

$$/*$$

[View Forum Message](#) <> [Reply to Message](#)

```

> index 7b84dc8..bf7cf6c 100644
> --- a/mm/filemap.c
> +++ b/mm/filemap.c
> @@ -2041,6 +2041,14 @@ generic_file_direct_write(struct kiocb *
>     mark_inode_dirty(inode);
> }
> *ppos = end;
> + } else if (written < 0) {
> +     loff_t isize = i_size_read(inode);
> +     /*
> +      * generic_file_direct_IO() may have instantiated a few blocks
> +      * outside i_size. Trim these off again.
> +      */
> +     if (pos + count > isize)
> +         vmtruncate(inode, isize);
> }
>
> /*
>
>
> -----
>

```

Subject: Re: [PATCH] incorrect error handling inside generic_file_direct_write
 Posted by [Andrew Morton](#) on Mon, 11 Dec 2006 20:40:52 GMT
[View Forum Message](#) <> [Reply to Message](#)

On Mon, 11 Dec 2006 16:34:27 +0300
 Dmitry Monakhov <dmonakhov@openvz.org> wrote:

```

> OpenVZ team has discovered error inside generic_file_direct_write()
> If generic_file_direct_IO() has fail (ENOSPC condition) it may have instantiated
> a few blocks outside i_size. And fsck will complain about wrong i_size
> (ext2, ext3 and reiserfs interpret i_size and biggest block difference as error),
> after fsck will fix error i_size will be increased to the biggest block,
> but this blocks contain gurbage from previous write attempt, this is not
> information leak, but its silence file data corruption.
> We need truncate any block beyond i_size after write have failed , do in simular
> generic_file_buffered_write() error path.
>
> Exampe:
> open("mnt2/FILE3", O_WRONLY|O_CREAT|O_DIRECT, 0666) = 3
> write(3, "aaaaaa"..., 4096) = -1 ENOSPC (No space left on device)
>
> stat mnt2/FILE3
> File: `mnt2/FILE3'
> Size: 0           Blocks: 4           IO Block: 4096   regular empty file

```


Page 5 of 20 ---- Generated from [OpenVZ Forum](#)

>
> XFS (at least) can call generic_file_direct_write() with i_mutex not held.
How could it be ?

from mm/filemap.c:2046 generic_file_direct_write() comment right after
place where i want to add vmtruncate()

```
/*  
 * Sync the fs metadata but not the minor inode changes and  
 * of course not the data as we did direct DMA for the IO.  
 * i_mutex is held, which protects generic_osync_inode() from  
 * livelocking.  
 */
```

> And vmtruncate() expects i_mutex to be held.
generic_file_direct_IO must called under i_mutex too
from mm/filemap.c:2388

```
/*  
 * Called under i_mutex for writes to S_ISREG files. Returns -EIO if something  
 * went wrong during pagecache shutdown.  
 */  
static ssize_t  
generic_file_direct_IO(int rw, struct kiocb *iocb, const struct iovec *iov,
```

This means XFS generic_file_direct_write() call generic_file_direct_IO() without
i_mutex held too?

>
> I guess a suitable solution would be to push this problem back up to the
> callers: let them decide whether to run vmtruncate() and if so, to ensure
> that i_mutex is held.
>
> The existence of generic_file_aio_write_nolock() makes that rather messy
> though.

Subject: Re: [PATCH] incorrect error handling inside generic_file_direct_write
Posted by [Andrew Morton](#) on Tue, 12 Dec 2006 06:36:30 GMT
[View Forum Message](#) <> [Reply to Message](#)

