Linux内核延迟写机制

http://www.ilinuxkernel.com

目 录Table of Contents

1 概述 4
2 Linux内核bdi系统 5
2.1 writeback主要数据结构 6
2.2 bdi-default内核线程 9
2.2.1 bdi_register（） 10
2.2.2 bdi_forker_task（） 11
2.3 flush-x:y内核线程 13
2.3.1 bdi_start_fn（） 15
2.3.2 bdi_writeback_task（） 16
3 脏页回写到磁盘 18
3.1 wb_do_writeback（） 18
3.2 wb_writeback（） 19
3.3 writeback_inodes_wb（） 24
3.4 writeback_sb_inodes（） 25
3.5 writeback_single_inode（） 27
3.6 do_writepages（） 31
3.7 ext4文件系统写数据流程 32
3.7.1 ext4_da_writepages（） 32
3.7.2 mpage_da_map_and_submit（） 39
3.7.3 mpage_da_submit_io（） 43
3.7.4 ext4_writepage（） 45
3.8 address space写数据流程 47
3.8.1 block_write_full_page（） 47
3.8.2 block_write_full_page_endio（） 48
3.8.3 __block_write_full_page（） 48
3.9 写结束回调函数end_buffer_async_write（） 53
附录 56

图目录 List of Figures

图1 writeback主要数据结构关系 9
图2 bdi_writeback_task（）流程图.......... 18
图3 ext4文件系统extent基本原理.......... 33
图4 页面page与I/O wait_queue 56

1 概述

在分析sys_write（）的源码过程中，generic_perform_write（）函数执行完后，会逐层返回，直至sys_write（）返回，系统调用结束。但此时要写的数据，只是拷贝到内核缓冲区中，并将相应的页标记为脏；但数据并未真正写到磁盘上。那么何时才会将数据写到磁盘上，又由谁来负责将数据写到磁盘上呢？

由于页高速缓存的缓存作用，写操作实际上会被延迟。当页高速缓存中的数据比后台存储的数据更新时，那么该数据就被称做脏数据。在内存中累积起来的脏页最终必须被写回磁盘。在以下两种情况发生时，脏页被写回磁盘：

- 当空闲内存低于一个特定的阈值时，内核必须将脏页写回磁盘，以便释放内存。
- 当脏页在内存中驻留时间超过一个特定的阈值时，内核必须将超时的脏页写回磁盘，以确保脏页不会无限期地驻留在内存中。

进行间隔性同步工作的进程之前名叫pdflush。原有pdflush机制存在的问题：在多磁盘的系统中，pdflush管理了所有磁盘的page/buffer cache，从而导致一定程度的IO性能瓶颈。自2.6.32内核开始，放弃了原有的pdflush机制，改成了bdi_writeback机制。这种变化主要解决bdi_writeback机制为每个磁盘都创建一个线程，专门负责这个磁盘的page cache或者buffer cache的数据刷新工作，从而实现了每个磁盘的数据刷新程序在线程级的分离，这种处理可以提高IO性能。

在Linux内核Cache机制中，我们知道内核不断用块设备上的数据来填充页面Cache，只要进程修改了数据，相应的页就标记为脏，即设置页面的PG_dirty标志。

Linux允许把脏缓冲区写入块设备的操作延迟执行（write back），这样可以显著提高系统的性能。对页面Cache中的页几次写操作，可能只需要对相应的磁盘块进行一次缓慢的物理更新就可满足。另外，写操作没有读操作那么紧迫，因为进程通常不会因为延迟写而挂起，而大部分情况是因为延迟读阻塞。因为延迟写，使得任一物理块设备在平均情况下，处理读请求远多于写请求。

要写的数据保存在页面Cache中，延迟一段时间写到物理块设备上，这样某个脏页可能

可能直到系统关闭最后一刻，都一直在内存中。因此，从延迟写策略上，主要有两个缺点：

（1） 如发生了硬件错误或电源掉电的情况，那么就无法再获取RAM中的数据。因此，从系统启动以来对文件进行的很多修改就丢失了；

（2） 随着数据不断修改，页面Cache的会不断增大，占用内存。

由上面两个缺点，因此在下面条件下把脏页写入到磁盘：

- 页面Cache变得太满，并且还需要更多的内存页；或者脏页的数量非常大；
- 脏页停留在内存中的时间过长；
- 某个进程要求更改的块设备或某个文件数据刷新；通常调用sync（）、fsync（）或fdatasync（）系统来实现。

缓冲区页的存在，使得情况变得更为复杂。与每个缓冲区页对应的缓冲区头（buffer_head）使内核能够跟踪、记录每个独立块缓冲区的状态。若页面中的某个块缓冲区设置了BH_Dirty标志，那么该页就应设置PG_dirty标志。当内核选择脏的缓冲区页来刷新时，就扫描相应的缓冲区头，并只把脏块的内容写到磁盘。一旦缓冲区页中所有的脏块缓冲区都刷新到磁盘，内核就清除页面的PG_dirty标志。

2 Linux内核bdi系统

bdi是backing device info的缩写，它描述备用存储设备相关信息。bdi就是能够用来存储数据的设备，而这些设备存储的数据能够保证在计算机电源关闭时也不丢失，如硬盘、SSD等。

相对于内存来说，bdi设备（如硬盘）的读写速度是非常慢的。因此为了提高系统整体性能，Linux系统对bdi设备的读写内容进行了缓冲，那些读写的数据会临时保存在内存里，以避免每次都直接操作bdi设备，但这就需要在一定的时机（比如每隔3秒、脏数据达到的一定的比率等）把它们同步到bdi设备，否则持久保存在内存里的数据容易丢失（如机器突然宕机）。

bdi机制中，并且是为每个设备创建了名为bdi-default、flush-x:y的线程，用于脏数据的下刷。x代表主设备好，y代表此设备号。flush-x:y内核进程是对应bdi整个设备的，比如单

个磁盘，而不是各个磁盘分区。

bdi-default和flush-x:y，这两个线程（flush-x:y可能为多个）的关系为父与子的关系，即bdi-default根据当前的状态Create或Destroy flush-x:y，x为块设备类型，y为此类设备的序号。例如有两块硬盘（有两个硬盘盘符），则会有两个flush-8:0、flush-8:1线程。

```
00547: void add_disk(struct gendisk *disk)
00548: {
00549:        struct backing_dev_info *bdi;
00550:        dev_t devt;
00551:        int retval;
               … …
00575:        / * Register BDI before referencing it from bdev */
00576:        bdi = &disk->queue->backing_dev_info;
00577:        bdi_register_dev(bdi, disk_devt(disk));
00578:
00579:        blk_register_region(disk_devt(disk), disk->minors, NULL,
00580:                   exact_match, exact_lock, disk);
00581:        register_disk(disk);
00582:        blk_register_queue(disk);
00583:
               … …
00590:        retval = sysfs_create_link(&disk_to_dev(disk)->kobj,
00591:        &bdi->dev->kobj,"bdi");
00592:        WARN_ON(retval);
00593: } ?  end add_disk ?
00594:
```

通常一个Linux系统会挂载很多bdi设备，注册bdi设备时，会调用add_disk（）向系统添加磁盘设备。在add_disk（）中调用bdi_register_dev(bdi, disk_devt(disk))（577行）这些bdi设备会以链表的形式组织在全局变量bdi_list下。bdi_list全局变量在文件include/linux/backing-dev.h。

```
extern spinlock_t bdi_lock;
extern struct list_head bdi_list;

```

除了一个比较特别的bdi设备以外，它就是default bdi设备（default_backing_dev_info），它除了被加进到bdi_list，还会新建一个bdi-default内核线程。

2.1 writeback主要数据结构

与bdi_writeback机制相关的主要数据结构有：

backing_dev_info：该数据结构描述了backing_dev的所有信息，通常块设备的request queue中会包含backing_dev对象。

bdi_writeback：该数据结构封装了writeback的内核线程以及需要操作的inode队列。

wb_writeback_work：该数据结构封装了writeback的工作任务。

backing_dev_info和bdi_writeback定义在文件include/linux/backing-dev.h。

```
struct backing_dev_info {
    struct list_head bdi_list;
    struct rcu_head rcu_head;
    unsigned long ra_pages;   /* max readahead in PAGE_CACHE_SIZE units */
    unsigned long state;   /* Always use atomic bitops on this */
    unsigned int capabilities; /* Device capabilities */
    congested_fn *congested_fn; /* Function pointer if device is md/dm */
    void *congested_data;   /* Pointer to aux data for congested func */
    void (*unplug_io_fn)(struct backing_dev_info *,
                struct page *);
    void *unplug_io_data;

    char *name;

    struct percpu_counter bdi_stat[NR_BDI_STAT_ITEMS];

    struct prop_local_percpu completions;
    int dirty_exceeded;

    unsigned int min_ratio;
    unsigned int max_ratio, max_prop_frac;

    struct bdi_writeback wb;  /* default writeback info for this bdi */
    spinlock_t wb_lock;   /* protects update side of wb_list */
    struct list_head wb_list; /* the flusher threads hanging off this bdi */

    struct list_head work_list;

    struct device *dev;

#ifdef CONFIG_DEBUG_FS
    struct dentry *debug_dir;
    struct dentry *debug_stats;
#endif
} ?  end backing_dev_info ?  ;

```

bdi_writeback对象封装了内核线程task以及需要处理的inode队列。当page cache/buffer cache需要刷新radix tree上的inode时，可以将该inode挂载到writeback对象的b_dirty队列上，然后唤醒writeback线程。在处理过程中，inode会被移到b_io队列上进行处

理。多条链表的方式可以降低多线程之间的资源共享。writeback数据结构具体定义如下：

```
struct bdi_writeback {
      struct list_head list;          /* hangs off the bdi */

      struct backing_dev_info *bdi;          /* our parent bdi */
      unsigned int nr;

      unsigned long last_old_flush;     /* last old data flush */

      struct task_struct *task;      /* writeback task */
      struct list_head    b_dirty;   /* dirty inodes */
      struct list_head    b_io;      /* parked for writeback */
      struct list_head    b_more_io;     /* parked for more writeback */
};
```

结构体中个成员变量含义：

bdi：和本结构实例相关联的 backing_device_info。

nr：要刷入的页的总数。

las_old_flush：最近刷新数据时间。

task：指向缺省flush线程的指针，这个线程用于启动进行刷写工作的线程。

b_dirty：本bdi上面，需要刷入块设备的所有脏inode。

b_io：等待执行的I/O inode。

b_more_io：更多的要进行I/O的inodes，所有要刷入的 inode 都先被插入到这个队列中来，之后再移送到 b_io。

wb_writeback_work数据结构是对writeback任务的封装，不同的任务可以采用不同的刷新策略。writeback线程的处理对象就是wb_writeback_work。如果writeback_work队列为空，那么内核线程就可以睡眠。wb_writeback_work结构体定义在文件fs/fs-writeback.c中：

```
/*
 * Passed into wb_writeback(), essentially a subset of writeback_control
 */
struct wb_writeback_work {
      long nr_pages;
      struct super_block *sb;
      enum writeback_sync_modes sync_mode;
      int for_kupdate:1;
      int range_cyclic:1;
      int for_background:1;
      struct list_head list;       /* pending work list */
```

```
        struct completion *done; /* set if the caller waits */
};
```

主要成员变量含义如下：

nr_pages：待回写页面数量；

sb: 该 writeback 任务所属的 super_block；

sync_mode：指定同步模式，WB_SYNC_ALL 表示当遇到锁住的 inode 时，它必须等待该 inode 解锁，而不能跳过。WB_SYNC_NONE 表示跳过被锁住的 inode；

for_kupdated: 若值为 1，则表示回写操作是周期性的机制；否则值为 0；

range_cyclic: 若值为 0，则表示回写操作范围限制在[range_start, range_end]限定范围；若值为 1，则表示内核可以对 mapping 里的页面执行多次回写操作。

for_background: 若值为 1，表示后台回写；否则值为 0；

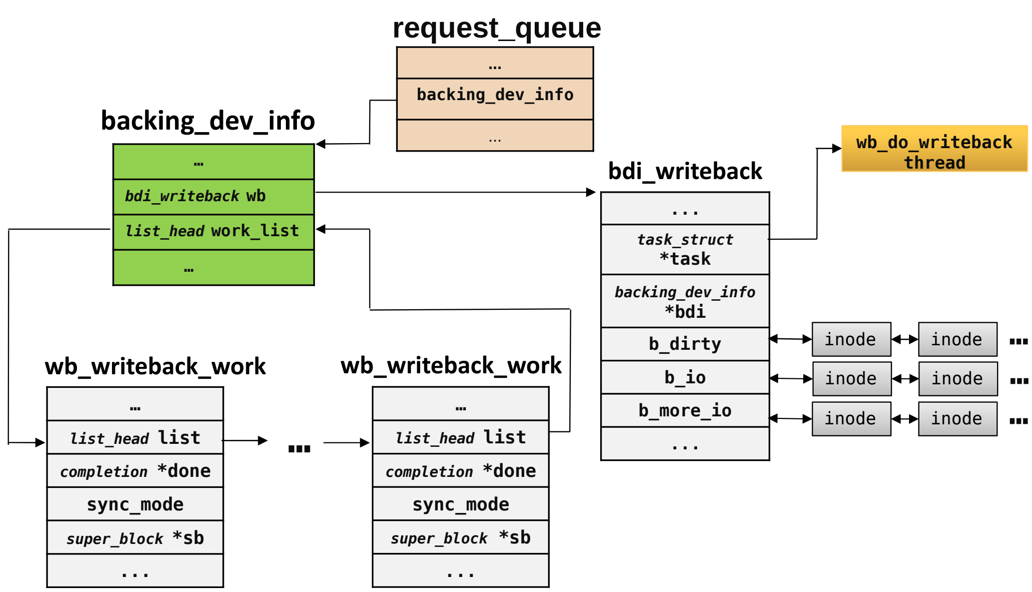

图1 writeback主要数据结构关系

2.2 bdi-default内核线程

```
static int __init default_bdi_init(void)
{
        int err;

        sync_supers_tsk = kthread_run(bdi_sync_supers, NULL,
                                      "sync_supers");
        BUG_ON(IS_ERR(sync_supers_tsk));

```

```
00241:     init_timer(&sync_supers_timer);
00242:     setup_timer(&sync_supers_timer,
               sync_supers_timer_fn, 0);
00243:     bdi_arm_supers_timer();
00244:
00245:     err = bdi_init(&default_backing_dev_info);
00246:     if (!err)
00247:         bdi_register(&default_backing_dev_info, NULL,
                   "default");
00248:
00249:     return err;
00250: }
00251: subsys_initcall(default_bdi_init);
```

Linux内核启动时，会执行bdi模块default_bdi_init（），代码定义在文件mm/backing-dev.c中。主要工作如下：

- 创建名为sync_supers的线程，此线程由定时器来唤醒，此外无其他唤醒模式（238行）。每间隔dirty_writeback_interval * 10（即5秒钟）唤醒执行一次函数sync_supers，用来下刷系统super_blocks链表中所有的元数据块信息。dirty_writeback_interval的值可以通过/proc/sys/vm/dirty_writeback_centisecs来修改，默认值是500，单位是10ms。定时器函数sync_supers_timer_fn用于唤醒同步超级块的线程sync_supers_tsk，并且更新定时器的到期时间（241～242行）。

- 定义默认数据结构default_backing_dev_info，同时会创建一个线程bdi-default，此线程执行函数为bdi_forker_task（）。

- 当bdi_pending_list链表为空时会睡眠5s，然后再自我唤醒，继续运行。

2.2.1 bdi_register（）

bdi_register（）函数定义见文件mm/backing-dev.c。创建一个内核线程，来执行函数bdi_forker_ task（）（563行）。

```
00531: int bdi_register(struct backing_dev_info *bdi, struct device
00532:         *parent,const char *fmt, ...)
00533: {
00534:     va_list args;
00535:     int ret = 0;
00536:     struct device *dev;
00537:
```

```
    if (bdi->dev)   /* The driver needs to use separate queues per device */
        goto ↓exit;
        … …

        wb->task = kthread_run(bdi_forker_task, wb, "bdi-%s",
                    dev_name(dev));
        if (IS_ERR(wb->task)) {
            wb->task = NULL;
            ret = -ENOMEM;

            bdi_remove_from_list(bdi);
            goto ↓exit;
        }
        … …
exit:
    return ret;
} ?  end bdi_register ?
```

2.2.2 bdi_forker_task（）

bdi_forker_task（）函数定义见文件mm/backing-dev.c。bdi_forker_task（）函数函数为一个死循环，中途无任何退出条件。

```
static int bdi_forker_task(void *ptr)
{
    struct bdi_writeback *me = ptr;

    bdi_task_init(me->bdi, me);

    for (;;) {
        struct backing_dev_info *bdi, *tmp;
        struct bdi_writeback *wb;

        /*
         * Temporary measure, we want to make sure we don't see
         * dirty data on the default backing_dev_info
         */
        if (wb_has_dirty_io(me) || !
            list_empty(&me->bdi->work_list))
            wb_do_writeback(me, 0);

        spin_lock_bh(&bdi_lock);

        /*
         * Check if any existing bdi's have dirty data without
         * a thread registered. If so, set that up.
         */
        list_for_each_entry_safe(bdi, tmp, &bdi_list, bdi_list)
        {
            if (bdi->wb.task)
```

```
                continue;
            if (list_empty(&bdi->work_list) &&
                !bdi_has_dirty_io(bdi))
                continue;

            bdi_add_default_flusher_task(bdi);
        }

        set_current_state(TASK_INTERRUPTIBLE);

        if (list_empty(&bdi_pending_list)) {
            unsigned long wait;

            spin_unlock_bh(&bdi_lock);
            wait = msecs_to_jiffies(dirty_writeback_interval
                                    * 10);
            if (wait)
                schedule_timeout(wait);
            else
                schedule();
            try_to_freeze();
            continue;
        }

        __set_current_state(TASK_RUNNING);

        /*
         * This is our real job - check for pending entries in
         * bdi_pending_list, and create the tasks that got added
         */
        bdi = list_entry(bdi_pending_list.next, struct
                    backing_dev_info,bdi_list);
        list_del_init(&bdi->bdi_list);
        spin_unlock_bh(&bdi_lock);

        wb = &bdi->wb;
        wb->task = kthread_run(bdi_start_fn, wb, "flush-%s",
                    dev_name(bdi->dev));
        /*
         * If task creation fails, then readd the bdi to
         * the pending list and force writeout of the bdi
         * from this forker thread. That will free some memory
         * and we can try again.
         */
        if (IS_ERR(wb->task)) {
            wb->task = NULL;

            /*
             * Add this 'bdi' to the back, so we get
             * a chance to flush other bdi's to free
             * memory.
             */
            spin_lock_bh(&bdi_lock);
```

```
            list_add_tail(&bdi->bdi_list, &bdi_pending_list);
            spin_unlock_bh(&bdi_lock);

            bdi_flush_io(bdi);
        }
    } ?  end for ;; ?

    return 0;
} ?  end bdi_forker_task ?
```

