
Subject: [RFC][PATCH] EXT3: problem with page fault inside a transaction
Posted by [Dmitriy Monakhov](#) on Thu, 12 Oct 2006 05:57:26 GMT
[View Forum Message](#) <> [Reply to Message](#)

While reading Andrew's generic_file_buffered_write patches i've remembered one more EXT3 issue. journal_start() in prepare_write() causes different ranking violations if copy_from_user() triggers a page fault. It could cause GFP_FS allocation, re-entering into ext3 code possibly with a different superblock and journal, ranking violation of journalling serialization and mmap_sem and page lock and all other kinds of funny consequences. Our customers complain about this issue.

Andrey Savochkin already discussed the issue in lkml.
(subj EXT3: problem with copy_from_user inside a transaction)
as a result we slightly changed prepare_write()/ commit_write() semantic.

To eliminate this possibility we read in not up-to-date buffers in prepare_write(), and do the rest including hole instantiation and inode extension in commit_write(). ENOSPC case must be specifically handled.

- 1) we ought to have similar error handling block as we have after prepare_write() because we may have instantiated a few blocks outside i_size.
- 2) We cant use block_prepare_write() for allocation needs inside commit_write() due to page cache already contains valid data.

Here is the patch moving journal_start() together with space allocation from ext3_prepare_write() to commit_write().

Reorganization of ext3_prepare_write/ext3_commit_write to eliminate the possibility of the page fault in between, inside a transaction.

It could cause GFP_FS allocation, re-entering into ext3 code possibly with a different superblock and journal, ranking violation of journalling serialization and mmap_sem and page lock and all other kinds of funny consequences.

signed-off-by: Dmitriy Monakhov <dmonakhov@openvz.org>

```
fs/buffer.c    | 5 +
fs/ext3/inode.c | 194 +++++
mm/filemap.c   | 42 +++++
3 files changed, 207 insertions(+), 34 deletions(-)
```

```
diff --git a/fs/buffer.c b/fs/buffer.c
```

```
index eeb8ac1..11bf84c 100644
```

```
--- a/fs/buffer.c
```

```
+++ b/fs/buffer.c
```

```
@@ -1768,8 +1768,9 @@ static int __block_prepare_write(struct
    if (err)
```

```

        break;
    if (buffer_new(bh)) {
-   unmap_underlying_metadata(bh->b_bdev,
-   bh->b_blocknr);
+   if (buffer_mapped(bh))
+   unmap_underlying_metadata(bh->b_bdev,
+   bh->b_blocknr);
        if (PageUptodate(page)) {
            set_buffer_uptodate(bh);
            continue;
diff --git a/fs/ext3/inode.c b/fs/ext3/inode.c
index 03ba5bc..7eafd0a 100644
--- a/fs/ext3/inode.c
+++ b/fs/ext3/inode.c
@@ -1122,6 +1122,12 @@ static int walk_page_buffers( handle_t *
 * and the commit_write(). So doing the journal_start at the start of
 * prepare_write() is the right place.
 *
+ * [2004/09/04 SAW] journal_start() in prepare_write() causes different ranking
+ * violations if copy_from_user() triggers a page fault (mmap_sem, may be page
+ * lock, plus __GFP_FS allocations).
+ * Now we read in not up-to-date buffers in prepare_write(), and do the rest
+ * including hole instantiation and inode extension in commit_write().
+ *
 * Also, this function can nest inside ext3_writepage() ->
 * block_write_full_page(). In that case, we *know* that ext3_writepage()
 * has generated enough buffer credits to do the whole page. So we won't
@@ -1140,6 +1146,72 @@ static int walk_page_buffers( handle_t *
 * is elevated. We'll still have enough credits for the tiny quotafile
 * write.
 */
+
+static int ext3_get_block_delay(struct inode *inode, sector_t iblock,
+ struct buffer_head *bh_result, int create)
+{
+ int ret;
+
+ ret = ext3_get_blocks_handle(NULL, inode, iblock,
+ 1, bh_result, 0, 0);
+ if (ret > 0) {
+ bh_result->b_size = (ret << inode->i_blkbits);
+ ret = 0;
+ }
+ if (ret)
+ return ret;
+ if (!buffer_mapped(bh_result))
+ set_buffer_new(bh_result);
+ return ret;