On Tue, 12 Dec 2006 12:22:14 +0300
Dmitriy Monakhov <dmonakhov@sw.ru> wrote:

```
> >> @@ -2041,6 +2041,14 @@ generic_file_direct_write(struct kiocb *  
> >>     mark_inode_dirty(inode);  
> >> }  
> >> *ppos = end;  
> >> + } else if (written < 0) {  
> >> + loff_t isize = i_size_read(inode);  
> >> + /*
```

```

>>> + * generic_file_direct_IO() may have instantiated a few blocks
>>> + * outside i_size. Trim these off again.
>>> + */
>>> + if (pos + count > isize)
>>> + vmtruncate(inode, isize);
>>> }
>>>
>>
>> XFS (at least) can call generic_file_direct_write() with i_mutex not held.
> How could it be ?
>
> from mm/filemap.c:2046 generic_file_direct_write() comment right after
> place where i want to add vmtruncate()
> /*
> * Sync the fs metadata but not the minor inode changes and
> * of course not the data as we did direct DMA for the IO.
> * i_mutex is held, which protects generic_osync_inode() from
> * livelocking.
> */
>
>> And vmtruncate() expects i_mutex to be held.
> generic_file_direct_IO must called under i_mutex too
> from mm/filemap.c:2388
> /*
> * Called under i_mutex for writes to S_ISREG files. Returns -EIO if something
> * went wrong during pagecache shutdown.
> */
> static ssize_t
> generic_file_direct_IO(int rw, struct kiocb *iocb, const struct iovec *iov,

```

yup, the comments are wrong.

```

> This means XFS generic_file_direct_write() call generic_file_direct_IO() without
> i_mutex held too?

```

Think so. XFS uses `blockdev_direct_IO_own_locking()`. We'd need to check with the XFS guys regarding its precise operation and what needs to be done here.

```

>>
>> I guess a suitable solution would be to push this problem back up to the
>> callers: let them decide whether to run vmtruncate() and if so, to ensure
>> that i_mutex is held.
>>
>> The existence of generic_file_aio_write_nolock() makes that rather messy
>> though.

```

Subject: Re: [PATCH] incorrect error handling inside generic_file_direct_write
Posted by [Dmitriy Monakhov](#) on Tue, 12 Dec 2006 09:20:59 GMT
[View Forum Message](#) <> [Reply to Message](#)

Andrew Morton <akpm@osdl.org> writes:

[illegible]

```

>> + * outside i_size. Trim these off again.
>> + */
>> + if (pos + count > isize)
>> + vmtruncate(inode, isize);
>> }
>>
>
> XFS (at least) can call generic_file_direct_write() with i_mutex not held.
> And vmtruncate() expects i_mutex to be held.
>
> I guess a suitable solution would be to push this problem back up to the
> callers: let them decide whether to run vmtruncate() and if so, to ensure
> that i_mutex is held.
>
> The existence of generic_file_aio_write_nolock() makes that rather messy
> though.
This means we may call generic_file_aio_write_nolock() without i_mutex, right?
but call trace is :
    generic_file_aio_write_nolock()
->generic_file_buffered_write() /* i_mutex not held here */
but according to filemaps locking rules: mm/filemap.c:77
..
* ->i_mutex (generic_file_buffered_write)
* ->mmap_sem (fault_in_pages_readable->do_page_fault)
..
I'm confused a little bit, where is the truth?

```

Subject: Re: [PATCH] incorrect error handling inside generic_file_direct_write
 Posted by [Andrew Morton](#) on Tue, 12 Dec 2006 09:52:32 GMT
[View Forum Message](#) <> [Reply to Message](#)

On Tue, 12 Dec 2006 15:20:52 +0300
 Dmitry Monakhov <dmonakhov@sw.ru> wrote:

```

> > XFS (at least) can call generic_file_direct_write() with i_mutex not held.
> > And vmtruncate() expects i_mutex to be held.
> >
> > I guess a suitable solution would be to push this problem back up to the
> > callers: let them decide whether to run vmtruncate() and if so, to ensure
> > that i_mutex is held.
> >
> > The existence of generic_file_aio_write_nolock() makes that rather messy
> > though.
> This means we may call generic_file_aio_write_nolock() without i_mutex, right?
> but call trace is :
> generic_file_aio_write_nolock()
> ->generic_file_buffered_write() /* i_mutex not held here */

```

> but according to filemaps locking rules: mm/filemap.c:77
> ..
> * ->i_mutex (generic_file_buffered_write)
> * ->mmap_sem (fault_in_pages_readable->do_page_fault)
> ..
> I'm confused a little bit, where is the truth?

xfs_write() calls generic_file_direct_write() without taking i_mutex for
O_DIRECT writes.