当有新的设备被创建时：

1、为每个设备定义一个结构backing_dev_info，然后将此结构挂到bdi_list链表尾；

2、bdi-default线程会从bdi_list链表中获取每个设备的结构bdi信息，将其从bdi_list链表中删除，再加入bdi_pending_list；

3、从bdi_pending_list链表中获取每个设备的bdi信息，同时将此设备的bdi信息从bdi_pending_list链表中删除；

4、为每个设备创建一个下刷线程“flush-主设备号:次设备号”，描述线程信息的结构为bdi_writeback，其执行函数为**bdi_start_fn**；

5、当flush线程执行起来后，会再次将设备的bdi信息加入链表bdi_list尾，这样每个设备的flush线程就可以从bdi_list上找到并唤醒了。

2.3 flush-x:y内核线程

flush-x:y内核线程有两个作用：

（1）系统地扫描页面Cache，以找到要刷新的脏页；

（2）保证所有的页不会长时间处于“脏”状态。

分析flush-x:y内核线程前，我们先看一下基于硬盘设备（控制器为LSI 2308，驱动为mpt2sas）的内核线程调用栈信息。

```
Pid: 1222, comm: flush-8:0 Tainted: G           ----------- HT 2.6.32279.debug #28
Call Trace:
 [<ffffffffa0019abb>] ? mpt2sas_base_get_smid_scsiio+0x6b/0xb0 [mpt2sas]
 [<ffffffffa001a6d7>] ? mpt2sas_base_get_msg_frame+0x57/0x60 [mpt2sas]
 [<ffffffffa00246fe>] ? _scsih_qcmd+0x17e/0x9b0 [mpt2sas]
 [<ffffffff81253de3>] ? ftrace_raw_event_id_block_rq+0x153/0x190
 [<ffffffff81363591>] ? scsi_dispatch_cmd+0x101/0x360
 [<ffffffff8136b08d>] ? scsi_request_fn+0x41d/0x790
```

```
[<ffffffff8107c981>] ? ftrace_raw_event_timer_cancel+0xa1/0xb0
[<ffffffff81255601>] ? __blk_run_queue+0x31/0x40
[<ffffffff8126e4f9>] ? cfq_insert_request+0x469/0x5b0
[<ffffffff8124f6d1>] ? elv_insert+0xd1/0x1a0
[<ffffffff8124f7ea>] ? __elv_add_request+0x4a/0x90
[<ffffffff81258903>] ? __make_request+0x103/0x5a0
[<ffffffff81254512>] ? ftrace_raw_event_id_block_bio+0xf2/0x100
[<ffffffff81256efe>] ? generic_make_request+0x25e/0x530
[<ffffffff8125728c>] ? submit_bio+0xbc/0x160
[<ffffffff811acd46>] ? submit_bh+0xf6/0x150
[<ffffffffa01097a3>] ? ext4_mb_init_cache+0x883/0x9f0 [ext4]
[<ffffffff8112b560>] ? __lru_cache_add+0x40/0x90
[<ffffffffa0109a2e>] ? ext4_mb_init_group+0x11e/0x210 [ext4]
[<ffffffffa0109bed>] ? ext4_mb_good_group+0xcd/0x110 [ext4]
[<ffffffffa010b38b>] ? ext4_mb_regular_allocator+0x19b/0x410 [ext4]
[<ffffffffa010d25d>] ? ext4_mb_new_blocks+0x38d/0x560 [ext4]
[<ffffffffa0100afe>] ? ext4_ext_find_extent+0x2be/0x320 [ext4]
[<ffffffffa0103bb3>] ? ext4_ext_get_blocks+0x1113/0x1a10 [ext4]
[<ffffffff810edb54>] ? rb_reserve_next_event+0xb4/0x370
[<ffffffff810edfc2>] ? ring_buffer_lock_reserve+0xa2/0x160
[<ffffffffa00dfd79>] ? ext4_get_blocks+0xf9/0x2a0 [ext4]
[<ffffffff81012bd9>] ? read_tsc+0x9/0x20
[<ffffffff8109cd39>] ? ktime_get_ts+0xa9/0xe0
[<ffffffffa00e1c21>] ? mpage_da_map_and_submit+0xa1/0x450 [ext4]
[<ffffffff81277ef5>] ? radix_tree_gang_lookup_tag_slot+0x95/0xe0
[<ffffffff81113bd0>] ? find_get_pages_tag+0x40/0x120
[<ffffffffa00e203d>] ? mpage_add_bh_to_extent+0x6d/0xf0 [ext4]
[<ffffffffa00e238f>] ? write_cache_pages_da+0x2cf/0x470 [ext4]
[<ffffffffa00e2802>] ? ext4_da_writepages+0x2d2/0x620 [ext4]
[<ffffffff811299e1>] ? do_writepages+0x21/0x40
[<ffffffff811a500d>] ? writeback_single_inode+0xdd/0x2c0
[<ffffffff811a544e>] ? writeback_sb_inodes+0xce/0x180
[<ffffffff811a55ab>] ? writeback_inodes_wb+0xab/0x1b0
[<ffffffff811a594b>] ? wb_writeback+0x29b/0x3f0
[<ffffffff814fd9b0>] ? thread_return+0x4e/0x76e
[<ffffffff8107eb42>] ? del_timer_sync+0x22/0x30
[<ffffffff811a5c39>] ? wb_do_writeback+0x199/0x240
[<ffffffff811a5d43>] ? bdi_writeback_task+0x63/0x1b0
[<ffffffff81091f97>] ? bit_waitqueue+0x17/0xd0
[<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
[<ffffffff811386c6>] ? bdi_start_fn+0x86/0x100
[<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
[<ffffffff81091d66>] ? kthread+0x96/0xa0
[<ffffffff8100c14a>] ? child_rip+0xa/0x20s
```

```
[<ffffffff81091cd0>] ? kthread+0x0/0xa0
[<ffffffff8100c140>] ? child_rip+0x0/0x20
```

从函数栈信息，我们可以看出flush-x:y内核线程执行流程为 bdi_start_fn（）--> bdi_writeback_task（）--> wb_do_writeback（） -->wb_writeback（）--> writeback_inodes_wb（）--> writeback_sb_inodes（） --> writeback_single_inode（）--> do_writepages（） --> ext4_da_writepages（） --> ...

2.3.1 bdi_start_fn（）

在bdi_forker_task（）函数分析过程中，我们知道flush-x:y内核线程的执行函数是bdi_start_fn（）。

```
static int bdi_start_fn(void *ptr)
{
	struct bdi_writeback *wb = ptr;
	struct backing_dev_info *bdi = wb->bdi;
	int ret;

	/*
	 * Add us to the active bdi_list
	 */
	spin_lock_bh(&bdi_lock);
	list_add_rcu(&bdi->bdi_list, &bdi_list);
	spin_unlock_bh(&bdi_lock);

	bdi_task_init(bdi, wb);

	/*
	 * Clear pending bit and wakeup anybody waiting to tear us down
	 */
	clear_bit(BDI_pending, &bdi->state);
	smp_mb__after_clear_bit();
	wake_up_bit(&bdi->state, BDI_pending);

	ret = bdi_writeback_task(wb);

	/*
	 * Remove us from the list
	 */
	spin_lock(&bdi->wb_lock);
	list_del_rcu(&wb->list);
	spin_unlock(&bdi->wb_lock);

	/*
	 * Flush any work that raced with us exiting. No new work
	* will be added, since this bdi isn't discoverable anymore.
```

```
     */
    if (!list_empty(&bdi->work_list))
        wb_do_writeback(wb, 1);

    wb->task = NULL;
    return ret;
} ?  end bdi_start_fn ?
```

bdi_start_fn（）主要是对函数bdi_writeback_task（）的封装，其实现在文件fs/fs-writeback.c中。

2.3.2 bdi_writeback_task（）

```
/*
 * Handle writeback of dirty data for the device backed by this bdi. Also
 * wakes up periodically and does kupdated style flushing.
 */
int bdi_writeback_task(struct bdi_writeback *wb)
{
    unsigned long last_active = jiffies;
    unsigned long wait_jiffies = -1UL;
    long pages_written;

    trace_writeback_task_start(wb->bdi);

    while (!kthread_should_stop()) {
        pages_written = wb_do_writeback(wb, 0);

        trace_writeback_pages_written(pages_written);

        if (pages_written)
            last_active = jiffies;
        else if (wait_jiffies != -1UL) {
            unsigned long max_idle;

            /*
             * Longest period of inactivity that we tolerate. If we
             * see dirty data again later, the task will get
             * recreated automatically.
             */
            max_idle = max(5UL * 60 * HZ, wait_jiffies);
            if (time_after(jiffies, max_idle + last_active))
                break;
        }

        if (dirty_writeback_interval) {
            wait_jiffies =
             msecs_to_jiffies(dirty_writeback_interval * 10);
            schedule_timeout_interruptible(wait_jiffies);
        } else
            schedule();
```

```
00827:
00828:         try_to_freeze();
00829:     } ?  end while ! kthread_should_stop() ?
00830:
00831:     trace_writeback_task_stop(wb->bdi);
00832:
00833:     return 0;
00834: } ?  end bdi_writeback_task ?
```

函数主体是一个while循环，while语句调用kthread_should_stop（）决定是否该结束循环（802行），内核线程flush-x:y创建后，就会一直存在，直到对应的块设备停止运行。

```
/ **
  * kthread_should_stop - should this kthread return now?
  *
  * When someone calls kthread_stop() on your kthread, it will be woken
  * and this will return true.  You should then return, and your return
  * value will be passed through to kthread_stop().
  */
int kthread_should_stop(void)
{
    return to_kthread(current)->should_stop;
}
 EXPORT_SYMBOL(kthread_should_stop);

```

修改最近活动时间为当前时间（796行）。

若在进行一次同步操作之后，最近writeback时间闲超过max_idle（817～819行），则要重新执行一次脏页刷新动作wb_do_writeback（）。若两次同步操作时间间隔没超过max_idle，那么先睡眠，等间隔时间（默认5秒）后超时醒来继续工作（822～824行）。

从代码中，可以看出执行脏页刷新动作的函数是wb_do_writeback（）。接下来，我们继续分析。

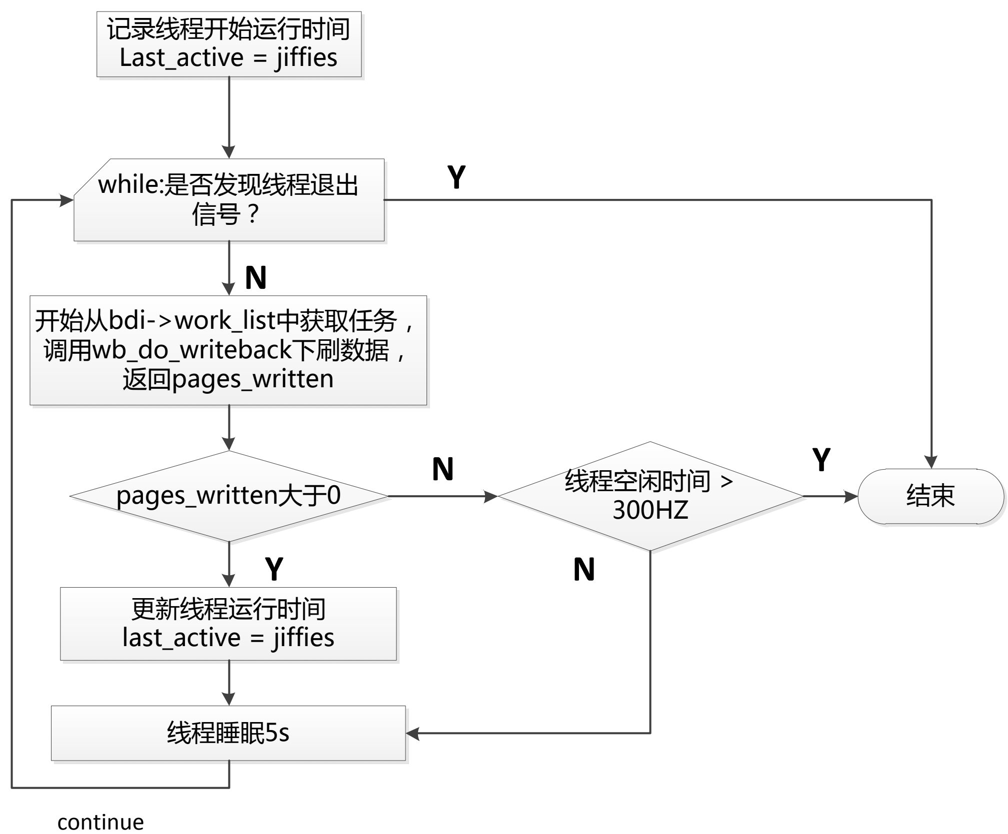

图2 bdi_writeback_task（）流程图

3 脏页回写到磁盘

前面我们分析了Linux内核bdi相关数据结构、bdi-default、flush-x:y内核线程。接下来，我们继续分析沿着flush-x:y内核线程中bdi_writeback_task（）函数回写脏数据过程。

3.1 wb_do_writeback（）

wb_do_writeback（）函数在文件fs/fs-writeback.c中。

从bdi_writeback_task（）中调用pages_written = wb_do_writeback(wb, 0)看到，并没有要求刷新所有脏数据，即不设置work->sync_mode为WB_SYNC_ALL。

```
/ *
 * Retrieve work items and do the writeback they describe
 */
```

```
00753: long wb_do_writeback(struct bdi_writeback *wb, int
                                        force_wait)
00754: {
00755:     struct backing_dev_info *bdi = wb->bdi;
00756:     struct wb_writeback_work *work;
00757:     long wrote = 0;
00758:
00759:     while ((work = get_next_work_item(bdi, wb)) != NULL) {
00760:         /*
00761:          * Override sync mode, in case we must wait for completion
00762:          * because this thread is exiting now.
00763:          */
00764:         if (force_wait)
00765:             work->sync_mode = WB_SYNC_ALL;
00766:
00767:         trace_writeback_exec(bdi, work);
00768:
00769:         wrote += wb_writeback(wb, work);
00770:
00771:         /*
00772:          * Notify the caller of completion if this is a synchronous
00773:          * work item, otherwise just free it.
00774:          */
00775:         if (work->done)
00776:             complete(work->done);
00777:         else
00778:             kfree(work);
00779:     } ?  end while (work=get_next_work_i... ?
00780:     trace_mm_background_writeout(wrote);
00781:
00782:     /*
00783:     * Check for periodic writeback, kupdated() style
00784:      */
00785:     wrote += wb_check_old_data_flush(wb);
00786:
00787:     return wrote;
00788: } ?  end wb_do_writeback ?
00789:
```

函数主体函数动作比较简单，就是遍历所有wb_writeback_work，对其执行wb_writeback（）（759～779行）。对队列上所有的work执行回写操作后，会再次检查是否还有脏页待回写（785行）。

3.2 wb_writeback（）

wb_writeback（）函数主要依赖于数据结构writeback_control，其定义在include/linux/writeback.h中。数据结构有双向通信作用：一方面它告诉辅助函数

writeback_inodes（）做什么，另一方面，它保存回写到磁盘上的页面统计信息。

```
/*
 * A control structure which tells the writeback code what to do.  These are
 * always on the stack, and hence need no locking.  They are always initialised
 * in a manner such that unspecified fields are set to zero.
 */
struct writeback_control {
    enum writeback_sync_modes sync_mode;
    unsigned long *older_than_this;   /* If !NULL, only write back
                inodes older than this */
    unsigned long wb_start;     /* Time writeback_inodes_wb was
                        called. This is needed to avoid
                        extra jobs and livelock */
    long nr_to_write;       /* Write this many pages, and decrement
                        this for each page written */
    long pages_skipped;     /* Pages which were not written */

    /*
     * For a_ops->writepages(): is start or end are non-zero then this is
     * a hint that the filesystem need only write out the pages inside that
     * byterange.  The byte at `end' is included in the writeout request.
     */
    loff_t range_start;
    loff_t range_end;

    unsigned nonblocking:1;     /* Don't get stuck on request queues */
    unsigned encountered_congestion:1; /* An output: a queue is full
*/
    unsigned for_kupdate:1;     /* A kupdate writeback */
    unsigned for_background:1;      /* A background writeback */
    unsigned for_reclaim:1;     /* Invoked from the page allocator */
    unsigned range_cyclic:1;/* range_start is cyclic */
    unsigned more_io:1;     /* more io to be dispatched */

    /* reserved for Red Hat */
    unsigned long rh_reserved[5];
} ?  end writeback_control ?  ;
```

writeback_control结构中重要的成员变量如下：

sync_mode：指定同步模式，WB_SYNC_ALL表示当遇到锁住的inode时，它必须等待该inode解锁，而不能跳过。WB_SYNC_NONE表示跳过被锁住的inode；

older_than_this：若不为NULL，则表示仅回写比指定时间晚的inode；

wb_start：回写操作开始执行时间；

nr_to_write：要回写的脏页数量；

pages_skipped：跳过回写的页面数量；

nonblocking：若该标志被设置，就表示该进程不能被阻塞；

for_kupdated: 若值为1，则表示回写操作是周期性的机制；否则值为0；

for_background: 若值为1，表示后台回写；否则值为0；

range_cyclic: 若值为0，则表示回写操作范围限制在[range_start, range_end]限定范围；若值为1，则表示内核可以对mapping里的页面执行多次回写操作。

```
/*
 * Explicit flushing or periodic writeback of "old" data.
 *
 * Define "old": the first time one of an inode's pages is dirtied, we mark the
 * dirtying-time in the inode's address_space.  So this periodic writeback code
 * just walks the superblock inode list, writing back any inodes which are
 * older than a specific point in time.
 *
 * Try to run once per dirty_writeback_interval.  But if a writeback event
 * takes longer than a dirty_writeback_interval interval, then leave a
 * one-second gap.
 *
 * older_than_this takes precedence over nr_to_write.  So we'll only write back
 * all dirty pages if they are all attached to "old" mappings.
 */
static long wb_writeback(struct bdi_writeback *wb,
                struct wb_writeback_work *work)
{
    struct writeback_control wbc = {
        .sync_mode          = work->sync_mode,
        .older_than_this    = NULL,
        .for_kupdate        = work->for_kupdate,
        .for_background         = work->for_background,
        .range_cyclic       = work->range_cyclic,
    };
    unsigned long oldest_jif;
    long wrote = 0;
    struct inode *inode;

    if (wbc.for_kupdate) {
        wbc.older_than_this = &oldest_jif;
        oldest_jif = jiffies -
                msecs_to_jiffies(dirty_expire_interval * 10);
    }
    if (!wbc.range_cyclic) {
        wbc.range_start = 0;
        wbc.range_end = LLONG_MAX;
    }

    wbc.wb_start = jiffies; /* livelock avoidance */
```

若当前回写数据动作是周期性的调用，就要记录最晚的时间点为：[当前时间jiffies -

dirty_expire_interval * 10（624～628行）。

若不是回写指定范围的脏页，那么就设置回写所有页面[0~LLONG_MAX]（629～632行）。

记录回写开始时间到wb_start中（634行）。

```
00635:     for (;;) {
00636:         /*
00637:          * Stop writeback when nr_pages has been consumed
00638:          */
00639:         if (work->nr_pages <= 0)
00640:             break;
00641:
00642:         /*
00643:          * For background writeout, stop when we are below the
00644:          * background dirty threshold
00645:          */
00646:         if (work->for_background && !over_bground_thresh())
00647:             break;
00648:
00649:         wbc.more_io = 0;
00650:         wbc.nr_to_write = MAX_WRITEBACK_PAGES;
00651:         wbc.pages_skipped = 0;
00652:
```