```

```

+}
+
+/*
+ * Here we make sure that the set of buffers written at once either
+ * is fully mapped at the start, or is fully not mapped and will be allocated
+ * at commit_write time.
+ */
+static int ext3_check_block_mapping(struct page *page,
+ unsigned from, unsigned to)
+{
+ struct buffer_head *bh, *head, *next;
+ unsigned block_start, block_end;
+ unsigned blocksize;
+ int mapping, c;
+
+ head = page_buffers(page);
+ blocksize = head->b_size;
+ if (blocksize == PAGE_SIZE)
+ return 0;
+
+ mapping = -1;
+ for ( bh = head, block_start = 0;
+ bh != head || !block_start;
+ block_start = block_end, bh = next)
+ {
+ next = bh->b_this_page;
+ block_end = block_start + blocksize;
+ if (block_end <= from || block_start >= to)
+ continue;
+ c = !!buffer_mapped(bh);
+ if (mapping < 0)
+ mapping = c;
+ else if (mapping != c)
+ return block_start - from;
+ }
+ return 0;
+}
+
+static int ext3_prepare_write(struct file *file, struct page *page,
+ unsigned from, unsigned to)
+{
+ int ret;
+
+ ret = block_prepare_write(page, from, to, ext3_get_block_delay);
+ if (ret)
+ return ret;
+ return ext3_check_block_mapping(page, from, to);
+}

```

```

+
+ static int do_journal_get_write_access(handle_t *handle,
+     struct buffer_head *bh)
+ {
@@ -1148,8 +1220,81 @@ static int do_journal_get_write_access(h
+     return ext3_journal_get_write_access(handle, bh);
+ }

-static int ext3_prepare_write(struct file *file, struct page *page,
-     unsigned from, unsigned to)
+/*
+ * This function zeroes buffers not mapped to disk.
+ * The purpose of it is the same as of error recovery in
+ * __block_prepare_write(), to avoid keeping garbage in the page cache.
+ * The code is repeated here since on-disk space is now allocated from
+ * commit_write, not from prepare_write.
+ *
+ * This function is called only for the range of freshly allocated buffers,
+ * and they can be cleared without reservations.
+ */
+static void ext3_rollback_write(struct page *page,
+     unsigned from, unsigned to)
+{
+ struct buffer_head *bh, *head, *next;
+ unsigned block_start, block_end;
+ unsigned blocksize;
+ void *kaddr;
+
+ kaddr = kmap_atomic(page, KM_USER0);
+ memset(kaddr + from, 0, to - from);
+ flush_dcache_page(page);
+ kunmap_atomic(kaddr, KM_USER0);
+
+ head = page_buffers(page);
+ blocksize = head->b_size;
+ for ( bh = head, block_start = 0;
+     bh != head || !block_start;
+     block_start = block_end, bh = next)
+ {
+     next = bh->b_this_page;
+     block_end = block_start + blocksize;
+     if (block_end <= from || block_start >= to)
+         continue;
+     set_buffer_uptodate(bh);
+     if (buffer_mapped(bh))
+         mark_buffer_dirty(bh);
+ }
+}

```

```

+
+static int ext3_do_map_write(struct inode *inode, struct page *page,
+ unsigned from, unsigned to)
+{
+ struct buffer_head *bh, *head, *next;
+ unsigned block_start, block_end;
+ unsigned blocksize, bbits;
+ sector_t block;
+ int err;
+
+ head = page_buffers(page);
+ bbits = inode->i_blkbits;
+ blocksize = 1 << bbits;
+ block = (sector_t)page->index << (PAGE_CACHE_SHIFT - bbits);
+ for ( bh = head, block_start = 0;
+   bh != head || !block_start;
+   block++, block_start = block_end, bh = next)
+ {
+   next = bh->b_this_page;
+   block_end = block_start + blocksize;
+   if (block_end <= from || block_start >= to)
+     continue;
+   if (buffer_mapped(bh))
+     continue;
+   err = ext3_get_block(inode, block, bh, 1);
+   if (err)
+     return err;
+   /* buffer must be new and mapped here */
+   clear_buffer_new(bh);
+   unmap_underlying_metadata(bh->b_bdev, bh->b_blocknr);
+ }
+ return 0;
+}
+
+static int ext3_map_write(struct file *file, struct page *page,
+ unsigned from, unsigned to)
+
+ {
+   struct inode *inode = page->mapping->host;
+   int ret, needed_blocks = ext3_writepage_trans_blocks(inode);
@@ -1160,24 +1305,24 @@ retry:
+   handle = ext3_journal_start(inode, needed_blocks);
+   if (IS_ERR(handle)) {
+     ret = PTR_ERR(handle);
- goto out;
- }
- if (test_opt(inode->i_sb, NOBH) && ext3_should_writeback_data(inode))
- ret = nobh_prepare_write(page, from, to, ext3_get_block);