Subject: Re: [PATCH] incorrect error handling inside generic_file_direct_write
Posted by [Dmitriy Monakhov](#) on Tue, 12 Dec 2006 10:18:38 GMT
[View Forum Message](#) <> [Reply to Message](#)

Andrew Morton <akpm@osdl.org> writes:

> On Tue, 12 Dec 2006 15:20:52 +0300
> Dmitriy Monakhov <dmonakhov@sw.ru> wrote:
>
>> > XFS (at least) can call generic_file_direct_write() with i_mutex not held.
>> > And vmtruncate() expects i_mutex to be held.
>> >
>> > I guess a suitable solution would be to push this problem back up to the
>> > callers: let them decide whether to run vmtruncate() and if so, to ensure
>> > that i_mutex is held.
>> >
>> > The existence of generic_file_aio_write_nolock() makes that rather messy
>> > though.
>> This means we may call generic_file_aio_write_nolock() without i_mutex, right?
>> but call trace is :
>> generic_file_aio_write_nolock()
>> ->generic_file_buffered_write() /* i_mutex not held here */
>> but according to filemaps locking rules: mm/filemap.c:77
>> ..
>> * ->i_mutex (generic_file_buffered_write)
>> * ->mmap_sem (fault_in_pages_readable->do_page_fault)
>> ..
>> I'm confused a little bit, where is the truth?
>
> xfs_write() calls generic_file_direct_write() without taking i_mutex for
> O_DIRECT writes.
Yes, but my question is about __generic_file_aio_write_nolock().
As I understand _nolock suffix means that i_mutex was already locked
by caller, am I right ?
If yes, then __generic_file_aio_write_nolock() is a better place for vmtruncate()
activity after generic_file_direct_write() has failed.
Signed-off-by: Dmitriy Monakhov <dmonakhov@openvz.org>

```
diff --git a/mm/filemap.c b/mm/filemap.c
index 7b84dc8..723d2ca 100644
--- a/mm/filemap.c
+++ b/mm/filemap.c
@@ -2282,6 +2282,15 @@ __generic_file_aio_write_nolock(struct k
    written = generic_file_direct_write(iocb, iov, &nr_segs, pos,
        ppos, count, ocount);
+   if (written < 0) {
+       loff_t isize = i_size_read(inode);
+       /*
+        * generic_file_direct_write() may have instantiated
+        * a few blocks outside i_size. Trim these off again.
+        */
+       if (pos + count > isize)
+           vmtruncate(inode, isize);
+   }
    if (written < 0 || written == count)
        goto out;
    /*
```

Subject: Re: [PATCH] incorrect error handling inside generic_file_direct_write
Posted by [Andrew Morton](#) on Tue, 12 Dec 2006 10:40:27 GMT
[View Forum Message](#) <> [Reply to Message](#)

On Tue, 12 Dec 2006 16:18:32 +0300
Dmitriy Monakhov <dmonakhov@sw.ru> wrote:

```
> >> but according to filemaps locking rules: mm/filemap.c:77
> >> ..
> >> * ->i_mutex (generic_file_buffered_write)
> >> * ->mmap_sem (fault_in_pages_readable->do_page_fault)
> >> ..
> >> I'm confused a little bit, where is the truth?
> >
> > xfs_write() calls generic_file_direct_write() without taking i_mutex for
> > O_DIRECT writes.
> Yes, but my question is about __generic_file_aio_write_nolock().
> As I understand _nolock suffix means that i_mutex was already locked
> by caller, am I right ?
```

Nope. It just means that __generic_file_aio_write_nolock() doesn't take the lock. We don't assume or require that the caller took it. For example the raw driver calls generic_file_aio_write_nolock() without taking i_mutex. Raw isn't relevant to the problem (although ocfs2 might be). But

we cannot assume that all callers have taken `i_mutex`, I think.

I guess we can make that a rule (document it, add `BUG_ON(!mutex_is_locked(..))` if it isn't a blockdev) if needs be. After really checking that this matches reality for all callers.