首先判断当前回写任务中，nr_page是否大于0（639行），若不大于0，说明没有脏页面需要回写，结束执行。接着检查是否后台回写，且没有超过一定脏页比例（646行）；脏页超过一定比例，就需要执行回写动作。脏数据比例阀值可以通过修改/proc/sys/vm/dirty_background_ratio来修改。

设置不需要回写更多的IO，回写数据的页面数量nr_to_write为MAX_WRITEBACK_PAGES，不跳过部分页面（649～651行）。

```
00653:         trace_wbc_writeback_start(&wbc, wb->bdi);
00654:         if (work->sb)
00655:             __writeback_inodes_sb(work->sb, wb, &wbc);
00656:         else
00657:             writeback_inodes_wb(wb, &wbc);
00658:         trace_wbc_writeback_written(&wbc, wb->bdi);
00659:         work->nr_pages -= MAX_WRITEBACK_PAGES -
                                   wbc.nr_to_write;
00660:         wrote += MAX_WRITEBACK_PAGES - wbc.nr_to_write;
00661:
```

若定义了回写任务中的 work->sb，则表示只回写该 superblock 下面的脏 inodes（654～655 行）。不管是否回写指定 superblock 下面的 inodes，最终都会调用 writeback_inodes_wb

（）来执行写操作。后面会详细分析这个函数。

```
00662:         /*
00663:          * If we consumed everything, see if we have more
00664:          */
00665:         if (wbc.nr_to_write <= 0)
00666:             continue;
00667:         /*
00668:         * Didn't write everything and we don't have more IO, bail
00669:          */
00670:         if (!wbc.more_io)
00671:             break;
00672:         /*
00673:          * Did we write something? Try for more
00674:          */
00675:         if (wbc.nr_to_write < MAX_WRITEBACK_PAGES)
00676:             continue;
```

回写操作完成后，若wbc.nr_to_write小于等于0或小于MAX_WRITEBACK_PAGES，说明仍有脏页面需要回写（665～666行、675～676行）。

若没有更多的IO操作需要执行，就退出（670～671行）。

```
00677:         /*
00678:          * Nothing written. Wait for some inode to
00679:          * become available for writeback. Otherwise
00680:          * we'll just busyloop.
00681:          */
00682:         spin_lock(&inode_lock);
00683:         if (!list_empty(&wb->b_more_io)) {
00684:             inode = list_entry(wb->b_more_io.prev,
00685:                         struct inode, i_list);
00686:             trace_wbc_writeback_wait(&wbc, wb->bdi);
00687:             inode_wait_for_writeback(inode);
00688:         }
00689:         spin_unlock(&inode_lock);
00690:     } ?  end for ;; ?
00691:
00692:     return wrote;
00693: } ?  end wb_writeback ?
00694:
```

执行到现在，仍没有成功回写数据的话，就要一直等待inode变成可回写状态，然后重复上面的动作（682～690行），当inode设置了__I_SYNC标志时，就表明该inode可回写。

最后返回成功回写的页面数量（692行）。

3.3 writeback_inodes_wb（）

writeback_inodes_wb（）函数定义见文件fs/fs-writeback.c。

```
00534: void writeback_inodes_wb(struct bdi_writeback *wb,
00535:         struct writeback_control *wbc)
00536: {
00537:     int ret = 0;
00538:
00539:     if (!wbc->wb_start)
00540:         wbc->wb_start = jiffies; /* livelock avoidance */
00541:     spin_lock(&inode_lock);
00542:     if (!wbc->for_kupdate || list_empty(&wb->b_io))
00543:         queue_io(wb, wbc->older_than_this);
00544:
```

重新确认回写开始时间（539～540行）。然后将所有比当前wb->b_dirty队列上时间更早的inode全部移到即将回写wb->b_io队列上（542～543行）。

```
00545:     while (!list_empty(&wb->b_io)) {
00546:         struct inode *inode = list_entry(wb->b_io.prev,
00547:                         struct inode, i_list);
00548:         struct super_block *sb = inode->i_sb;
00549:
00550:         if (!pin_sb_for_writeback(sb)) {
00551:             requeue_io(inode);
00552:             continue;
00553:         }
00554:         ret = writeback_sb_inodes(sb, wb, wbc, false);
00555:         drop_super(sb);
00556:
00557:         if (ret)
00558:             break;
00559:     }
00560:     spin_unlock(&inode_lock);
00561:     /* Leave any unwritten inodes on b_io */
00562: } ?  end writeback_inodes_wb ?
```

在后台执行writeback时，并没有设置inode依附于superblock。因此这里我们要确认当前要回写的inode与superblock是否建立附属关系，以防止回写inode过程中，superblock消失（如文件系统卸载）。这就是pin_sb_for_writeback（）函数的功能。

545～558行，对待回写wb->b_io脏inode队列上所有节点进行以下操作：

1、若当前inode还没与相应的superblock建立附属关系，则将次inode移到移到wb->b_more_io队列中，跳过这个inode（550～553行），下次再刷新此inode上的页面。

2、调用writeback_sb_inodes（）来回写inodes上的脏页面。

3.4 writeback_sb_inodes（）

writeback_sb_inodes（）函数定义见文件fs/fs-writeback.c。

函数依次从wb->b_io链表中取出脏inode，判断inode是否需要回写。如需要回写，则调用writeback_single_inode（）完成回写；否则，将其添加到某个链表中或返回。

```
/ *
 * Write a portion of b_io inodes which belong to @sb.
 *
 * If @only_this_sb is true, then find and write all such
 * inodes. Otherwise write only ones which go sequentially
 * in reverse order.
 *
 * Return 1, if the caller writeback routine should be
 * interrupted. Otherwise return 0.
 */
static int writeback_sb_inodes(struct super_block *sb,
            struct bdi_writeback *wb,
            struct writeback_control *wbc, bool only_this_sb)
{
    while (!list_empty(&wb->b_io)) {
        long pages_skipped;
        struct inode *inode = list_entry(wb->b_io.prev,
                        struct inode, i_list);

        if (inode->i_sb != sb) {
            if (only_this_sb) {
                / *
                 * We only want to write back data for this
                 * superblock, move all inodes not belonging
                 * to it back onto the dirty list.
                 */
                redirty_tail(inode);
                continue;
            }

            / *
             * The inode belongs to a different superblock.
             * Bounce back to the caller to unpin this and
             * pin the next superblock.
             */
            return 0;
        }

```

首先检查 bdi_writeback 任务上的 inode 所属的 superblock 是否与传递进来的 superblock

一致。若参数中指定必须回写属于某个文件系统的脏 inode，那么通过 redirty_tail（）将该 inode 重新弄脏，redirty_tail（）会修改 inode 弄脏的时间并将其添加到 dirty 链表的头部，然后继续下一次循环（478～487 行）。

若不一致，说明该 inode 属于另外一个 superblock。若参数中并未指定一定回写属于某个文件系统（superblock）的脏 inode，那么直接向调用者返回 0（494 行）。

```
00497:          if (inode->i_state & (I_NEW | I_WILL_FREE)) {
00498:              requeue_io(inode);
00499:              continue;
00500:          }
00501:          / *
00502:           * Was this inode dirtied after sync_sb_inodes was called?
00503:           * This keeps sync from extra jobs and livelock.
00504:           */
00505:          if (inode_dirtied_after(inode, wbc->wb_start))
00506:              return 1;
00507:
00508:          BUG_ON(inode->i_state & (I_FREEING | I_CLEAR));
00509:          __iget(inode);
```

若 inode 的状态为 I_NEW 或者 I_WILL_FREE，表明该 inode 当前是不需回写（新的或即将释放的 inode），调用 requeue_io（）将该 inode 添加到 more_io 链表中（497～499 行）。

inode_dirtied_after（）判断该 inode 变为脏的时间位于本次 writeback 开始之后。若这个 inode 在本次 writeback 动作开始之后变脏的，说明时间很短，就不回写，直接返回（505～506 行）。

排除上面不需要回写的 inode 之后，接下来就要调用 writeback_single_inode（）来回写 inode 上的脏页面。首先要增加 inode 计数（509 行）。

```
00510:          pages_skipped = wbc->pages_skipped;
00511:          writeback_single_inode(inode, wbc);
00512:          if (wbc->pages_skipped != pages_skipped) {
00513:              / *
00514:               * writeback is not making progress due to locked
00515:               * buffers.  Skip this inode for now.
00516:               */
00517:              redirty_tail(inode);
00518:          }
00519:          spin_unlock(&inode_lock);
00520:          iput(inode);
00521:          cond_resched();
00522:          spin_lock(&inode_lock);
00523:          if (wbc->nr_to_write <= 0) {
00524:              wbc->more_io = 1;
```

```
00525:                 return 1;
00526:             }
00527:             if (!list_empty(&wb->b_more_io))
00528:                 wbc->more_io = 1;
00529:         } ?   end while ! list_empty(&wb- >b_io) ?
00530:         / * b_io is empty */
00531:         return 1;
00532: } ?   end writeback_sb_inodes ?
```

回写过程中可能忽略了某些页面，比如某个页面正被locked无法立即回写，回写就要跳过这个页面，pages_skipped记录跳过回写的页面数量。若跳过了一些页面，那么必须重新要将该inode弄脏且添加到wb->b_dirty链表（512～518行）。

writeback_single_inode（）完成后，就要判断设定的回写页面数量是否已全部完成，如果是，那么将wbc->more_io设置为1，并向调用者返回1，表明本次回写可结束（523～525行）。若尚未全部完成，那么必须得进行下一次循环，在重新循环之前还要判断more_io链表是否为空，如果不为空，设置wbc->more_io=1（527～528行）。

3.5 writeback_single_inode（）

writeback_single_inode（）函数主要任务就是将某个inode下的脏页回写到存储上，代码实现在fs/fs-writeback.c文件中。

```
/ *
 * Write out an inode's dirty pages.  Called under inode_lock.  Either the
 * caller has ref on the inode (either via ____iget or via syscall against an fd)
 * or the inode has I_WILL_FREE set (via generic_forget_inode)
 *
 * If `wait' is set, wait on the writeout.
 *
 * The whole writeout design is quite complex and fragile.  We want to avoid
 * starvation of particular inodes when others are being redirtied, prevent
 * livelocks, etc.
 *
 * Called under inode_lock.
 */
static int
writeback_single_inode(struct inode *inode, struct
writeback_control *
wbc)
{
    struct address_space *mapping = inode->i_mapping;
    unsigned dirty;
    int ret;

```

```
00305:    if (!atomic_read(&inode->i_count))
00306:        WARN_ON(!(inode->i_state &
                  (I_WILL_FREE|I_FREEING)));
00307:    else
00308:        WARN_ON(inode->i_state & I_WILL_FREE);
00309:
00310:    if (inode->i_state & I_SYNC) {
00311:        /*
00312:         * If this inode is locked for writeback and we are not doing
00313:         * writeback-for-data-integrity, move it to b_more_io so that
00314:         * writeback can proceed with the other inodes on s_io.
00315:         *
00316:         * We'll have another go at writing back this inode when we
00317:         * completed a full scan of b_io.
00318:         */
00319:        if (wbc->sync_mode != WB_SYNC_ALL) {
00320:            requeue_io(inode);
00321:            return 0;
00322:        }
00323:
00324:        /*
00325:         * It's a data-integrity sync.  We must wait.
00326:         */
00327:        inode_wait_for_writeback(inode);
00328:    }
00329:
00330:    BUG_ON(inode->i_state & I_SYNC);
00331:
```

I_SYNC是inode增加的一种状态，功能类似于I_LOCK，但仅用于writeback inode脏数据。若当前inode的状态为I_SYNC，且同步模式不为WB_SYNC_ALL，表示这个inode需要重新移到sb->b_more_io队列中，将来会处理b_more_io队列（319～322行）。

若当前inode的状态为I_SYNC，且同步模式是WB_SYNC_ALL，就表示一定要完整同步完成回写该inode上的所有脏页（327行）。

再次检查inode状态，若包含I_SYNC标志，说明系统出了Bug（330行）。

```
00332:    /* Set I_SYNC, reset I_DIRTY */
00333:    dirty = inode->i_state & I_DIRTY;
00334:    inode->i_state |= I_SYNC;
00335:    inode->i_state &= ~I_DIRTY;
00336:
00337:    spin_unlock(&inode_lock);
00338:
00339:    ret = do_writepages(mapping, wbc);
00340:
```

接下来设置inode为I_SYNC状态，并清除inode脏标志（334～335行）。接下来调用do_writepages（）来回写脏页（339行）。

```
00341:      /*
00342:       * Make sure to wait on the data before writing out the metadata.
00343:       * This is important for filesystems that modify metadata on data
00344:       * I/O completion.
00345:       */
00346:      if (wbc->sync_mode == WB_SYNC_ALL) {
00347:          int err = filemap_fdatawait(mapping);
00348:          if (ret == 0)
00349:              ret = err;
00350:      }
00351:
00352:      /* Don't write the inode if only I_DIRTY_PAGES was set */
00353:      if (dirty & (I_DIRTY_SYNC | I_DIRTY_DATASYNC)) {
00354:          int err = write_inode(inode, wbc);
00355:          if (ret == 0)
00356:              ret = err;
00357:      }
00358:
```

若同步模式是WB_SYNC_ALL，就要保证所有脏数据回写成功后，再更改文件系统的metadata（346～350行）。

I_DIRTY_SYNC标志含义为inode处于脏状态，但调用fdatasync时，不要求必须回写，inode最后访问时间（i_atime）是设置I_DIRTY_SYNC最常见原因。I_DIRTY_DATASYNC表示inode数据修改导致处于脏状态。两个标志用于区分是访问inode还是修改inode数据引起的。

若inode为脏(注意inode为脏和页面数据为脏，是两回事)，就是用超级块的write_inode（）方法将inode写到磁盘上（353～357行）。

```
00359:      spin_lock(&inode_lock);
00360:      inode->i_state &= ~I_SYNC;
00361:      if (!(inode->i_state & (I_FREEING | I_CLEAR))) {
00362:          if ((inode->i_state & I_DIRTY_PAGES) &&
                          wbc->for_kupdate) {
00363:              /*
00364:               * More pages get dirtied by a fast dirtier.
00365:               */
00366:              goto ↓select_queue;
00367:          } else if (inode->i_state & I_DIRTY) {
00368:              /*
00369:               * At least XFS will redirty the inode during the
00370:               * writeback (delalloc) and on io completion (isize).
00371:               */
00372:              redirty_tail(inode);
00373:          } else if (mapping_tagged(mapping,
                          PAGECACHE_TAG_DIRTY)) {
```

```
            /*
             * We didn't write back all the pages.  nfs_writepages()
             * sometimes bales out without doing anything. Redirty
             * the inode; Move it from b_io onto b_more_io/b_dirty.
             */
            /*
             * akpm: if the caller was the kupdate function we put
             * this inode at the head of b_dirty so it gets first
             * consideration.  Otherwise, move it to the tail, for
             * the reasons described there.  I'm not really sure
             * how much sense this makes.  Presumably I had a good
             * reasons for doing it this way, and I'd rather not
             * muck with it at present.
             */
            if (wbc->for_kupdate) {
                /*
                 * For the kupdate function we move the inode
                 * to b_more_io so it will get more writeout as
                 * soon as the queue becomes uncongested.
                 */
                inode->i_state |= I_DIRTY_PAGES;
select_queue:
                if (wbc->nr_to_write <= 0) {
                    /*
                     * slice used up: queue for next turn
                     */
                    requeue_io(inode);
                } else {
                    /*
                     * somehow blocked: retry later
                     */
                    redirty_tail(inode);
                }
            } ?  end if wbc- >for_kupdate ?  else {
                /*
                 * Otherwise fully redirty the inode so that
                 * other inodes on this superblock will get some
                 * writeout.  Otherwise heavy writing to one
                 * file would indefinitely suspend writeout of
                 * all the other files.
                 */
                inode->i_state |= I_DIRTY_PAGES;
                redirty_tail(inode);
            }
        } ?  end if mapping_tagged(mappin... ?  else if
            (atomic_read(&inode->i_count)) {
            /*
             * The inode is clean, inuse
             */
            list_move(&inode->i_list, &inode_in_use);
        } else {
            /*
             * The inode is clean, unused
             */
```

```
00427:             list_move(&inode->i_list, &inode_unused);
00428:         }
00429:     } ?  end if ! (inode- >i_state&(I_F... ?
00430:     inode_sync_complete(inode);
00431:     return ret;
00432: } ?  end writeback_single_inode ?
00433:
```

回写过程中，inode的状态可能发生了变化。do_writepages（）来回写脏页动作完成后，需要清除I_SYNC标志（完成一轮回写），然后再次检查inode状态。若inode的状态变为脏，那么需要再次放入相应的链表上（361～429行）。

最后调用inode_sync_complete（）将该inode上的所有等待进程唤醒（430行）。

3.6 do_writepages（）

```
int do_writepages(struct address_space *mapping, struct writeback_control *
wbc)
{
    int ret;

    if (wbc->nr_to_write <= 0)
        return 0;
    if (mapping->a_ops->writepages)
        ret = mapping->a_ops->writepages(mapping, wbc);
    else
        ret = generic_writepages(mapping, wbc);
    return ret;
}
```

do_writepages（）函数简单封装a_ops->writepages（）或generic_writepages（）。本节以ext4文件系统delay allocation机制为例，定义了相应的writepages方法ext4_da_writepages（）。

继续分析接下来的执行流程前，我们先看一下内核调用栈信息。

```
Pid: 1211, comm: flush-8:0 Tainted: G       --------------- HT 2.6.32279.debug #33
Call Trace:
 [<ffffffff8125720f>] ? generic_make_request+0x56f/0x580
 [<ffffffff812572dc>] ? submit_bio+0xbc/0x160
 [<ffffffff811acd46>] ? submit_bh+0xf6/0x150
 [<ffffffff811aeab0>] ? __block_write_full_page+0x1e0/0x3b0
 [<ffffffff811ae3f0>] ? end_buffer_async_write+0x0/0x190
 [<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
```

```
[<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
[<ffffffff811af6f0>] ? block_write_full_page_endio+0xe0/0x120
[<ffffffffa00dbe40>] ? ext4_bh_delay_or_unwritten+0x0/0x30 [ext4]
[<ffffffff811af745>] ? block_write_full_page+0x15/0x20
[<ffffffffa00e1722>] ? ext4_writepage+0x172/0x400 [ext4]
[<ffffffffa00e1af7>] ? mpage_da_submit_io+0x147/0x1d0 [ext4]
[<ffffffffa00e1d22>] ? mpage_da_map_and_submit+0x1a2/0x450 [ext4]
[<ffffffff81277f45>] ? radix_tree_gang_lookup_tag_slot+0x95/0xe0
[<ffffffff81113bd0>] ? find_get_pages_tag+0x40/0x120
[<ffffffffa00e203d>] ? mpage_add_bh_to_extent+0x6d/0xf0 [ext4]
[<ffffffffa00e238f>] ? write_cache_pages_da+0x2cf/0x470 [ext4]
[<ffffffffa00e2802>] ? ext4_da_writepages+0x2d2/0x620 [ext4]
[<ffffffff811299e1>] ? do_writepages+0x21/0x40
[<ffffffff811a500d>] ? writeback_single_inode+0xdd/0x2c0
[<ffffffff811a544e>] ? writeback_sb_inodes+0xce/0x180
[<ffffffff811a5812>] ? wb_writeback+0x162/0x3f0
[<ffffffff8107c981>] ? ftrace_raw_event_timer_cancel+0xa1/0xb0
[<ffffffff8107eb42>] ? del_timer_sync+0x22/0x30
[<ffffffff811a5b5b>] ? wb_do_writeback+0xbb/0x240
[<ffffffff811a5d43>] ? bdi_writeback_task+0x63/0x1b0
[<ffffffff81091f97>] ? bit_waitqueue+0x17/0xd0
[<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
[<ffffffff811386c6>] ? bdi_start_fn+0x86/0x100
[<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
[<ffffffff81091d66>] ? kthread+0x96/0xa0
[<ffffffff8100c14a>] ? child_rip+0xa/0x20
[<ffffffff81091cd0>] ? kthread+0x0/0xa0
[<ffffffff8100c140>] ? child_rip+0x0/0x20
```

