就是“延迟写”技术。

我们将在Linux内核延迟写机制中详细分析数据写入磁盘过程。