It's important, too - if we have an unprotected `i_size_write()` then the seqlock can get out of sync due to a race and then `i_size_read()` locks up the kernel.

Subject: Re: [PATCH] incorrect error handling inside `generic_file_direct_write`
Posted by [Dmitriy Monakhov](#) on Tue, 12 Dec 2006 20:14:31 GMT
[View Forum Message](#) <> [Reply to Message](#)

Andrew Morton <akpm@osdl.org> writes:

```
> On Tue, 12 Dec 2006 16:18:32 +0300
> Dmitriy Monakhov <dmonakhov@sw.ru> wrote:
>
>> >> but according to filemaps locking rules: mm/filemap.c:77
>> >> ..
>> >> * ->i_mutex  (generic_file_buffered_write)
>> >> * ->mmap_sem (fault_in_pages_readable->do_page_fault)
>> >> ..
>> >> I'm confused a little bit, where is the truth?
>> >
>> > xfs_write() calls generic_file_direct_write() without taking i_mutex for
>> > O_DIRECT writes.
>> Yes, but my question is about __generic_file_aio_write_nolock().
>> As I understand _nolock suffix means that i_mutex was already locked
>> by caller, am I right ?
>
> Nope. It just means that __generic_file_aio_write_nolock() doesn't take
> the lock. We don't assume or require that the caller took it. For example
> the raw driver calls generic_file_aio_write_nolock() without taking
> i_mutex. Raw isn't relevant to the problem (although ocfs2 might be). But
> we cannot assume that all callers have taken i_mutex, I think.
>
> I guess we can make that a rule (document it, add
> BUG_ON(!mutex_is_locked(..)) if it isn't a blockdev) if needs be. After
> really checking that this matches reality for all callers.
I've checked generic_file_aio_write_nolock() callers for non blockdev.
Only ocfs2 call it explicitly, and do it under i_mutex.
So we need to do:
1) Change wrong comments.
2) Add BUG_ON(!mutex_is_locked(..)) for non blkdev.
3) Invoke vmtruncate only for non blkdev.
```

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

```
diff --git a/mm/filemap.c b/mm/filemap.c
index 7b84dc8..540ef5e 100644
--- a/mm/filemap.c
+++ b/mm/filemap.c
@@ -2046,8 +2046,8 @@ generic_file_direct_write(struct kiocb *
/*
 * Sync the fs metadata but not the minor inode changes and
 * of course not the data as we did direct DMA for the IO.
- * i_mutex is held, which protects generic_osync_inode() from
- * livelocking.
+ * i_mutex may not being held, if so some specific locking
+ * ordering must protect generic_osync_inode() from livelocking.
 */
if (written >= 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode))) {
    int err = generic_osync_inode(inode, mapping, O_SYNC_METADATA);
@@ -2282,6 +2282,17 @@ __generic_file_aio_write_nolock(struct k

    written = generic_file_direct_write(iocb, iov, &nr_segs, pos,
        ppos, count, ocount);
+ /*
+ * If host is not S_ISBLK generic_file_direct_write() may
+ * have instantiated a few blocks outside i_size files
+ * Trim these off again.
+ */
+ if (unlikely(written < 0) && !S_ISBLK(inode->i_mode)) {
+     loff_t isize = i_size_read(inode);
+     if (pos + count > isize)
+         vmtruncate(inode, isize);
+ }
+
    if (written < 0 || written == count)
        goto out;
/*
@@ -2344,6 +2355,13 @@ ssize_t generic_file_aio_write_nolock(st
    ssize_t ret;

    BUG_ON(iocb->ki_pos != pos);
+ /*
+ * generic_file_buffered_write() may be called inside
+ * __generic_file_aio_write_nolock() even in case of
+ * O_DIRECT for non S_ISBLK files. So i_mutex must be held.
+ */
+ if (!S_ISBLK(inode->i_mode))
+     BUG_ON(!mutex_is_locked(&inode->i_mutex));
```

```

ret = __generic_file_aio_write_nolock(iocb, iov, nr_segs,
    &iocb->ki_pos);
@@ -2386,8 +2404,8 @@ ssize_t generic_file_aio_write(struct ki
EXPORT_SYMBOL(generic_file_aio_write);

/*
- * Called under i_mutex for writes to S_ISREG files. Returns -EIO if something
- * went wrong during pagecache shutdown.
+ * May be called without i_mutex for writes to S_ISREG files.
+ * Returns -EIO if something went wrong during pagecache shutdown.
 */
static ssize_t
generic_file_direct_IO(int rw, struct kiocb *iocb, const struct iovec *iov,