3.7 ext4文件系统写数据流程

3.7.1 ext4_da_writepages（）

ext4_da_writepages（）函数定义在文件fs/ext4/inode.c中。ext4文件系统delay allocation的核心思想：等到回写脏缓存页面时再建立脏页面与物理磁盘块之间的映射，并且文件逻辑上连续的块会映射到物理上连续的磁盘块。

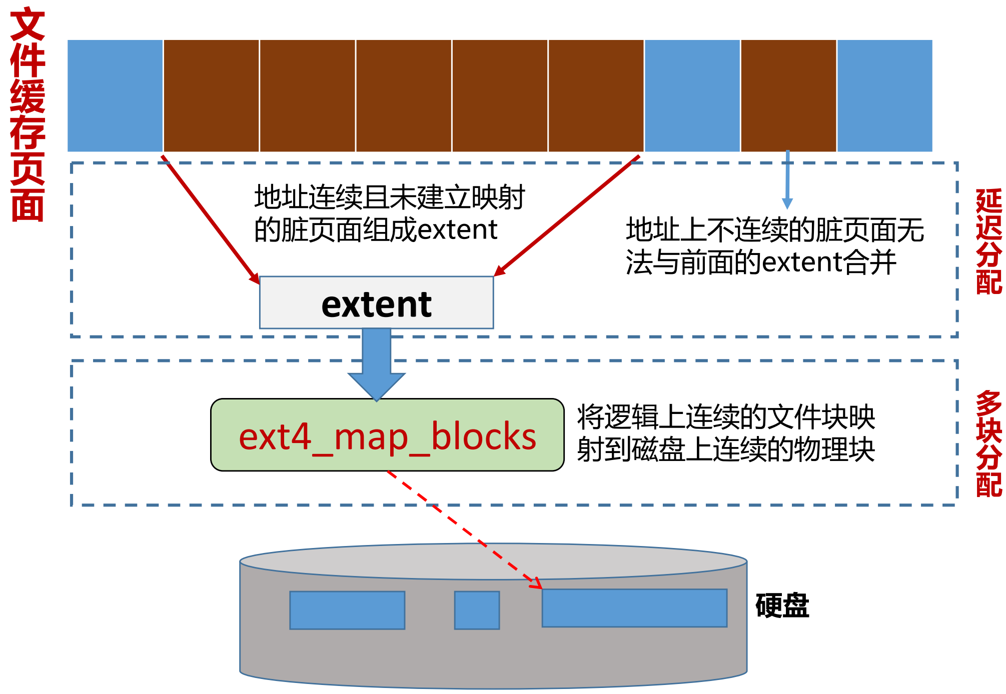

图3 ext4文件系统extent基本原理

函数主要功能：（1）若逻辑上连续的脏页面没有建立磁盘block映射，那么就要形成一个extent，用于ext4的mballoc分配策略，提升文件连续性。（2）对连续页面形成的extent进行磁盘块分配，分配采用了ext4的mballoc策略。此时已经为尚未映射的缓存页面分配了物理磁盘块。（3）提交extent至bio层完成脏页面的写入。

```
static int ext4_da_writepages(struct address_space
                    *mapping, struct writeback_control *wbc)
{
        pgoff_t index;
        int range_whole = 0;
        handle_t *handle = NULL;
        struct mpage_da_data mpd;
        struct inode *inode = mapping->host;
        int pages_written = 0;
        long pages_skipped;
        unsigned int max_pages;
        int range_cyclic, cycled = 1, io_done = 0;
        int needed_blocks, ret = 0;
        long desired_nr_to_write, nr_to_writebump = 0;
        loff_t range_start = wbc->range_start;
        struct ext4_sb_info *sbi = EXT4_SB(mapping->host->i_sb);
        pgoff_t done_index = 0;
```

```
03039:     pgoff_t end;
03040:
03041:     trace_ext4_da_writepages(inode, wbc);
03042:
03043:     /*
03044:      * No pages to write? This is mainly a kludge to avoid starting
03045:      * a transaction for special inodes like journal inode on last iput()
03046:      * because that could violate lock ordering on umount
03047:      */
03048:     if (!mapping->nrpages || !mapping_tagged(mapping,
03048:               PAGECACHE_TAG_DIRTY))
03049:         return 0;
03050:
```

首先检查mapping（含义及用途，请见address_space章节）的页面数量和检查radix tree中的页面是否有脏页面。若没有脏页面或nrpages数量为0，则表示没有页面可写，直接返回（3048～3049行）。

```
03051:     /*
03052:      * If the filesystem has aborted, it is read-only, so return
03053:      * right away instead of dumping stack traces later on that
03054:      * will obscure the real source of the problem.  We test
03055:      * EXT4_MF_FS_ABORTED instead of sb->s_flag's MS_RDONLY
03056:      * because the latter could be true if the filesystem is mounted
03057:      * read-only, and in that case, ext4_da_writepages should
03058:      * *never* be called, so if that ever happens, we would want
03059:      * the stack trace.
03060:      */
03061:     if (unlikely(sbi->s_mount_flags & EXT4_MF_FS_ABORTED))
03062:         return -EROFS;
03063:
```

若检测到文件系统加载标志包含EXT4_MF_FS_ABORTED，表明ext4文件系统出现过异常退出（如系统宕机），返回文件系统只读标志（3061～3062行）。

```
03064:     if (wbc->range_start == 0 && wbc->range_end ==
                                         LLONG_MAX)
03065:         range_whole = 1;
03066:
03067:     range_cyclic = wbc->range_cyclic;
03068:     if (wbc->range_cyclic) {
03069:         index = mapping->writeback_index;
03070:         if (index)
03071:             cycled = 0;
03072:         wbc->range_start = index << PAGE_CACHE_SHIFT;
03073:         wbc->range_end  = LLONG_MAX;
03074:         wbc->range_cyclic = 0;
03075:         end = -1;
03076:     } else {
```

```
03077:          index = wbc->range_start >> PAGE_CACHE_SHIFT;
03078:          end = wbc->range_end >> PAGE_CACHE_SHIFT;
03079:      }
03080:
```

检查回写范围，若范围为[0, LLONG_MAX]，就表示同步所有页面（3064～3065 行）。wbc->range_cyclic 值为 0 表示回写操作范围要限制在[range_start, range_end]（3076～3079 行）；若值为 1，则表示内核可以对 mapping 里的页面执行多次回写操作（3068～3075 行）。

```
03081:      /*
03082:       * This works around two forms of stupidity.  The first is in
03083:       * the writeback code, which caps the maximum number of pages
03084:       * written to be 1024 pages.  This is wrong on multiple
03085:       * levels; different architectues have a different page size,
03086:       * which changes the maximum amount of data which gets
03087:       * written.  Secondly, 4 megabytes is way too small.  XFS
03088:       * forces this value to be 16 megabytes by multiplying
03089:       * nr_to_write parameter by four, and then relies on its
03090:       * allocator to allocate larger extents to make them
03091:       * contiguous.  Unfortunately this brings us to the second
03092:       * stupidity, which is that ext4's mballoc code only allocates
03093:       * at most 2048 blocks.  So we force contiguous writes up to
03094:       * the number of dirty blocks in the inode, or
03095:       * sbi->max_writeback_mb_bump whichever is smaller.
03096:       */
03097:      max_pages = sbi->s_max_writeback_mb_bump << (20 -
                                PAGE_CACHE_SHIFT);
03098:      if (!range_cyclic && range_whole) {
03099:          if (wbc->nr_to_write == LONG_MAX)
03100:              desired_nr_to_write = wbc->nr_to_write;
03101:          else
03102:              desired_nr_to_write = wbc->nr_to_write * 8;
03103:      } else
03104:          desired_nr_to_write = ext4_num_dirty_pages(inode,
03105:                                  index, max_pages);
03106:      if (desired_nr_to_write > max_pages)
03107:          desired_nr_to_write = max_pages;
03108:
03109:      if (wbc->nr_to_write < desired_nr_to_write) {
03110:          nr_to_writebump = desired_nr_to_write -
                                      wbc->nr_to_write;
03111:          wbc->nr_to_write = desired_nr_to_write;
03112:      }
03113:
```

ext4_num_dirty_pages（）函数功能是返回指定inode下面以起始页面号为idx的连续脏页面数量（3104～3105行）。

ext4文件系统的delay allocation机制是为文件连续的逻辑块分配磁盘上连续的物理存

储块，建立连续块映射关系后，也希望将页面缓存的连续脏页数据一次性写入。但传递过来的wb->nr_to_write可能和内核返回的连续脏页数量不一致，这就是desired_nr_to_write，nr_to_writebump两个变量的用途，用于更新wbc->nr_to_write（）的值（3097～3112行，回写的页面数量有可能比nr_to_write多）。

```
03114:     mpd.wbc = wbc;
03115:     mpd.inode = mapping->host;
03116:
03117:     pages_skipped = wbc->pages_skipped;
03118:
03119: retry:
03120:     if (wbc->sync_mode == WB_SYNC_ALL)
03121:         tag_pages_for_writeback(mapping, index, end);
03122:
```

若同步模式为WB_SYNC_ALL，则调用tag_pages_for_writeback（）来扫描mapping中[index, end]范围内的所有脏页面，然后将页面设置特殊标志TOWRITE，表示该页面是可回写的有效页面。

若没有回写错误（ret不等于0）并且要回写的脏页面数量没完成，就要执行3123~3201行来执行回写页面操作。

```
03123:     while (!ret && wbc->nr_to_write > 0) {
03124:
03125:         /*
03126:         * we  insert one extent at a time. So we need
03127:          * credit needed for single extent allocation.
03128:          * journalled mode is currently not supported
03129:          * by delalloc
03130:          */
03131:         BUG_ON(ext4_should_journal_data(inode));
03132:         needed_blocks =
                     ext4_da_writepages_trans_blocks(inode);
03133:
03134:         /* start a new transaction*/
03135:         handle = ext4_journal_start(inode, needed_blocks);
03136:         if (IS_ERR(handle)) {
03137:             ret = PTR_ERR(handle);
03138:             ext4_msg(inode->i_sb, KERN_CRIT, "%s: jbd2_start: "
03139:                 "%ld pages, ino %lu; err %d\n", __func__,
03140:                 wbc->nr_to_write, inode->i_ino, ret);
03141:             goto ↓out_writepages;
03142:         }
03143:
03144:         /*
03145:          * Now call ____mpage_da_writepage to find the next
```

```
         * contiguous region of logical blocks that need
         * blocks to be allocated by ext4.  We don't actually
         * submit the blocks for I/O here, even though
         * write_cache_pages thinks it will, and will set the
         * pages as clean for write before calling
         * __mpage_da_writepage().
         */
        mpd.b_size = 0;
        mpd.b_state = 0;
        mpd.b_blocknr = 0;
        mpd.first_page = 0;
        mpd.next_page = 0;
        mpd.io_done = 0;
        mpd.pages_written = 0;
        mpd.retval = 0;
        ret = write_cache_pages_da(mapping, wbc, &mpd,
                &done_index);
        /*
         * If we have a contigous extent of pages and we
         * haven't done the I/O yet, map the blocks and submit
         * them for I/O.
         */
        if (!mpd.io_done && mpd.next_page != mpd.first_page) {
            mpage_da_map_and_submit(&mpd);
            ret = MPAGE_DA_EXTENT_TAIL;
        }
        trace_ext4_da_write_pages(inode, &mpd);
        wbc->nr_to_write -= mpd.pages_written;

        ext4_journal_stop(handle);

        if ((mpd.retval == -ENOSPC) && sbi->s_journal) {
            /* commit the transaction which would
             * free blocks released in the transaction
             * and try again
             */
            jbd2_journal_force_commit_nested(sbi->s_journal);
            wbc->pages_skipped = pages_skipped;
            ret = 0;
        } else if (ret == MPAGE_DA_EXTENT_TAIL) {
            /*
             * Got one extent now try with rest of the pages.
             * If mpd.retval is set -EIO, journal is aborted.
             * So we don't need to write any more.
             */
            pages_written += mpd.pages_written;
            wbc->pages_skipped = pages_skipped;
            ret = mpd.retval;
            io_done = 1;
        } else if (wbc->nr_to_write)
            /*
             * There is no more writeout needed
             * or we requested for a noblocking writeout
             * and we found the device congested
```

```
03199:                 */
03200:                break;
03201:        } ?   end while ! ret&&wbc- >nr_to_write>0 ?
```

在回写页面操作执行前，调用ext4_da_writepages_trans_blocks（）计算回写操作时journal区域大小（3132行），然后开始journal（3135行）。启动journal失败的话，就要报错，不能执行回写页面操作（3136～3142行）。

变量mpd的数据结构类型为mpage_da_data，主要用于记录跟踪ext4文件系统delay allocation信息。write_cache_pages_da（）函数负责将逻辑上连续的文件块合并成一个extent（3161行）。

若我们现在已经有了extent连续页面，且没有执行I/O（3167行），就调用mpage_da_map_and_sumbit（），将I/O回写请求提交到下一层。调用返回后，数据回写完成，更新wbc->nr_to_write的值（3172行），停止ext4的journal（3174行）。

回写完成后，就需要检查返回值情况（3176～3200行）。回写数据过程中，可能存储写满了，返回值为-ENOSPC，就要确保journal执行完成（3176～3183行）。多数情况下，是调用mpage_da_map_and_sumbit（）回写连续的extent脏页面，并设置ret返回值为MPAGE_DA_EXTENT_TAIL（3169行）。一个extent回写操作完成后，需要更新完成回写页面的数量、跳过的页面数量（3184～3193行）。最后一种情况是wbc->nr_to_write的值仍大于0，表示不需要更多的I/O回写或者是我们请求非阻塞写，但存储设备拥塞了；这两种情况，都要退出（3194～3200行）。

```
03202:        if (!io_done && !cycled) {
03203:            cycled = 1;
03204:            index = 0;
03205:            wbc->range_start = index << PAGE_CACHE_SHIFT;
03206:            wbc->range_end  = mapping->writeback_index - 1;
03207:            goto ↑retry;
03208:        }
03209:        if (pages_skipped != wbc->pages_skipped)
03210:            ext4_msg(inode->i_sb, KERN_CRIT,
03211:                 "This should not happen leaving %s "
03212:                 "with nr_to_write = %ld ret = %d\n",
03213:                  __func__, wbc->nr_to_write, ret);
03214:
03215:        /* Update index */
```

```
03216:     wbc->range_cyclic = range_cyclic;
03217:     if (wbc->range_cyclic || (range_whole && wbc->nr_to_write > 0))
03218:         /*
03219:          * set the writeback_index so that range_cyclic
03220:          * mode will write it back later
03221:          */
03222:         mapping->writeback_index = done_index;
03223:
03224: out_writepages:
03225:     wbc->nr_to_write -= nr_to_writebump;
03226:     wbc->range_start = range_start;
03227:     trace_ext4_da_writepages_result(inode, wbc, ret,
                        pages_written);
03228:     return ret;
03229: } ?  end ext4_da_writepages ?
03230:
```

经过前面的while循环后，回写脏页面请求基本上完成了，但仍有特殊情况。若I/O操作仍没完成（io_done的值为0，写操作完成的话会在3193行将io_done的值设为1）且write control work里的wbc->range_cyclic值为1（表示可以多次回写），就要重新尝试回写一次。注意，尝试回写仅有一次，不会有第二次或第三次回写（3203行设置了cycled的值为1）。

不管怎样，跳过的页面数量与回写任务（wbc->pages_skipped）里的值不一样，那就表示出现了异常（3209~3213行）。

最后就是回写任务里的变量值更新（3216～3226行），这里不作详细解释。

3.7.2 mpage_da_map_and_submit（）

mpage_da_map_and_submit（）函数定义在文件fs/ext4/inode.c中。主要功能是对回写任务空间进行扫描，必要时候建立映射关系，然后调用mpage_da_submit_io（）提交写I/O任务。

```
/*
 * mpage_da_map_and_submit - go through given space, map them
 *     if necessary, and then submit them for I/O
 *
 * @mpd - bh describing space
 *
 * The function skips space we know is already mapped to disk blocks.
 *
 */
static void mpage_da_map_and_submit(struct
                              mpage_da_data *mpd)
{
    int err, blks, get_blocks_flags;
```

```
	struct buffer_head new;
	sector_t next = mpd->b_blocknr;
	unsigned max_blocks = mpd->b_size >>
				mpd->inode->i_blkbits;
	loff_t disksize = EXT4_I(mpd->inode)->i_disksize;
	handle_t *handle = NULL;

	/*
	 * If the blocks are mapped already, or we couldn't accumulate
	 * any blocks, then proceed immediately to the submission stage.
	 */
	if ((mpd->b_size == 0) ||
	    ((mpd->b_state  & (1 << BH_Mapped)) &&
	     !(mpd->b_state & (1 << BH_Delay)) &&
	     !(mpd->b_state & (1 << BH_Unwritten))))
		goto ↓submit_io;