```

```

- else
- ret = block_prepare_write(page, from, to, ext3_get_block);
- if (ret)
- goto prepare_write_failed;
+ goto rollback_out;

- if (ext3_should_journal_data(inode)) {
+ }
+ ret = ext3_do_map_write(inode, page, from, to);
+ if (!ret && ext3_should_journal_data(inode)) {
    ret = walk_page_buffers(handle, page_buffers(page),
        from, to, NULL, do_journal_get_write_access);
    }
-prepare_write_failed:
- if (ret)
- ext3_journal_stop(handle);
+ if (!ret)
+ goto out;
+
+ ext3_journal_stop(handle);
    if (ret == -ENOSPC && ext3_should_retry_alloc(inode->i_sb, &retries))
        goto retry;
+
+rollback_out:
+ /* recovery from block allocation failures */
+ ext3_rollback_write(page, from, to);
out:
    return ret;
}
@@ -1210,10 +1355,15 @@ static int commit_write_fn(handle_t *han
static int ext3_ordered_commit_write(struct file *file, struct page *page,
    unsigned from, unsigned to)
{
- handle_t *handle = ext3_journal_current_handle();
+ handle_t *handle;
    struct inode *inode = page->mapping->host;
    int ret = 0, ret2;

+ ret = ext3_map_write(file, page, from, to);
+ if (ret)
+ return ret;
+ handle = ext3_journal_current_handle();
+
    ret = walk_page_buffers(handle, page_buffers(page),
        from, to, NULL, ext3_journal_dirty_data);

@@ -1239,11 +1389,16 @@ static int ext3_ordered_commit_write(str
static int ext3_writeback_commit_write(struct file *file, struct page *page,

```

```

        unsigned from, unsigned to)
    {
- handle_t *handle = ext3_journal_current_handle();
+ handle_t *handle;
    struct inode *inode = page->mapping->host;
    int ret = 0, ret2;
    loff_t new_i_size;

+ ret = ext3_map_write(file, page, from, to);
+ if (ret)
+ return ret;
+ handle = ext3_journal_current_handle();
+
    new_i_size = ((loff_t)page->index << PAGE_CACHE_SHIFT) + to;
    if (new_i_size > EXT3_I(inode)->i_disksize)
        EXT3_I(inode)->i_disksize = new_i_size;
@@ -1262,12 +1417,17 @@ static int ext3_writeback_commit_write(s
static int ext3_journalled_commit_write(struct file *file,
    struct page *page, unsigned from, unsigned to)
    {
- handle_t *handle = ext3_journal_current_handle();
+ handle_t *handle;
    struct inode *inode = page->mapping->host;
    int ret = 0, ret2;
    int partial = 0;
    loff_t pos;

+ ret = ext3_map_write(file, page, from, to);
+ if (ret)
+ return ret;
+ handle = ext3_journal_current_handle();
+
    /*
     * Here we duplicate the generic_commit_write() functionality
     */
diff --git a/mm/filemap.c b/mm/filemap.c
index 3464b68..f430e28 100644
--- a/mm/filemap.c
+++ b/mm/filemap.c
@@ -1219,21 +2129,25 @@ generic_file_buffered_write(struct kiocb

    status = a_ops->prepare_write(file, page, offset, offset+bytes);
    if (unlikely(status)) {
- loff_t isize = i_size_read(inode);
+ if (status < 0) {
+ loff_t isize = i_size_read(inode);

- if (status != AOP_TRUNCATED_PAGE)

```

```

- unlock_page(page);
- page_cache_release(page);
- if (status == AOP_TRUNCATED_PAGE)
- continue;
- /*
-  * prepare_write() may have instantiated a few blocks
-  * outside i_size. Trim these off again.
-  */
- if (pos + bytes > isize)
- vmtruncate(inode, isize);
- break;
+ if (status != AOP_TRUNCATED_PAGE)
+ unlock_page(page);
+ page_cache_release(page);
+ if (status == AOP_TRUNCATED_PAGE)
+ continue;
+ /*
+  * prepare_write() may have instantiated a few blocks
+  * outside i_size. Trim these off again.
+  */
+ if (pos + bytes > isize)
+ vmtruncate(inode, isize);
+ break;
+ } else if (status < bytes)
+ bytes = status;
+ }
+
+ if (likely(nr_segs == 1))
+ copied = filemap_copy_from_user(page, offset,
+     buf, bytes);
@@ -2183,8 +2187,16 @@ zero_length_segment:
+ unlock_page(page);
+ mark_page_accessed(page);
+ page_cache_release(page);
- if (status < 0)
- break;
+ if (status < 0) {
+ loff_t isize = i_size_read(inode);
+ /*
+  * prepare_write() may have instantiated a few blocks
+  * outside i_size. Trim these off again.
+  */
+ if (pos + bytes > isize)
+ vmtruncate(inode, isize);
+ break;
+ }
+ balance_dirty_pages_ratelimited(mapping);
+ cond_resched();

```

```
} while (count);
```

```
--
```

1.4.2.1