```

Subject: RE: [PATCH] incorrect error handling inside generic_file_direct_write
 Posted by [kenneth.w.chen](#) on Wed, 13 Dec 2006 02:43:37 GMT
[View Forum Message](#) <> [Reply to Message](#)

Andrew Morton wrote on Tuesday, December 12, 2006 2:40 AM
 > On Tue, 12 Dec 2006 16:18:32 +0300
 > Dmitriy Monakhov <dmonakhov@sw.ru> wrote:
 >
 > > > but according to filemaps locking rules: mm/filemap.c:77
 > > > ..
 > > > * ->i_mutex (generic_file_buffered_write)
 > > > * ->mmap_sem (fault_in_pages_readable->do_page_fault)
 > > > ..
 > > > I'm confused a little bit, where is the truth?
 > > >
 > > > xfs_write() calls generic_file_direct_write() without taking i_mutex for
 > > > O_DIRECT writes.
 > > Yes, but my question is about __generic_file_aio_write_nolock().
 > > As I understand _nolock suffix means that i_mutex was already locked
 > > by caller, am I right ?
 >
 > Nope. It just means that __generic_file_aio_write_nolock() doesn't take
 > the lock. We don't assume or require that the caller took it. For example
 > the raw driver calls generic_file_aio_write_nolock() without taking
 > i_mutex. Raw isn't relevant to the problem (although ocfs2 might be). But
 > we cannot assume that all callers have taken i_mutex, I think.

I think we should also clean up generic_file_aio_write_nolock. This was brought up a couple of weeks ago and I gave up too early. Here is my second attempt.

How about the following patch, I think we can kill generic_file_aio_write_nolock and merge both *file_aio_write_nolock into one function, then

```
generic_file_aio_write
ocfs2_file_aio_write
blk_dev->aio_write
```

all points to a non-lock version of __generic_file_aio_write(). First two already hold i_mutex, while the block device's aio_write method doesn't require i_mutex to be held.