```

代码中的注释清楚，当这些回写数据块已经与磁盘block建立了映射，或者不能累加更多的块，就直接跳转到submit_io（），提交I/O请求。

```
	handle = ext4_journal_current_handle();
	BUG_ON(!handle);

	/*
	 * Call ext4_get_blocks() to allocate any delayed allocation
	 * blocks, or to convert an uninitialized extent to be
	 * initialized (in the case where we have written into
	 * one or more preallocated blocks).
	 *
	 * We pass in the magic EXT4_GET_BLOCKS_DELALLOC_RESERVE to
	 * indicate that we are on the delayed allocation path.  This
	 * affects functions in many different parts of the allocation
	 * call path.  This flag exists primarily because we don't
	 * want to change *many* call functions, so ext4_get_blocks()
	 * will set the magic i_delalloc_reserved_flag once the
	 * inode's allocation semaphore is taken.
	 *
	 * If the blocks in questions were delalloc blocks, set
	 * EXT4_GET_BLOCKS_DELALLOC_RESERVE so the delalloc accounting
	 * variables are updated after the blocks have been allocated.
	 */
	new.b_state = 0;
	get_blocks_flags = EXT4_GET_BLOCKS_CREATE;
	if (mpd->b_state & (1 << BH_Delay))
		get_blocks_flags |=
			EXT4_GET_BLOCKS_DELALLOC_RESERVE;

	blks = ext4_get_blocks(handle, mpd->inode, next,
				max_blocks, &new, get_blocks_flags);
	if (blks < 0) {
		err = blks;
```

```
        /*
         * If get block returns EAGAIN or ENOSPC and there
         * appears to be free blocks we will call
         * ext4_writepage() for all of the pages which will
         * just redirty the pages.
         */
        if (err == -EAGAIN)
            goto ↓submit_io;

        if (err == -ENOSPC &&
            ext4_count_free_blocks(mpd->inode->i_sb)) {
            mpd->retval = err;
            goto ↓submit_io;
        }

        /*
         * get block failure will cause us to loop in
         * writepages, because a_ops->writepage won't be able
         * to make progress. The page will be redirtied by
         * writepage and writepages will again try to write
         * the same.
         */
        ext4_msg(mpd->inode->i_sb, KERN_CRIT,
             "delayed block allocation failed for inode %lu at "
             "logical offset %llu with max blocks %zd with "
             "error %d\n", mpd->inode->i_ino,
             (unsigned long long) next,
             mpd->b_size >> mpd->inode->i_blkbits, err);
        printk(KERN_CRIT "This should not happen!!    "
            "Data will be lost\n");
        if (err == -ENOSPC) {
            ext4_print_free_blocks(mpd->inode);
        }
        /* invalidate all the pages */
        ext4_da_block_invalidatepages(mpd);

        /* Mark this page range as having been completed */
        mpd->io_done = 1;
        return;
    } ?  end if blks<0 ?
    BUG_ON(blks == 0);
```

ext4_get_blocks（）函数功能：确保请求blocks与磁盘上的块建立映射关系。先尝试查找请求的块是否已经建立映射，若已有映射，则直接返回；否则就要从磁盘存储上分配块并建立映射。

成功查找到或分配空间并建立映射关系，ext4_get_blocks（）返回值就是建立映射的块数量；返回值为0，就是没有建立任何映射关系，也没有分配任何存储块；发生了错误，返回值就小于0。

返回值为-EAGAIN（表示可以重新尝试）或 ，返回值为-ENOSPC（表示没有设备上没有空间）且还有空闲block的话，就直接跳转到submit_io提交任务，在后面的ext4_writepage（）会处理，重新标记页面为脏（2327～2334行）。若不是这两种出错情况，那表明分配block确实失败了，这种情况不应该出现的（2343～2359行）， 只能调用ext4_da_block_invalidatepages（）将这次待回写的所有页面设为无效。

```
02363:      new.b_size = (blks << mpd->inode->i_blkbits);
02364:
02365:      if (buffer_new(&new))
02366:          __unmap_underlying_blocks(mpd->inode, &new);
02367:
02368:      /*
02369:       * If blocks are delayed marked, we need to
02370:       * put actual blocknr and drop delayed bit
02371:       */
02372:       if ((mpd->b_state & (1 << BH_Delay)) ||
02373:         (mpd->b_state & (1 << BH_Unwritten)))
02374:          mpage_put_bnr_to_bhs(mpd, next, &new);
02375:
02376:      if (ext4_should_order_data(mpd->inode)) {
02377:          err = ext4_jbd2_file_inode(handle, mpd->inode);
02378:          if (err) {
02379:              /* This only happens if the journal is aborted */
02380:              mpd->retval = err;
02381:              goto ↓submit_io;
02382:          }
02383:      }
02384:
```

若ext4_get_blocks（）是新分配blocks并建立映射，buffer head的状态就为BH_New，就取消buffer head下面的所有块映射（2365～2366行）。

若还没有为buffer在磁盘上分配空间（BH_Delay）或buffer有了磁盘空间但数据还没写入，则调用mpage_put_bnr_to_bhs（）建立blocks的映射并清除延迟标志。

当inode的journal模式为EXT4_INODE_ORDER_DATA_MODE时，需要通过ext4_jbd2_file_inode（）来更新inode的transaction。发生journal异常退出时，记录回写任务的返回值为-EIO，然后直接跳转到submit_io（2378～2382行）。

```
02385:      /*
02386:       * Update on-disk size along with block allocation.
02387:       */
02388:       disksize = ((loff_t) next + blks) << mpd->inode->i_blkbits;
02389:      if (disksize > i_size_read(mpd->inode))
02390:          disksize = i_size_read(mpd->inode);
```

```
02391:     if (disksize > EXT4_I(mpd->inode)->i_disksize) {
02392:         ext4_update_i_disksize(mpd->inode, disksize);
02393:         err = ext4_mark_inode_dirty(handle, mpd->inode);
02394:         if (err)
02395:             ext4_error(mpd->inode->i_sb,
02396:                    "Failed to mark inode %lu dirty",
02397:                    mpd->inode->i_ino);
02398:     }
02399:
02400: submit_io:
02401:     mpage_da_submit_io(mpd);
02402:     mpd->io_done = 1;
02403: } ?  end mpage_da_map_and_submit  ?
```

ext4_inode_info数据结构中i_disksize记录inode在磁盘上的大小，即在磁盘上占用的实际空间。2388～2398行，更新ext4 inode的i_disksize。

最后调用mpage_da_submit_io（）回写脏页面。

3.7.3 mpage_da_submit_io（）

mpage_da_submit_io（）函数定义在文件fs/ext4/inode.c中。调用pagevec_lookup（），根据回写任务的extent开始和结束位置找到address space中对应的页面，然后调用ext4_writepage（）将每个页面回写到磁盘上。

```
/ *
  * Delayed allocation stuff
  */

/ *
  * mpage_da_submit_io -  walks through extent of pages and try to write
  * them with writepage() call back
  *
  * @mpd- >inode: inode
  * @mpd- >first_page: first page of the extent
  * @mpd- >next_page: page after the last page of the extent
  *
  * By the time mpage_da_submit_io() is called we expect all blocks
  * to be allocated. this may be wrong if allocation failed.
  *
  * As pages are already locked by write_cache_pages(), we can't use it
  */
static int mpage_da_submit_io(struct mpage_da_data
                                    *mpd)
{
    long pages_skipped;
    struct pagevec pvec;
```

```
	unsigned long index, end;
	int ret = 0, err, nr_pages, i;
	struct inode *inode = mpd->inode;
	struct address_space *mapping = inode->i_mapping;

	BUG_ON(mpd->next_page <= mpd->first_page);
	/*
	 * We need to start from the first_page to the next_page - 1
	 * to make sure we also write the mapped dirty buffer_heads.
	 * If we look at mpd->b_blocknr we would only be looking
	 * at the currently mapped buffer_heads.
	 */
	index = mpd->first_page;
	end = mpd->next_page - 1;

	pagevec_init(&pvec, 0);
	while (index <= end) {
		nr_pages = pagevec_lookup(&pvec, mapping, index,
					PAGEVEC_SIZE);
		if (nr_pages == 0)
			break;
		for (i = 0; i < nr_pages; i++) {
			struct page *page = pvec.pages[i];

			index = page->index;
			if (index > end)
				break;
			index++;

			BUG_ON(!PageLocked(page));
			BUG_ON(PageWriteback(page));

			pages_skipped = mpd->wbc->pages_skipped;
			err = ext4_writepage(page, mpd->wbc);
			if (!err && (pages_skipped ==
					mpd->wbc->pages_skipped))
				/*
				 * have successfully written the page
				 * without skipping the same
				 */
				mpd->pages_written++;
			/*
			 * In error case, we have to continue because
			 * remaining pages are still locked
			 * XXX: unlock and re-dirty them?
			 */
			if (ret == 0)
				ret = err;
		} ?   end for i=0;i<nr_pages;i++ ?
		pagevec_release(&pvec);
	} ?   end while index<=end ?
	return ret;
} ?   end mpage_da_submit_io ?
```

3.7.4 ext4_writepage（）

ext4_writepage（）函数定义在文件fs/ext4/inode.c中。ext4_writepage（）回写单个页面，并调用block_write_full_page（）来把页面mark成PG_WRITEBACK。

```
02777: static int ext4_writepage(struct page *page,
02778:                 struct writeback_control *wbc)
02779: {
02780:     int ret = 0;
02781:     loff_t size;
02782:     unsigned int len;
02783:     struct buffer_head *page_bufs;
02784:     struct inode *inode = page->mapping->host;
02785:
02786:     trace_ext4_writepage(inode, page);
02787:     size = i_size_read(inode);
02788:     if (page->index == size >> PAGE_CACHE_SHIFT)
02789:         len = size & ~PAGE_CACHE_MASK;
02790:     else
02791:         len = PAGE_CACHE_SIZE;
02792:
02793:     if (page_has_buffers(page)) {
02794:         page_bufs = page_buffers(page);
02795:         if (walk_page_buffers(NULL, page_bufs, 0, len, NULL,
02796:                     ext4_bh_delay_or_unwritten)) {
02797:             /*
02798:              * We don't want to do  block allocation
02799:              * So redirty the page and return
02800:              * We may reach here when we do a journal commit
02801:              * via journal_submit_inode_data_buffers.
02802:              * If we don't have mapping block we just ignore
02803:              * them. We can also reach here via shrink_page_list
02804:              * but it should never be for direct reclaim so warn
02805:              * if that happens
02806:              */
02807:             WARN_ON_ONCE((current->flags &
                     (PF_MEMALLOC|PF_KSWAPD)) ==
02808:                     PF_MEMALLOC);
02809:             redirty_page_for_writepage(wbc, page);
02810:             unlock_page(page);
02811:             return 0;
02812:         }
02813:     } ?   end if page_has_buffers(page) ?  else {
```

若待写的页面已经在page cache中（2793行），接下来调用walk_page_buffers（）就

对页面中的buffer进行检查。当看页面是否脏（BH_Dirty），且页面已分配空间但未写数据（BH_Unwritten）或则需要延迟分配磁盘空间（BH_Delay），此时直接将页面重新标记为脏，不实际回写数据，直接返回（2795～2111行）。

```
02814:          / *
02815:           * The test for page_has_buffers() is subtle:
02816:           * We know the page is dirty but it lost buffers. That means
02817:           * that at some moment in time after write_begin()/ write_end()
02818:           * has been called all buffers have been clean and thus they
02819:           * must have been written at least once. So they are all
02820:           * mapped and we can happily proceed with mapping them
02821:           * and writing the page.
02822:           *
02823:           * Try to initialize the buffer_heads and check whether
02824:           * all are mapped and non delay. We don't want to
02825:           * do block allocation here.
02826:           */
02827:          ret = block_prepare_write(page, 0, len,
02828:                          noalloc_get_block_write);
02829:          if (!ret) {
02830:              page_bufs = page_buffers(page);
02831:              / * check whether all are mapped and non delay */
02832:              if (walk_page_buffers(NULL, page_bufs, 0, len, NULL,
02833:                              ext4_bh_delay_or_unwritten)) {
02834:                  redirty_page_for_writepage(wbc, page);
02835:                  unlock_page(page);
02836:                  return 0;
02837:              }
02838:          } else {
02839:              / *
02840:               * We can't do block allocation here
02841:               * so just redity the page and unlock
02842:               * and return
02843:               */
02844:              redirty_page_for_writepage(wbc, page);
02845:              unlock_page(page);
02846:              return 0;
02847:          }
02848:      / * now mark the buffer_heads as dirty and uptodate */
02849:          block_commit_write(page, 0, len);
02850:      } ?   end else ?
02851:
```

block_prepare_write（）函数主要功能是为一个page准备一组buffer_head结构，用于描述组成这个page的数据块 （2827～2828行）。为page准备好buffer head后，再次检查是否页面的buffer是否延迟分配（2832～2836行）。若为page建立buffer head失败，则重新将页面标记为脏，解锁页面，并返回（2844～2846行）。

通过上述准备后，调用block_commit_write（）将buffer标记为脏和update状态。

```
    if (PageChecked(page) && ext4_should_journal_data(inode)) 
{
        /*
         * It's mmapped pagecache.  Add buffers and journal it.  There
         * doesn't seem much point in redirtying the page here.
         */
        ClearPageChecked(page);
        return __ext4_journalled_writepage(page, wbc, len);
    }

    if (test_opt(inode->i_sb, NOBH) &&
                ext4_should_writeback_data(inode))
        ret = nobh_writepage(page, noalloc_get_block_write,
                wbc);
    else
        ret = block_write_full_page(page,
        noalloc_get_block_write,wbc);

    return ret;
} ?  end ext4_writepage ?
```

ext4是日志文件系统，有多种日志模式。若写入数据时，需要采用日志模式（ext4_should_journal_data（））,就调用__ext4_journalled_writepage（）来写数据。

若写入数据，不需要buffer head（NOBH），就调用nobh_writepage（），否则调用block_write_full_page（）来执行写页面操作。

默认情况下，执行回写页面的函数是block_write_full_page（）。

3.8 address space写数据流程

3.8.1 block_write_full_page（）

block_write_full_page（）函数是对block_write_full_page_endio（）的封装。函数源码在fs/buffer.c中。

```
/*
 * The generic - >writepage function for buffer- backed address_spaces
 */
int block_write_full_page(struct page *page,
        get_block_t *get_block,struct writeback_control *wbc)
{
    return block_write_full_page_endio(page, get_block,
            wbc, end_buffer_async_write);
}
```

3.8.2 block_write_full_page_endio（）

block_write_full_page_endio（）函数是对__block_write_full_page（）的封装。函数源码在fs/buffer.c中。

```
02965: /*
02966:  * The generic ->writepage function for buffer-backed address_spaces
02967:  * this form passes in the end_io handler used to finish the IO.
02968:  */
02969: int block_write_full_page_endio(
02969:             struct page *page, get_block_t *get_block,
02970:             struct writeback_control *wbc, bh_end_io_t *handler)
02971: {
02972:     struct inode * const inode = page->mapping->host;
02973:     loff_t i_size = i_size_read(inode);
02974:     const pgoff_t end_index = i_size >> PAGE_CACHE_SHIFT;
02975:     unsigned offset;
02976:
02977:     /* Is the page fully inside i_size? */
02978:     if (page->index < end_index)
02979:         return __block_write_full_page(inode, page,
02980:                     get_block, wbc, handler);
02981:
02982:     /* Is the page fully outside i_size? (truncate in progress) */
02983:     offset = i_size & (PAGE_CACHE_SIZE-1);
02984:     if (page->index >= end_index+1 || !offset) {
02985:         /*
02986:          * The page may have dirty, unmapped buffers.  For example,
02987:          * they may have been added in ext3_writepage().  Make them
02988:          * freeable here, so the page does not leak.
02989:          */
02990:         do_invalidatepage(page, 0);
02991:         unlock_page(page);
02992:         return 0; /* don't care */
02993:     }
02994:
```

写数据之前，还是要检查待写入的数据是否超过文件大小（2978行）。为了分析方便，这里只考虑写入数据没超过文件大小，调用__block_write_full_page（）来进一步执行写数据。

3.8.3 __block_write_full_page（）

__block_write_full_page（）主要功能是：若目前该页面不是一个缓冲页面，则为该页面分配buffer heads；然后对每个buffer，调用submit_bh（）来处理。

```
01638: static int __block_write_full_page(struct inode
```

```
*inode, struct page *page,
                get_block_t *get_block, struct writeback_control *wbc,
                bh_end_io_t *handler)
{
        int err;
        sector_t block;
        sector_t last_block;
        struct buffer_head *bh, *head;
        const unsigned blocksize = 1 << inode->i_blkbits;
        int nr_underway = 0;
        int write_op = (wbc->sync_mode == WB_SYNC_ALL ?
                        WRITE_SYNC_PLUG : WRITE);

        BUG_ON(!PageLocked(page));

        last_block = (i_size_read(inode) - 1) >> inode->i_blkbits;

        if (!page_has_buffers(page)) {
                create_empty_buffers(page, blocksize,
                                        (1 << BH_Dirty)|(1 << BH_Uptodate));
        }

        /*
         * Be very careful.  We have no exclusion from
                        __set_page_dirty_buffers
         * here, and the (potentially unmapped) buffers may become dirty at
         * any time.  If a buffer becomes dirty here after we've inspected it
         * then we just miss that fact, and the page stays dirty.
         *
         * Buffers outside i_size may be dirtied by ____set_page_dirty_buffers;
         * handle that here by just cleaning them.
         */

        block = (sector_t)page->index << (PAGE_CACHE_SHIFT -
                                inode->i_blkbits);
        head = page_buffers(page);
        bh = head;

        /*
         * Get all the dirty buffers mapped to disk addresses and
         * handle any aliases from the underlying blockdev's mapping.
         */
        do {
                if (block > last_block) {
                        /*
                         * mapped buffers outside i_size will occur, because
                         * this page can be outside i_size when there is a
                         * truncate in progress.
                         */
                        /*
                         * The buffer was zeroed by block_write_full_page()
                         */
```

```
            clear_buffer_dirty(bh);
            set_buffer_uptodate(bh);
        } else if ((!buffer_mapped(bh) || buffer_delay(bh)) &&
              buffer_dirty(bh)) {
            WARN_ON(bh->b_size != blocksize);
            err = get_block(inode, block, bh, 1);
            if (err)
                goto ↓recover;
            clear_buffer_delay(bh);
            if (buffer_new(bh)) {
                /* blockdev mappings never come here */
                clear_buffer_new(bh);
                unmap_underlying_metadata(bh->b_bdev,
                            bh->b_blocknr);
            }
        }
        bh = bh->b_this_page;
        block++;
    } ?   end do ?  while (bh != head);

    do {
        if (!buffer_mapped(bh))
            continue;
        /*
         * If it's a fully non-blocking write attempt and we cannot
         * lock the buffer then redirty the page.  Note that this can
         * potentially cause a busy-wait loop from writeback threads
         * and kswapd activity, but those code paths have their own
         * higher-level throttling.
         */
        if (wbc->sync_mode != WB_SYNC_NONE || !
            wbc->nonblocking) {
            lock_buffer(bh);
        } else if (!trylock_buffer(bh)) {
            redirty_page_for_writepage(wbc, page);
            continue;
        }
        if (test_clear_buffer_dirty(bh)) {
            mark_buffer_async_write_endio(bh, handler);
        } else {
            unlock_buffer(bh);
        }
    } ?   end do ?  while ((bh = bh->b_this_page) != head);

    /*
     * The page and its buffers are protected by PageWriteback(), so we can
     * drop the bh refcounts early.
     */
    BUG_ON(PageWriteback(page));
    set_page_writeback(page);

    do {
```

```
		struct buffer_head *next = bh->b_this_page;
		if (buffer_async_write(bh)) {
			submit_bh(write_op, bh);
			nr_underway++;
		}
		bh = next;
	} while (bh != head);
	unlock_page(page);

	err = 0;
done:
	if (nr_underway == 0) {
		/*
		 * The page was marked dirty, but the buffers were
		 * clean.  Someone wrote them back by hand with
		 * ll_rw_block/submit_bh.  A rare case.
		 */
		end_page_writeback(page);

		/*
		 * The page and buffer_heads can be released at any time from
		 * here on.
		 */
	}
	return err;

recover:
	/*
	 * ENOSPC, or some other error.  We may already have added some
	 * blocks to the file, so we need to write these out to avoid
	 * exposing stale data.
	 * The page is currently locked and not marked for writeback
	 */
	bh = head;
	/* Recovery: lock and submit the mapped buffers */
	do {
		if (buffer_mapped(bh) && buffer_dirty(bh) &&
		    !buffer_delay(bh)) {
			lock_buffer(bh);
			mark_buffer_async_write_endio(bh, handler);
		} else {
			/*
			 * The buffer may have been set dirty during
			 * attachment to a dirty page.
			 */
			clear_buffer_dirty(bh);
		}
	} while ((bh = bh->b_this_page) != head);
	SetPageError(page);
	BUG_ON(PageWriteback(page));
	mapping_set_error(page->mapping, err);
	set_page_writeback(page);
	do {
```

```
            struct buffer_head *next = bh->b_this_page;
            if (buffer_async_write(bh)) {
                clear_buffer_dirty(bh);
                submit_bh(write_op, bh);
                nr_underway++;
            }
            bh = next;
        } while (bh != head);
        unlock_page(page);
        goto ↑done;
} ?   end ____block_write_full_page ?

```

submit_bh（）进而调用submit_bio（），最终将写请求放入块设备请求队列中，块设备驱动负责将缓冲区数据写入设备。关于如何从huffer head中组装bio，然后将请求放入块设备请求队列、块设备驱动如何处理请求然后唤醒等待进程，这些内容可参考Linux通用块设备层、Linux内核I/O调度层。

下面内核栈信息为将回写I/O，到SAS控制器驱动处理请求完整过程。

```
Pid: 1222, comm: flush-8:0 Tainted: G           --------------- HT 2.6.32279.debug #28
Call Trace:
 [<ffffffffa0019abb>] ? mpt2sas_base_get_smid_scsiio+0x6b/0xb0 [mpt2sas]
 [<ffffffffa001a6d7>] ? mpt2sas_base_get_msg_frame+0x57/0x60 [mpt2sas]
 [<ffffffffa00248db>] ? _scsih_qcmd+0x35b/0x9b0 [mpt2sas]
 [<ffffffff81253de3>] ? ftrace_raw_event_id_block_rq+0x153/0x190
 [<ffffffff81363591>] ? scsi_dispatch_cmd+0x101/0x360
 [<ffffffff8136b08d>] ? scsi_request_fn+0x41d/0x790
 [<ffffffff8107e0bd>] ? del_timer+0x7d/0xe0
 [<ffffffff81255601>] ? __blk_run_queue+0x31/0x40
 [<ffffffff8126e36b>] ? cfq_insert_request+0x2db/0x5b0
 [<ffffffff8124f6d1>] ? elv_insert+0xd1/0x1a0
 [<ffffffff8124f7ea>] ? __elv_add_request+0x4a/0x90
 [<ffffffff81258903>] ? __make_request+0x103/0x5a0
 [<ffffffff81254512>] ? ftrace_raw_event_id_block_bio+0xf2/0x100
 [<ffffffff81256efe>] ? generic_make_request+0x25e/0x530
 [<ffffffff8125728c>] ? submit_bio+0xbc/0x160
 [<ffffffff811acd46>] ? submit_bh+0xf6/0x150
 [<ffffffffa01097a3>] ? ext4_mb_init_cache+0x883/0x9f0 [ext4]
 [<ffffffff8112b560>] ? __lru_cache_add+0x40/0x90
 [<ffffffffa0109a2e>] ? ext4_mb_init_group+0x11e/0x210 [ext4]
 [<ffffffffa0109f85>] ? ext4_mb_load_buddy+0x355/0x390 [ext4]
 [<ffffffffa010adad>] ? ext4_mb_find_by_goal+0x6d/0x2e0 [ext4]
 [<ffffffff81256efe>] ? generic_make_request+0x25e/0x530
```

[<ffffffffa010b249>] ? ext4_mb_regular_allocator+0x59/0x410 [ext4]
[<ffffffffa0106380>] ? ext4_mb_normalize_request+0x2d0/0x480 [ext4]
[<ffffffffa010d25d>] ? ext4_mb_new_blocks+0x38d/0x560 [ext4]
[<ffffffffa0100afe>] ? ext4_ext_find_extent+0x2be/0x320 [ext4]
[<ffffffffa0103bb3>] ? ext4_ext_get_blocks+0x1113/0x1a10 [ext4]
[<ffffffff810edb54>] ? rb_reserve_next_event+0xb4/0x370
[<ffffffff810137f3>] ? native_sched_clock+0x13/0x80
[<ffffffff810edb54>] ? rb_reserve_next_event+0xb4/0x370
[<ffffffff810edb54>] ? rb_reserve_next_event+0xb4/0x370
[<ffffffffa00dfd79>] ? ext4_get_blocks+0xf9/0x2a0 [ext4]
[<ffffffff810edb54>] ? rb_reserve_next_event+0xb4/0x370
[<ffffffffa00e1c21>] ? mpage_da_map_and_submit+0xa1/0x450 [ext4]
[<ffffffff81277ef5>] ? radix_tree_gang_lookup_tag_slot+0x95/0xe0
[<ffffffff81113bd0>] ? find_get_pages_tag+0x40/0x120
[<ffffffffa00e203d>] ? mpage_add_bh_to_extent+0x6d/0xf0 [ext4]
[<ffffffffa00e238f>] ? write_cache_pages_da+0x2cf/0x470 [ext4]
[<ffffffffa00e2802>] ? ext4_da_writepages+0x2d2/0x620 [ext4]
[<ffffffff811299e1>] ? do_writepages+0x21/0x40
[<ffffffff811a500d>] ? writeback_single_inode+0xdd/0x2c0
[<ffffffff811a544e>] ? writeback_sb_inodes+0xce/0x180
[<ffffffff811a55ab>] ? writeback_inodes_wb+0xab/0x1b0
[<ffffffff811a594b>] ? wb_writeback+0x29b/0x3f0
[<ffffffff814fd9b0>] ? thread_return+0x4e/0x76e
[<ffffffff8107eb42>] ? del_timer_sync+0x22/0x30
[<ffffffff811a5c39>] ? wb_do_writeback+0x199/0x240
[<ffffffff811a5d43>] ? bdi_writeback_task+0x63/0x1b0
[<ffffffff81091f97>] ? bit_waitqueue+0x17/0xd0
[<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
[<ffffffff811386c6>] ? bdi_start_fn+0x86/0x100
[<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
[<ffffffff81091d66>] ? kthread+0x96/0xa0
[<ffffffff8100c14a>] ? child_rip+0xa/0x20
[<ffffffff81091cd0>] ? kthread+0x0/0xa0
[<ffffffff8100c140>] ? child_rip+0x0/0x20

3.9 写结束回调函数end_buffer_async_write（）

写I/O请求通过submit_bh（）下发，SAS/RAID控制器完成数据写入后，会发一个中断给CPU，通知I/O完成。下面栈信息为写I/O完成，控制器发中断，内核处理过程。

Pid: 0, comm: swapper Tainted: G --------------- HT 2.6.32279.debug #36
Call Trace:
<IRQ> [<ffffffff811ae595>] ? end_buffer_async_write+0x1a5/0x1c0

```
[<ffffffff811b2664>] ? bio_free+0x64/0x70
[<ffffffff811acdcf>] ? end_bio_bh_io_sync+0x2f/0x60
[<ffffffff811b131d>] ? bio_endio+0x1d/0x40
[<ffffffff81254d0b>] ? req_bio_endio+0x9b/0xe0
[<ffffffff812568a7>] ? blk_update_request+0x107/0x490
[<ffffffff81256c57>] ? blk_update_bidi_request+0x27/0xa0
[<ffffffff812580df>] ? blk_end_bidi_request+0x2f/0x80
[<ffffffff81258180>] ? blk_end_request+0x10/0x20
[<ffffffff8136c1bf>] ? scsi_io_completion+0xaf/0x6c0
[<ffffffff813632c2>] ? scsi_finish_command+0xc2/0x130
[<ffffffff8136c935>] ? scsi_softirq_done+0x145/0x170
[<ffffffff8125d7b5>] ? blk_done_softirq+0x85/0xa0
[<ffffffff81073ec1>] ? __do_softirq+0xc1/0x1e0
[<ffffffff810738c6>] ? ftrace_raw_event_softirq_raise+0x16/0x20
[<ffffffff8100c24c>] ? call_softirq+0x1c/0x30
[<ffffffff8100de85>] ? do_softirq+0x65/0xa0
[<ffffffff81073ca5>] ? irq_exit+0x85/0x90
[<ffffffff8102a905>] ? smp_call_function_single_interrupt+0x35/0x40
[<ffffffff8100bdb3>] ? call_function_single_interrupt+0x13/0x20
<EOI>   [<ffffffff812f7d9f>] ? acpi_idle_enter_simple+0x117/0x14b
[<ffffffff812f7d98>] ? acpi_idle_enter_simple+0x110/0x14b
[<ffffffff814077d7>] ? cpuidle_idle_call+0xa7/0x140
[<ffffffff81009e06>] ? cpu_idle+0xb6/0x110
[<ffffffff814f6e8f>] ? start_secondary+0x22a/0x26d
```

在block_write_bull_page（）处理写I/O过程中，清除buffer head的BH_Dirty标志，取而代之的是设置为BH_Async_Write，同时设置页面page标志为PG_writeback。通过PG_writeback和BH_Async_Write这两个标志，就可以判断page和buffer是否正在回写。

```
/*
 * Completion handler for block_write_full_page() - pages which are unlocked
 * during I/O, and which have PageWriteback cleared upon I/O completion.
 */
void end_buffer_async_write(struct buffer_head *bh,
                            int uptodate)
{
        char b[BDEVNAME_SIZE];
        unsigned long flags;
        struct buffer_head *first;
        struct buffer_head *tmp;
        struct page *page;

        BUG_ON(!buffer_async_write(bh));

```

```
    page = bh->b_page;
    if (uptodate) {
        set_buffer_uptodate(bh);
    } else {
        if (!quiet_error(bh)) {
            buffer_io_error(bh);
            printk(KERN_WARNING "lost page write due to "
                    "I/O error on %s\n",
                bdevname(bh->b_bdev, b));
        }
        set_bit(AS_EIO, &page->mapping->flags);
        set_buffer_write_io_error(bh);
        clear_buffer_uptodate(bh);
        SetPageError(page);
    }

    first = page_buffers(page);
    local_irq_save(flags);
    bit_spin_lock(BH_Uptodate_Lock, &first->b_state);

    clear_buffer_async_write(bh);
    unlock_buffer(bh);
    tmp = bh->b_this_page;
    while (tmp != bh) {
        if (buffer_async_write(tmp)) {
            BUG_ON(!buffer_locked(tmp));
            goto ↓still_busy;
        }
        tmp = tmp->b_this_page;
    }
    bit_spin_unlock(BH_Uptodate_Lock, &first->b_state);
    local_irq_restore(flags);
    end_page_writeback(page);
    return;

still_busy:
    bit_spin_unlock(BH_Uptodate_Lock, &first->b_state);
    local_irq_restore(flags);
    return;
} ?   end end_buffer_async_write  ?
EXPORT_SYMBOL(end_buffer_async_write);
```

写I/O完成后，调用buffer_head->b_end_io写I/O回调函数end_buffer_async_write（），该函数实现在文件fs/buffer.c中。清除buffer head标志BH_Aysnc_Write，设置为BH_Uptodate；同时清除page的PG_writeback标志。最后唤醒该页面上的等待队列，若有进程等待该page writeback完成，则要唤醒相应的进程。

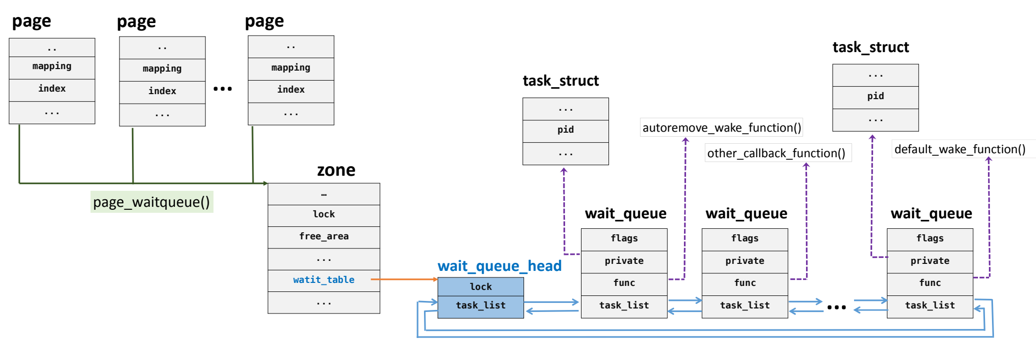

图4 页面page与I/O wait_queue

附录

栈信息补充汇总。

Pid: 1158, comm: flush-8:0 Tainted: G --------------- HT 2.6.32279.debug #34
Call Trace:
<IRQ> [<ffffffff811ae595>] ? end_buffer_async_write+0x1a5/0x1c0
[<ffffffff811b2664>] ? bio_free+0x64/0x70
[<ffffffff811acdcf>] ? end_bio_bh_io_sync+0x2f/0x60
[<ffffffff811b131d>] ? bio_endio+0x1d/0x40
[<ffffffff81254d0b>] ? req_bio_endio+0x9b/0xe0
[<ffffffff812568a7>] ? blk_update_request+0x107/0x490
[<ffffffff8107cb22>] ? ftrace_raw_event_hrtimer_start+0xc2/0xd0
[<ffffffff81256c57>] ? blk_update_**bidi**_request+0x27/0xa0
[<ffffffff8125812f>] ? blk_end_bidi_request+0x2f/0x80
[<ffffffff812581d0>] ? blk_end_request+0x10/0x20
[<ffffffff8136c20f>] ? scsi_io_completion+0xaf/0x6c0
[<ffffffff81363312>] ? scsi_finish_command+0xc2/0x130
[<ffffffff8136c985>] ? scsi_softirq_done+0x145/0x170
[<ffffffff8125d805>] ? blk_done_softirq+0x85/0xa0
[<ffffffff81073ec1>] ? __do_softirq+0xc1/0x1e0
[<ffffffff810738c6>] ? ftrace_raw_event_softirq_raise+0x16/0x20
[<ffffffff8100c24c>] ? call_softirq+0x1c/0x30
[<ffffffff8100de85>] ? do_softirq+0x65/0xa0
[<ffffffff81073ca5>] ? irq_exit+0x85/0x90

```
[<ffffffff8102a905>] ? smp_call_function_single_interrupt+0x35/0x40
[<ffffffff8100bdb3>] ? call_function_single_interrupt+0x13/0x20
<EOI>    [<ffffffff8106c621>] ? vprintk+0x251/0x560
[<ffffffff8150358a>] ? atomic_notifier_call_chain+0x1a/0x20
[<ffffffff81327b2e>] ? notify_update+0x2e/0x30
[<ffffffffa00dbe40>] ? ext4_bh_delay_or_unwritten+0x0/0x30 [ext4]
[<ffffffff814fd423>] ? printk+0x41/0x46
[<ffffffffa00dbe40>] ? ext4_bh_delay_or_unwritten+0x0/0x30 [ext4]
[<ffffffffa00dbe40>] ? ext4_bh_delay_or_unwritten+0x0/0x30 [ext4]
[<ffffffff8100f201>] ? printk_address+0x31/0x40
[<ffffffff8100f38c>] ? print_trace_address+0x3c/0x50
[<ffffffff8100f5a1>] ? print_context_stack+0xa1/0x140
[<ffffffffa00dbe40>] ? ext4_bh_delay_or_unwritten+0x0/0x30 [ext4]
[<ffffffff8100e520>] ? dump_trace+0x190/0x3b0
[<ffffffff8100f315>] ? show_trace_log_lvl+0x55/0x70
[<ffffffff8100f345>] ? show_trace+0x15/0x20
[<ffffffff814fd273>] ? dump_stack+0x6f/0x76
[<ffffffff8125723f>] ? generic_make_request+0x56f/0x580
[<ffffffff8125730c>] ? submit_bio+0xbc/0x160
[<ffffffff811acd46>] ? submit_bh+0xf6/0x150
[<ffffffff811aeae0>] ? __block_write_full_page+0x1e0/0x3b0
[<ffffffff811ae3f0>] ? end_buffer_async_write+0x0/0x1c0
[<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
[<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
[<ffffffff811af720>] ? block_write_full_page_endio+0xe0/0x120
[<ffffffffa00dbe40>] ? ext4_bh_delay_or_unwritten+0x0/0x30 [ext4]
[<ffffffff811af775>] ? block_write_full_page+0x15/0x20
[<ffffffffa00e1722>] ? ext4_writepage+0x172/0x400 [ext4]
[<ffffffffa00e1af7>] ? mpage_da_submit_io+0x147/0x1d0 [ext4]
[<ffffffffa00e1d22>] ? mpage_da_map_and_submit+0x1a2/0x450 [ext4]
[<ffffffff81277f75>] ? radix_tree_gang_lookup_tag_slot+0x95/0xe0
[<ffffffff81113bd0>] ? find_get_pages_tag+0x40/0x120
[<ffffffffa00e203d>] ? mpage_add_bh_to_extent+0x6d/0xf0 [ext4]
[<ffffffffa00e238f>] ? write_cache_pages_da+0x2cf/0x470 [ext4]
[<ffffffffa00e2802>] ? ext4_da_writepages+0x2d2/0x620 [ext4]
[<ffffffff811299e1>] ? do_writepages+0x21/0x40
[<ffffffff811a500d>] ? writeback_single_inode+0xdd/0x2c0
[<ffffffff811a544e>] ? writeback_sb_inodes+0xce/0x180
[<ffffffff811a55ab>] ? writeback_inodes_wb+0xab/0x1b0
[<ffffffff811a594b>] ? wb_writeback+0x29b/0x3f0
[<ffffffff811a5c39>] ? wb_do_writeback+0x199/0x240
[<ffffffff811a5d43>] ? bdi_writeback_task+0x63/0x1b0
[<ffffffff81091f97>] ? bit_waitqueue+0x17/0xd0
[<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
```

```
 [<ffffffff811386c6>] ? bdi_start_fn+0x86/0x100
 [<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
 [<ffffffff81091d66>] ? kthread+0x96/0xa0
 [<ffffffff8100c14a>] ? child_rip+0xa/0x20
 [<ffffffff81091cd0>] ? kthread+0x0/0xa0
 [<ffffffff8100c140>] ? child_rip+0x0/0x20


Pid: 0, comm: swapper Tainted: G           --------------- HT 2.6.32279.debug #36
Call Trace:
 <IRQ>  [<ffffffff811ae595>] ? end_buffer_async_write+0x1a5/0x1c0
 [<ffffffff811b2664>] ? bio_free+0x64/0x70
 [<ffffffff811acdcf>] ? end_bio_bh_io_sync+0x2f/0x60
 [<ffffffff811b131d>] ? bio_endio+0x1d/0x40
 [<ffffffff81254d0b>] ? req_bio_endio+0x9b/0xe0
 [<ffffffff812568a7>] ? blk_update_request+0x107/0x490
 [<ffffffff81256c57>] ? blk_update_bidi_request+/0x130
 [<ffffffff8136c935>] ? scsi_softirq_done+0x145/0x170
 [<ffffffff8125d7b5>] ? blk_done_softirq+0x85/0xa0
 [<ffffffff81073ec1>] ? __do_softirq+0xc1/0x1e0
 [<ffffffff810db896>] ? handle_IRQ_event+0xf6/0x_intr+0x0/0x11
 <EOI>  [<ffffffff812f7d9f>] ? acpi_idle_enter_simple+0x117/0x14b
 [<ffffffff812f7d98>] ? acpi_idle_enter_simple+0x110/0x14b
 [<ffffffff814077d7>] ? cpuidle_idle_call+0xa7/0x140
 [<ffffffff81009x86_64_start_kernel+0xfa/0x109



======== bio->bi_sector:100938528
Pid: 1102, comm: jbd2/sda1-8 Tainted: G          --------------- HT 2.6.32279.debug #34
Call Trace:
 [<ffffffff8125723f>] ? generic_make_request+0x56f/0x580
 [<ffffffff8125730c>] ? submit_bio+0xbc/0x160
 [<ffffffff811acd46>] ? submit_bh+0xf6/0x150
 [<ffffffffa00ace08>] ? jbd2_journal_commit_transaction+0x5b8/0x1580 [jbd2]
 [<ffffffff8107c981>] ? ftrace_raw_event_timer_cancel+0xa1/0xb0
 [<ffffffff8107eabb>] ? try_to_del_timer_sync+0x7b/0xe0
 [<ffffffffa00b3218>] ? kjournald2+0xb8/0x220 [jbd2]
 [<ffffffff810920d0>] ? autoremove_wake_function+0x0/0x40
 [<ffffffffa00b3160>] ? kjournald2+0x0/0x220 [jbd2]
 [<ffffffff81091d66>] ? kthread+0x96/0xa0
 [<ffffffff8100c14a>] ? child_rip+0xa/0x20
 [<ffffffff81091cd0>] ? kthread+0x0/0xa0
 [<ffffffff8100c140>] ? child_rip+0x0/0x20
```

```
Pid: 1102, comm: jbd2/sda1-8 Tainted: G           --------------- HT 2.6.32279.debug #34
Call Trace:
 [<ffffffff8125723f>] ? generic_make_request+0x56f/0x580
 [<ffffffff8125730c>] ? submit_bio+0xbc/0x160
 [<ffffffff811acd46>] ? submit_bh+0xf6/0x150
 [<ffffffffa00ace08>] ? jbd2_journal_commit_transaction+0x5b8/0x1580 [jbd2]
 [<ffffffff8107c981>] ? ftrace_raw_event_timer_cancel+0xa1/0xb0
 [<ffffffff8107eabb>] ? try_to_del_timer_sync+0x7b/0xe0
 [<ffffffffa00b3218>] ? kjournald2+0xb8/0x220 [jbd2]
 [<ffffffff810920d0>] ? autoremove_wake_function+0x0/0x40
 [<ffffffffa00b3160>] ? kjournald2+0x0/0x220 [jbd2]
 [<ffffffff81091d66>] ? kthread+0x96/0xa0
 [<ffffffff8100c14a>] ? child_rip+0xa/0x20
 [<ffffffff81091cd0>] ? kthread+0x0/0xa0
 [<ffffffff8100c140>] ? child_rip+0x0/0x20
 [<ffffffffa00e203d>] ? mpage_add_bh_to_extent+0x6d/0xf0 [ext4]


======== bio->bi_sector:333352
Pid: 1158, comm: flush-8:0 Tainted: G          -------------- HT 2.6.32279.debug #34
Call Trace:
 [<ffffffff8125723f>] ? generic_make_request+0x56f/0x580
 [<ffffffff8125730c>] ? submit_bio+0xbc/0x160
 [<ffffffff811acd46>] ? submit_bh+0xf6/0x150
 [<ffffffff811aeae0>] ? __block_write_full_page+0x1e0/0x3b0
 [<ffffffff811ae3f0>] ? end_buffer_async_write+0x0/0x1c0
 [<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
 [<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
 [<ffffffff811af720>] ? block_write_full_page_endio+0xe0/0x120
 [<ffffffffa00dbe40>] ? ext4_bh_delay_or_unwritten+0x0/0x30 [ext4]
 [<ffffffff811af775>] ? block_write_full_page+0x15/0x20
 [<ffffffffa00e1722>] ? ext4_writepage+0x172/0x400 [ext4]
 [<ffffffffa00e1af7>] ? mpage_da_submit_io+0x147/0x1d0 [ext4]
 [<ffffffffa00e1d22>] ? mpage_da_map_and_submit+0x1a2/0x450 [ext4]
 [<ffffffff81277f75>] ? radix_tree_gang_lookup_tag_slot+0x95/0xe0
 [<ffffffff81113bd0>] ? find_get_pages_tag+0x40/0x120


======== bio->bi_sector:333192
Pid: 1158, comm: flush-8:0 Tainted: G               --------------- HT 2.6.32279.debug #34
Call Trace:
```

```
 [<ffffffff8125723f>] ? generic_make_request+0x56f/0x580
 [<ffffffff8125730c>] ? submit_bio+0xbc/0x160
 [<ffffffff811acd46>] ? submit_bh+0xf6/0x150
 [<ffffffff811aeae0>] ? __block_write_full_page+0x1e0/0x3b0
 [<ffffffff811ae3f0>] ? end_buffer_async_write+0x0/0x1c0
 [<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
 [<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
 [<ffffffff811af720>] ? block_write_full_page_endio+0xe0/0x120
 [<ffffffffa00dbe40>] ? ext4_bh_delay_or_unwritten+0x0/0x30 [ext4]
 [<ffffffff811af775>] ? block_write_full_page+0x15/0x20
 [<ffffffffa00e1722>] ? ext4_writepage+0x172/0x400 [ext4]
 [<ffffffffa00e1af7>] ? mpage_da_submit_io+0x147/0x1d0 [ext4]
 [<ffffffffa00e1d22>] ? mpage_da_map_and_submit+0x1a2/0x450 [ext4]
 [<ffffffff81277f75>] ? radix_tree_gang_lookup_tag_slot+0x95/0xe0
 [<ffffffff81113bd0>] ? find_get_pages_tag+0x40/0x120
 [<ffffffffa00e203d>] ? mpage_add_bh_to_extent+0x6d/0xf0 [ext4]
 [<ffffffffa00e238f>] ? write_cache_pages_da+0x2cf/0x470 [ext4]
 [<ffffffffa00e2802>] ? ext4_da_writepages+0x2d2/0x620 [ext4]
 [<ffffffff811299e1>] ? do_writepages+0x21/0x40
 [<ffffffff811a500d>] ? writeback_single_inode+0xdd/0x2c0
 [<ffffffff811a544e>] ? writeback_sb_inodes+0xce/0x180
 [<ffffffff811a55ab>] ? writeback_inodes_wb+0xab/0x1b0
 [<ffffffff811a594b>] ? wb_writeback+0x29b/0x3f0
 [<ffffffff811a5c39>] ? wb_do_writeback+0x199/0x240
 [<ffffffff811a5d43>] ? bdi_writeback_task+0x63/0x1b0
 [<ffffffff81091f97>] ? bit_waitqueue+0x17/0xd0
 [<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
 [<ffffffff811386c6>] ? bdi_start_fn+0x86/0x100
 [<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
 [<ffffffff81091d66>] ? kthread+0x96/0xa0
 [<ffffffff8100c14a>] ? child_rip+0xa/0x20
 [<ffffffff81091cd0>] ? kthread+0x0/0xa0
 [<ffffffff8100c140>] ? child_rip+0x0/0x20


Pid: 0, comm: swapper Tainted: G           -------------- HT 2.6.32279.debug #35
Call Trace:
 <IRQ>  [<ffffffff811ae595>] ? end_buffer_async_write+0x1a5/0x1c0
 [<ffffffff811b2664>] ? bio_free+0x64/0x70
 [<ffffffff811acdcf>] ? end_bio_bh_io_sync+0x2f/0x60
 [<ffffffff811b131d>] ? bio_endio+0x1d/0x40
 [<ffffffff81254d0b>] ? req_bio_endio+0x9b/0xe0
 [<ffffffff812568a7>] ? blk_update_request+0x107/0x490
```

```
 [<ffffffff8125578f>] ? blk_run_queue+0x3f/0x50
 [<ffffffff81256c57>] ? blk_update_bidi_request+0x27/0xa0
 [<ffffffff812580df>] ? blk_end_bidi_request+0x2f/0x80
 [<ffffffff81258180>] ? blk_end_request+0x10/0x20
 [<ffffffff8136c1bf>] ? scsi_io_completion+0xaf/0x6c0
 [<ffffffff813632c2>] ? scsi_finish_command+0xc2/0x130
 [<ffffffff8136c935>] ? scsi_softirq_done+0x145/0x170
 [<ffffffff8125d7b5>] ? blk_done_softirq+0x85/0xa0
 [<ffffffff81073ec1>] ? __do_softirq+0xc1/0x1e0
 [<ffffffff810db896>] ? handle_IRQ_event+0xf6/0x170
 [<ffffffff8100c24c>] ? call_softirq+0x1c/0x30
 [<ffffffff8100de85>] ? do_softirq+0x65/0xa0
 [<ffffffff81073ca5>] ? irq_exit+0x85/0x90
 [<ffffffff81505ca5>] ? do_IRQ+0x75/0xf0
 [<ffffffff8100ba53>] ? ret_from_intr+0x0/0x11
 <EOI>  [<ffffffff812f7d9f>] ? acpi_idle_enter_simple+0x117/0x14b
 [<ffffffff812f7d98>] ? acpi_idle_enter_simple+0x110/0x14b
 [<ffffffff814077d7>] ? cpuidle_idle_call+0xa7/0x140
 [<ffffffff81009e06>] ? cpu_idle+0xb6/0x110
 [<ffffffff814e44ea>] ? rest_init+0x7a/0x80
 [<ffffffff81c21f7b>] ? start_kernel+0x424/0x430
 [<ffffffff81c2133a>] ? x86_64_start_reservations+0x125/0x129
 [<ffffffff81c21438>] ? x86_64_start_kernel+0xfa/0x109

Pid: 1101, comm: jbd2/sda1-8 Tainted: G        --------------- HT 2.6.32279.debug #28
Call Trace:
 [<ffffffffa0019abb>] ? mpt2sas_base_get_smid_scsiio+0x6b/0xb0 [mpt2sas]
 [<ffffffffa001a6d7>] ? mpt2sas_base_get_msg_frame+0x57/0x60 [mpt2sas]
 [<ffffffffa00248db>] ? _scsih_qcmd+0x35b/0x9b0 [mpt2sas]
 [<ffffffff81253de3>] ? ftrace_raw_event_id_block_rq+0x153/0x190
 [<ffffffff81363591>] ? scsi_dispatch_cmd+0x101/0x360
 [<ffffffff8136b08d>] ? scsi_request_fn+0x41d/0x790
 [<ffffffff8107e0bd>] ? del_timer+0x7d/0xe0
 [<ffffffff811140c0>] ? sync_page+0x0/0x50
 [<ffffffff812557a2>] ? __generic_unplug_device+0x32/0x40
 [<ffffffff812557de>] ? generic_unplug_device+0x2e/0x50
 [<ffffffff81250324>] ? blk_unplug+0x34/0x70
 [<ffffffff81250372>] ? blk_backing_dev_unplug+0x12/0x20
 [<ffffffff811ac57e>] ? block_sync_page+0x3e/0x50
 [<ffffffff811140f8>] ? sync_page+0x38/0x50
 [<ffffffff814fe9aa>] ? __wait_on_bit_lock+0x5a/0xc0
 [<ffffffff81114097>] ? __lock_page+0x67/0x70
 [<ffffffff81092110>] ? wake_bit_function+0x0/0x50
 [<ffffffff8112a835>] ? pagevec_lookup_tag+0x25/0x40
```

```
 [<ffffffff81129882>] ? write_cache_pages+0x392/0x4a0
 [<ffffffff81128310>] ? __writepage+0x0/0x40
 [<ffffffff811299b4>] ? generic_writepages+0x24/0x30
 [<ffffffffa00ac6b7>] ? journal_submit_inode_data_buffers+0x47/0x50 [jbd2]
 [<ffffffffa00acbdd>] ? jbd2_journal_commit_transaction+0x38d/0x1580 [jbd2]
 [<ffffffff810920d0>] ? autoremove_wake_function+0x0/0x40
 [<ffffffff8107eabb>] ? try_to_del_timer_sync+0x7b/0xe0
 [<ffffffffa00b3218>] ? kjournald2+0xb8/0x220 [jbd2]
 [<ffffffff810920d0>] ? autoremove_wake_function+0x0/0x40
 [<ffffffffa00b3160>] ? kjournald2+0x0/0x220 [jbd2]
 [<ffffffff81091d66>] ? kthread+0x96/0xa0
 [<ffffffff8100c14a>] ? child_rip+0xa/0x20
 [<ffffffff81091cd0>] ? kthread+0x0/0xa0
 [<ffffffff8100c140>] ? child_rip+0x0/0x20


Pid: 1158, comm: flush-8:0 Tainted: G           --------------- HT 2.6.32279.debug #34
Call Trace:
 [<ffffffff8125723f>] ? generic_make_request+0x56f/0x580
 [<ffffffff8125730c>] ? submit_bio+0xbc/0x160
 [<ffffffff811acd46>] ? submit_bh+0xf6/0x150
 [<ffffffff811aeae0>] ? __block_write_full_page+0x1e0/0x3b0
 [<ffffffff811ae3f0>] ? end_buffer_async_write+0x0/0x1c0
 [<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
 [<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
 [<ffffffff811af720>] ? block_write_full_page_endio+0xe0/0x120
 [<ffffffffa00dbe40>] ? ext4_bh_delay_or_unwritten+0x0/0x30 [ext4]
 [<ffffffff811af775>] ? block_write_full_page+0x15/0x20
 [<ffffffffa00e1722>] ? ext4_writepage+0x172/0x400 [ext4]
 [<ffffffffa00e1af7>] ? mpage_da_submit_io+0x147/0x1d0 [ext4]
 [<ffffffffa00e1d22>] ? mpage_da_map_and_submit+0x1a2/0x450 [ext4]
 [<ffffffff81277f75>] ? radix_tree_gang_lookup_tag_slot+0x95/0xe0
 [<ffffffff81113bd0>] ? find_get_pages_tag+0x40/0x120
 [<ffffffffa00e203d>] ? mpage_add_bh_to_extent+0x6d/0xf0 [ext4]
 [<ffffffffa00e238f>] ? write_cache_pages_da+0x2cf/0x470 [ext4]
 [<ffffffffa00e2802>] ? ext4_da_writepages+0x2d2/0x620 [ext4]
 [<ffffffff811299e1>] ? do_writepages+0x21/0x40
 [<ffffffff811a500d>] ? writeback_single_inode+0xdd/0x2c0
 [<ffffffff811a544e>] ? writeback_sb_inodes+0xce/0x180
 [<ffffffff811a55ab>] ? writeback_inodes_wb+0xab/0x1b0
 [<ffffffff811a594b>] ? wb_writeback+0x29b/0x3f0
 [<ffffffff811a5c39>] ? wb_do_writeback+0x199/0x240
 [<ffffffff811a5d43>] ? bdi_writeback_task+0x63/0x1b0
```

```
 [<ffffffff81091f97>] ? bit_waitqueue+0x17/0xd0
 [<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
 [<ffffffff811386c6>] ? bdi_start_fn+0x86/0x100
 [<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
 [<ffffffff81091d66>] ? kthread+0x96/0xa0
 [<ffffffff8100c14a>] ? child_rip+0xa/0x20
 [<ffffffff81091cd0>] ? kthread+0x0/0xa0
 [<ffffffff8100c140>] ? child_rip+0x0/0x20


[<ffffffff8135cf57>] ? brd_make_request+0x477/0x550
 [<ffffffff814fd603>] ? printk+0x41/0x46
 [<ffffffff81257024>] ? generic_make_request+0x2c4/0x5b0
 [<ffffffff812573cc>] ? submit_bio+0xbc/0x160
 [<ffffffff811acdb8>] ? submit_bh+0x108/0x210
 [<ffffffff81129d39>] ? test_set_page_writeback+0xe9/0x1a0
 [<ffffffff811aeb70>] ? __block_write_full_page+0x1e0/0x3b0
 [<ffffffff811ae4b0>] ? end_buffer_async_write+0x0/0x190
 [<ffffffff811b3430>] ? blkdev_get_block+0x0/0x70
 [<ffffffff811b3430>] ? blkdev_get_block+0x0/0x70
 [<ffffffff811af7b0>] ? block_write_full_page_endio+0xe0/0x120
 [<ffffffff81113bd0>] ? find_get_pages_tag+0x40/0x120
 [<ffffffff811af805>] ? block_write_full_page+0x15/0x20
 [<ffffffff811b4418>] ? blkdev_writepage+0x18/0x20
 [<ffffffff81128327>] ? __writepage+0x17/0x40
 [<ffffffff811296b9>] ? write_cache_pages+0x1c9/0x4a0
 [<ffffffff81128310>] ? __writepage+0x0/0x40
 [<ffffffff811299b4>] ? generic_writepages+0x24/0x30
 [<ffffffff811299e1>] ? do_writepages+0x21/0x40
 [<ffffffff811a500d>] ? writeback_single_inode+0xdd/0x2c0
 [<ffffffff811a544e>] ? writeback_sb_inodes+0xce/0x180
 [<ffffffff811a55ab>] ? writeback_inodes_wb+0xab/0x1b0
 [<ffffffff811a594b>] ? wb_writeback+0x29b/0x3f0
 [<ffffffff814fdc10>] ? thread_return+0x4e/0x76e
 [<ffffffff8107eb42>] ? del_timer_sync+0x22/0x30
 [<ffffffff811a5c39>] ? wb_do_writeback+0x199/0x240
 [<ffffffff811a5d43>] ? bdi_writeback_task+0x63/0x1b0
 [<ffffffff81091f97>] ? bit_waitqueue+0x17/0xd0
 [<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
 [<ffffffff811386c6>] ? bdi_start_fn+0x86/0x100
 [<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
 [<ffffffff81091d66>] ? kthread+0x96/0xa0
 [<ffffffff8100c14a>] ? child_rip+0xa/0x20
```

```
Pid: 1101, comm: jbd2/sda1-8 Tainted: G        --------------- HT 2.6.32279.debug #28
Call Trace:
 [<ffffffffa0019abb>] ? mpt2sas_base_get_smid_scsiio+0x6b/0xb0 [mpt2sas]
 [<ffffffffa001a6d7>] ? mpt2sas_base_get_msg_frame+0x57/0x60 [mpt2sas]
 [<ffffffffa00246fe>] ? _scsih_qcmd+0x17e/0x9b0 [mpt2sas]
 [<ffffffff81253de3>] ? ftrace_raw_event_id_block_rq+0x153/0x190
 [<ffffffff81363591>] ? scsi_dispatch_cmd+0x101/0x360
 [<ffffffff8136b08d>] ? scsi_request_fn+0x41d/0x790
 [<ffffffff8107e0bd>] ? del_timer+0x7d/0xe0
 [<ffffffff81255601>] ? __blk_run_queue+0x31/0x40
 [<ffffffff8126e36b>] ? cfq_insert_request+0x2db/0x5b0
 [<ffffffff8124f6d1>] ? elv_insert+0xd1/0x1a0
 [<ffffffff8124f7ea>] ? __elv_add_request+0x4a/0x90
 [<ffffffff81258903>] ? __make_request+0x103/0x5a0
 [<ffffffff81254512>] ? ftrace_raw_event_id_block_bio+0xf2/0x100
 [<ffffffff81256efe>] ? generic_make_request+0x25e/0x530
 [<ffffffff811ad539>] ? __find_get_block+0xa9/0x200
 [<ffffffff8125728c>] ? submit_bio+0xbc/0x160
 [<ffffffff811acd46>] ? submit_bh+0xf6/0x150
 [<ffffffffa00ac835>] ? journal_submit_commit_record+0x135/0x150 [jbd2]
 [<ffffffffa00ad385>] ? jbd2_journal_commit_transaction+0xb35/0x1580 [jbd2]
 [<ffffffff810920d0>] ? autoremove_wake_function+0x0/0x40
 [<ffffffff8107eabb>] ? try_to_del_timer_sync+0x7b/0xe0
 [<ffffffffa00b3218>] ? kjournald2+0xb8/0x220 [jbd2]
 [<ffffffff810920d0>] ? autoremove_wake_function+0x0/0x40
 [<ffffffffa00b3160>] ? kjournald2+0x0/0x220 [jbd2]
 [<ffffffff81091d66>] ? kthread+0x96/0xa0
 [<ffffffff8100c14a>] ? child_rip+0xa/0x20
 [<ffffffff81091cd0>] ? kthread+0x0/0xa0
 [<ffffffff8100c140>] ? child_rip+0x0/0x20


Pid: 0, comm: swapper Tainted: G          --------------- HT 2.6.32279.debug #28
Call Trace:
 <IRQ>  [<ffffffffa0019abb>] ? mpt2sas_base_get_smid_scsiio+0x6b/0xb0 [mpt2sas]
 [<ffffffffa001a6d7>] ? mpt2sas_base_get_msg_frame+0x57/0x60 [mpt2sas]
 [<ffffffffa00248db>] ? _scsih_qcmd+0x35b/0x9b0 [mpt2sas]
 [<ffffffff81253de3>] ? ftrace_raw_event_id_block_rq+0x153/0x190
 [<ffffffff81363591>] ? scsi_dispatch_cmd+0x101/0x360
 [<ffffffff8136b08d>] ? scsi_request_fn+0x41d/0x790
 [<ffffffff8108ce4f>] ? queue_work+0x1f/0x30
 [<ffffffff812548a5>] ? kblockd_schedule_work+0x15/0x20
 [<ffffffff81255601>] ? __blk_run_queue+0x31/0x40
 [<ffffffff81255750>] ? blk_run_queue+0x30/0x50
```

```
 [<ffffffff8136a4e7>] ? scsi_run_queue+0xd7/0x290
 [<ffffffff81363ef0>] ? __scsi_put_command+0x60/0xa0
 [<ffffffff8136b612>] ? scsi_next_command+0x42/0x60
 [<ffffffff8136c38e>] ? scsi_io_completion+0x2ae/0x6c0
 [<ffffffff81363292>] ? scsi_finish_command+0xc2/0x130
 [<ffffffff8136c905>] ? scsi_softirq_done+0x145/0x170
 [<ffffffff8125d785>] ? blk_done_softirq+0x85/0xa0
 [<ffffffff81073ec1>] ? __do_softirq+0xc1/0x1e0
 [<ffffffff810738c6>] ? ftrace_raw_event_softirq_raise+0x16/0x20
 [<ffffffff8100c24c>] ? call_softirq+0x1c/0x30
 [<ffffffff8100de85>] ? do_softirq+0x65/0xa0
 [<ffffffff81073ca5>] ? irq_exit+0x85/0x90
 [<ffffffff8102a905>] ? smp_call_function_single_interrupt+0x35/0x40
 [<ffffffff8100bdb3>] ? call_function_single_interrupt+0x13/0x20
 <EOI>  [<ffffffff812f7d6f>] ? acpi_idle_enter_simple+0x117/0x14b
 [<ffffffff812f7d68>] ? acpi_idle_enter_simple+0x110/0x14b
 [<ffffffff814077a7>] ? cpuidle_idle_call+0xa7/0x140
 [<ffffffff81009e06>] ? cpu_idle+0xb6/0x110
 [<ffffffff814f6e5f>] ? start_secondary+0x22a/0x26d


Pid: 0, comm: swapper Tainted: G           --------------- HT 2.6.32279.debug #28
Call Trace:
 <IRQ>  [<ffffffffa001a6d7>] ? mpt2sas_base_get_msg_frame+0x57/0x60 [mpt2sas]
 [<ffffffffa0027990>] ? _scsih_io_done+0xa0/0xd90 [mpt2sas]
 [<ffffffff810137f3>] ? native_sched_clock+0x13/0x80
 [<ffffffff810edb54>] ? rb_reserve_next_event+0xb4/0x370
 [<ffffffffa001cfc3>] ? _base_interrupt+0x1e3/0x8f0 [mpt2sas]
 [<ffffffff810f31d3>] ? trace_nowake_buffer_unlock_commit+0x43/0x60
 [<ffffffff810740c3>] ? ftrace_raw_event_irq_handler_entry+0xe3/0xf0
 [<ffffffff810db800>] ? handle_IRQ_event+0x60/0x170
 [<ffffffff81073f1f>] ? __do_softirq+0x11f/0x1e0
 [<ffffffff810ddf8e>] ? handle_edge_irq+0xde/0x180
 [<ffffffff8100df09>] ? handle_irq+0x49/0xa0
 [<ffffffff81505c6c>] ? do_IRQ+0x6c/0xf0
 [<ffffffff8100ba53>] ? ret_from_intr+0x0/0x11
 <EOI>  [<ffffffff812f7d6f>] ? acpi_idle_enter_simple+0x117/0x14b
 [<ffffffff812f7d68>] ? acpi_idle_enter_simple+0x110/0x14b
 [<ffffffff814077a7>] ? cpuidle_idle_call+0xa7/0x140
 [<ffffffff81009e06>] ? cpu_idle+0xb6/0x110
 [<ffffffff814e44ba>] ? rest_init+0x7a/0x80
 [<ffffffff81c21f7b>] ? start_kernel+0x424/0x430
 [<ffffffff81c2133a>] ? x86_64_start_reservations+0x125/0x129
 [<ffffffff81c21438>] ? x86_64_start_kernel+0xfa/0x109
```

```
 [<ffffffffa001a6d7>] ? mpt2sas_base_get_msg_frame+0x57/0x60 [mpt2sas]
 [<ffffffffa00246fe>] ? _scsih_qcmd+0x17e/0x9b0 [mpt2sas]
 [<ffffffff81253de3>] ? ftrace_raw_event_id_block_rq+0x153/0x190
 [<ffffffff81363591>] ? scsi_dispatch_cmd+0x101/0x360
 [<ffffffff8136b08d>] ? scsi_request_fn+0x41d/0x790
 [<ffffffff8108ce4f>] ? queue_work+0x1f/0x30
 [<ffffffff812548a5>] ? kblockd_schedule_work+0x15/0x20
 [<ffffffff81255601>] ? __blk_run_queue+0x31/0x40
 [<ffffffff81255750>] ? blk_run_queue+0x30/0x50
 [<ffffffff8136a4e7>] ? scsi_run_queue+0xd7/0x290
 [<ffffffff81363ef0>] ? __scsi_put_command+0x60/0xa0
 [<ffffffff8136b612>] ? scsi_next_command+0x42/0x60
 [<ffffffff8136c38e>] ? scsi_io_completion+0x2ae/0x6c0
 [<ffffffff81363292>] ? scsi_finish_command+0xc2/0x130
 [<ffffffff8136c905>] ? scsi_softirq_done+0x145/0x170
 [<ffffffff8125d785>] ? blk_done_softirq+0x85/0xa0
 [<ffffffff81073ec1>] ? __do_softirq+0xc1/0x1e0
 [<ffffffff810738c6>] ? ftrace_raw_event_softirq_raise+0x16/0x20
 [<ffffffff8100c24c>] ? call_softirq+0x1c/0x30
 [<ffffffff8100de85>] ? do_softirq+0x65/0xa0
 [<ffffffff81073ca5>] ? irq_exit+0x85/0x90
 [<ffffffff8102a905>] ? smp_call_function_single_interrupt+0x35/0x40
 [<ffffffff8100bdb3>] ? call_function_single_interrupt+0x13/0x20
 <EOI>  [<ffffffff812f7d6f>] ? acpi_idle_enter_simple+0x117/0x14b
 [<ffffffff812f7d68>] ? acpi_idle_enter_simple+0x110/0x14b
 [<ffffffff814077a7>] ? cpuidle_idle_call+0xa7/0x140
 [<ffffffff81009e06>] ? cpu_idle+0xb6/0x110
 [<ffffffff814f6e5f>] ? start_secondary+0x22a/0x26d


Pid: 0, comm: swapper Tainted: G           --------------- HT 2.6.32279.debug #28
Call Trace:
 <IRQ>  [<ffffffffa001a6d7>] ? mpt2sas_base_get_msg_frame+0x57/0x60 [mpt2sas]
 [<ffffffffa0027990>] ? _scsih_io_done+0xa0/0xd90 [mpt2sas]
 [<ffffffff810137f3>] ? native_sched_clock+0x13/0x80
 [<ffffffff810edb54>] ? rb_reserve_next_event+0xb4/0x370
 [<ffffffffa001cfc3>] ? _base_interrupt+0x1e3/0x8f0 [mpt2sas]
 [<ffffffff810f31d3>] ? trace_nowake_buffer_unlock_commit+0x43/0x60
 [<ffffffff810740c3>] ? ftrace_raw_event_irq_handler_entry+0xe3/0xf0
 [<ffffffff810db800>] ? handle_IRQ_eventart_kernel+0xfa/0x109
 [<ffffffffa001a6d7>] ? mpt2sas_base_get_msg_frame+0x57/0x60 [mpt2sas]
 [<ffffffffa00246fe>] ? _scsih_qcmd+0x17e/0x9b0 [mpt2sas]
 [<ffffffff81253de3>] ? ftrace_raw_event_id_block_rq+0x153/0x190
```

```
[<ffffffff81363591>] ? scsi_dispatch_cmd+0x101/0x360
[<ffffffff8136b08d>] ? scsi_request_fn+0x41d/0x790
[<ffffffff811adfc0>] ? sync_buffer+0x0/0x50
[<ffffffff8107e0bd>] ? del_timer+0x7d/0xe0
[<ffffffff811adfc0>] ? sync_buffer+0x0/0x50
[<ffffffff812557a2>] ? __generic_unplug_device+0x32/0x40
[<ffffffff812557de>] ? generic_unplug_device+0x2e/0x50
[<ffffffff81250324>] ? blk_unplug+0x34/0x70
[<ffffffff81250372>] ? blk_backing_dev_unplug+0x12/0x20
[<ffffffff811adffb>] ? sync_buffer+0x3b/0x50
[<ffffffff814feaff>] ? __wait_on_bit+0x5f/0x90
[<ffffffff811adfc0>] ? sync_buffer+0x0/0x50
[<ffffffff814feba8>] ? out_of_line_wait_on_bit+0x78/0x90
[<ffffffff81092110>] ? wake_bit_function+0x0/0x50
[<ffffffff811adfb6>] ? __wait_on_buffer+0x26/0x30
[<ffffffffa0109154>] ? ext4_mb_init_cache+0x234/0x9f0 [ext4]
[<ffffffff8112b560>] ? __lru_cache_add+0x40/0x90
[<ffffffffa0109a2e>] ? ext4_mb_init_group+0x11e/0x210 [ext4]
[<ffffffffa0109bed>] ? ext4_mb_good_group+0xcd/0x110 [ext4]
[<ffffffffa010b38b>] ? ext4_mb_regular_allocator+0x19b/0x410 [ext4]
[<ffffffffa010d25d>] ? ext4_mb_new_blocks+0x38d/0x560 [ext4]
[<ffffffffa0100afe>] ? ext4_ext_find_extent+0x2be/0x320 [ext4]
[<ffffffffa0103bb3>] ? ext4_ext_get_blocks+0x1113/0x1a10 [ext4]
[<ffffffff810edb54>] ? rb_reserve_next_event+0xb4/0x370
[<ffffffff810137f3>] ? native_sched_clock+0x13/0x80
[<ffffffff810edb54>] ? rb_reserve_next_event+0xb4/0x370
[<ffffffff810edb54>] ? rb_reserve_next_event+0xb4/0x370
[<ffffffffa00dfd79>] ? ext4_get_blocks+0xf9/0x2a0 [ext4]
[<ffffffff810edb54>] ? rb_reserve_next_event+0xb4/0x370
[<ffffffffa00e1c21>] ? mpage_da_map_and_submit+0xa1/0x450 [ext4]
[<ffffffff81277ef5>] ? radix_tree_gang_lookup_tag_slot+0x95/0xe0
[<ffffffff81113bd0>] ? find_get_pages_tag+0x40/0x120
[<ffffffffa00e203d>] ? mpage_add_bh_to_extent+0x6d/0xf0 [ext4]
[<ffffffffa00e238f>] ? write_cache_pages_da+0x2cf/0x470 [ext4]
[<ffffffffa00e2802>] ? ext4_da_writepages+0x2d2/0x620 [ext4]
[<ffffffff811299e1>] ? do_writepages+0x21/0x40
[<ffffffff811a500d>] ? writeback_single_inode+0xdd/0x2c0
[<ffffffff811a544e>] ? writeback_sb_inodes+0xce/0x180
[<ffffffff811a55ab>] ? writeback_inodes_wb+0xab/0x1b0
[<ffffffff811a594b>] ? wb_writeback+0x29b/0x3f0
[<ffffffff814fd9b0>] ? thread_return+0x4e/0x76e
[<ffffffff8107eb42>] ? del_timer_sync+0x22/0x30
[<ffffffff811a5c39>] ? wb_do_writeback+0x199/0x240
[<ffffffff811a5d43>] ? bdi_writeback_task+0x63/0x1b0
```

```
 [<ffffffff81091f97>] ? bit_waitqueue+0x17/0xd0
 [<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
 [<ffffffff811386c6>] ? bdi_start_fn+0x86/0x100
 [<ffffffff81138640>] ? bdi_start_fn+0x0/0x100
 [<ffffffff81091d66>] ? kthread+0x96/0xa0
 [<ffffffff8100c14a>] ? child_rip+0xa/0x20
 [<ffffffff81091cd0>] ? kthread+0x0/0xa0
 [<ffffffff8100c140>] ? child_rip+0x0/0x20



Pid: 4194, comm: jbd2/sda5-8 Tainted: G        --------------- HT 2.6.32279.debug #36
Call Trace:
 <IRQ>  [<ffffffff811ae595>] ? end_buffer_async_write+0x1a5/0x1c0
 [<ffffffff811b2664>] ? bio_free+0x64/0x70
 [<ffffffff811acdcf>] ? end_bio_bh_io_sync+0x2f/0x60
 [<ffffffff811b131d>] ? bio_endio+0x1d/0x40
 [<ffffffff81254d0b>] ? req_bio_endio+0x9b/0xe0
 [<ffffffff812568a7>] ? blk_update_request+0x107/0x490
 [<ffffffff8125578f>] ? blk_run_queue+0x3f/0x50
 [<ffffffff81256c57>] ? blk_update_bidi_request+0x27/0xa0
 [<ffffffff812580df>] ? blk_end_bidi_request+0x2f/0x80
 [<ffffffff81258180>] ? blk_end_request+0x10/0x20
 [<ffffffff8136c1bf>] ? scsi_io_completion+0xaf/0x6c0
 [<ffffffff813632c2>] ? scsi_finish_command+0xc2/0x130
 [<ffffffff8136c935>] ? scsi_softirq_done+0x145/0x170
 [<ffffffff8125d7b5>] ? blk_done_softirq+0x85/0xa0
 [<ffffffff81073ec1>] ? __do_softirq+0xc1/0x1e0
 [<ffffffff810db896>] ? handle_IRQ_event+0xf6/0x170
 [<ffffffff8100c24c>] ? call_softirq+0x1c/0x30
 [<ffffffff8100de85>] ? do_softirq+0x65/0xa0
 [<ffffffff81073ca5>] ? irq_exit+0x85/0x90
 [<ffffffff81505ca5>] ? do_IRQ+0x75/0xf0
 [<ffffffff8100ba53>] ? ret_from_intr+0x0/0x11
 <EOI>  [<ffffffff81500317>] ? _spin_unlock_irqrestore+0x17/0x20
 [<ffffffff81129d27>] ? test_set_page_writeback+0xd7/0x1a0
 [<ffffffff811ae3f0>] ? end_buffer_async_write+0x0/0x1c0
 [<ffffffff811aeaaf>] ? __block_write_full_page+0x1af/0x3b0
 [<ffffffff811ae3f0>] ? end_buffer_async_write+0x0/0x1c0
 [<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
 [<ffffffffa00e0460>] ? noalloc_get_block_write+0x0/0x60 [ext4]
 [<ffffffff811af720>] ? block_write_full_page_endio+0xe0/0x120
 [<ffffffffa00dbe40>] ? ext4_bh_delay_or_unwritten+0x0/0x30 [ext4]
 [<ffffffff811af775>] ? block_write_full_page+0x15/0x20
```

```
[<ffffffffa00e1722>] ? ext4_writepage+0x172/0x400 [ext4]
[<ffffffff81128327>] ? __writepage+0x17/0x40
[<ffffffff811296b9>] ? write_cache_pages+0x1c9/0x4a0
[<ffffffff81128310>] ? __writepage+0x0/0x40
[<ffffffff811299b4>] ? generic_writepages+0x24/0x30
[<ffffffffa00ac6b7>] ? journal_submit_inode_data_buffers+0x47/0x50 [jbd2]
[<ffffffffa00acbdd>] ? jbd2_journal_commit_transaction+0x38d/0x1580 [jbd2]
[<ffffffff810f31d3>] ? trace_nowake_buffer_unlock_commit+0x43/0x60
[<ffffffff810096f0>] ? __switch_to+0xd0/0x320
[<ffffffff814fd9e0>] ? thread_return+0x4e/0x76e
[<ffffffff8107e00c>] ? lock_timer_base+0x3c/0x70
[<ffffffff8107eabb>] ? try_to_del_timer_sync+0x7b/0xe0
[<ffffffffa00b3218>] ? kjournald2+0xb8/0x220 [jbd2]
[<ffffffff810920d0>] ? autoremove_wake_function+0x0/0x40
[<ffffffffa00b3160>] ? kjournald2+0x0/0x220 [jbd2]
[<ffffffff81091d66>] ? kthread+0x96/0xa0
[<ffffffff8100c14a>] ? child_rip+0xa/0x20
[<ffffffff81091cd0>] ? kthread+0x0/0xa0
[<ffffffff8100c140>] ? child_rip+0x0/0x20
```