Signed-off-by: Ken Chen <kenneth.w.chen@intel.com>

```
diff -Nurp linux-2.6.19/drivers/char/raw.c linux-2.6.19.ken/drivers/char/raw.c
--- linux-2.6.19/drivers/char/raw.c 2006-11-29 13:57:37.000000000 -0800
+++ linux-2.6.19.ken/drivers/char/raw.c 2006-12-12 16:41:39.000000000 -0800
@@ -242,7 +242,7 @@ static const struct file_operations raw_
 .read = do_sync_read,
 .aio_read = generic_file_aio_read,
 .write = do_sync_write,
- .aio_write = generic_file_aio_write_nolock,
+ .aio_write = __generic_file_aio_write,
 .open = raw_open,
 .release = raw_release,
 .ioctl = raw_ioctl,
diff -Nurp linux-2.6.19/fs/block_dev.c linux-2.6.19.ken/fs/block_dev.c
--- linux-2.6.19/fs/block_dev.c 2006-11-29 13:57:37.000000000 -0800
+++ linux-2.6.19.ken/fs/block_dev.c 2006-12-12 16:47:58.000000000 -0800
@@ -1198,7 +1198,7 @@ const struct file_operations def_blk_fop
 .read = do_sync_read,
 .write = do_sync_write,
 .aio_read = generic_file_aio_read,
- .aio_write = generic_file_aio_write_nolock,
+ .aio_write = __generic_file_aio_write,
 .mmap = generic_file_mmap,
 .fsync = block_fsync,
 .unlocked_ioctl = block_ioctl,
diff -Nurp linux-2.6.19/fs/ocfs2/file.c linux-2.6.19.ken/fs/ocfs2/file.c
--- linux-2.6.19/fs/ocfs2/file.c 2006-11-29 13:57:37.000000000 -0800
+++ linux-2.6.19.ken/fs/ocfs2/file.c 2006-12-12 16:42:09.000000000 -0800
@@ -1107,7 +1107,7 @@ static ssize_t ocfs2_file_aio_write(stru
 /* communicate with ocfs2_dio_end_io */
 ocfs2_iocb_set_rw_locked(iocb);

- ret = generic_file_aio_write_nolock(iocb, iov, nr_segs, iocb->ki_pos);
+ ret = __generic_file_aio_write(iocb, iov, nr_segs, iocb->ki_pos);
```

```

/* buffered aio wouldn't have proper lock coverage today */
BUG_ON(ret == -EIOCBQUEUED && !(filp->f_flags & O_DIRECT));
diff -Nurp linux-2.6.19/include/linux/fs.h linux-2.6.19.ken/include/linux/fs.h
--- linux-2.6.19/include/linux/fs.h 2006-11-29 13:57:37.000000000 -0800
+++ linux-2.6.19.ken/include/linux/fs.h 2006-12-12 16:41:58.000000000 -0800
@@ -1742,7 +1742,7 @@ extern int file_send_actor(read_descript
int generic_write_checks(struct file *file, loff_t *pos, size_t *count, int isblk);
extern ssize_t generic_file_aio_read(struct kiocb *, const struct iovec *, unsigned long, loff_t);
extern ssize_t generic_file_aio_write(struct kiocb *, const struct iovec *, unsigned long, loff_t);
-extern ssize_t generic_file_aio_write_nolock(struct kiocb *, const struct iovec *,
+extern ssize_t __generic_file_aio_write(struct kiocb *, const struct iovec *,
    unsigned long, loff_t);
extern ssize_t generic_file_direct_write(struct kiocb *, const struct iovec *,
    unsigned long *, loff_t, loff_t *, size_t, size_t);
diff -Nurp linux-2.6.19/mm/filemap.c linux-2.6.19.ken/mm/filemap.c
--- linux-2.6.19/mm/filemap.c 2006-11-29 13:57:37.000000000 -0800
+++ linux-2.6.19.ken/mm/filemap.c 2006-12-12 16:47:58.000000000 -0800
@@ -2219,9 +2219,9 @@ zero_length_segment:
}
EXPORT_SYMBOL(generic_file_buffered_write);

-static ssize_t
-__generic_file_aio_write_nolock(struct kiocb *iocb, const struct iovec *iov,
-    unsigned long nr_segs, loff_t *ppos)
+ssize_t
+__generic_file_aio_write(struct kiocb *iocb, const struct iovec *iov,
+    unsigned long nr_segs, loff_t pos)
{
    struct file *file = iocb->ki_filp;
    struct address_space * mapping = file->f_mapping;
@@ -2229,9 +2229,10 @@ __generic_file_aio_write_nolock(struct k
    size_t count; /* after file limit checks */
    struct inode *inode = mapping->host;
    unsigned long seg;
-    loff_t pos;
+    loff_t *ppos = &iocb->ki_pos;
    ssize_t written;
    ssize_t err;
+    ssize_t ret;

    ocount = 0;
    for (seg = 0; seg < nr_segs; seg++) {
@@ -2254,7 +2255,6 @@ __generic_file_aio_write_nolock(struct k
    }

    count = ocount;
-    pos = *ppos;

```

```

    vfs_check_frozen(inode->i_sb, SB_FREEZE_WRITE);

@@ -2332,32 +2332,16 @@ __generic_file_aio_write_nolock(struct k
    }
    out:
    current->backing_dev_info = NULL;
- return written ? written : err;
-}
-
-ssize_t generic_file_aio_write_nolock(struct kiocb *iocb,
- const struct iovec *iov, unsigned long nr_segs, loff_t pos)
-{
- struct file *file = iocb->ki_filp;
- struct address_space *mapping = file->f_mapping;
- struct inode *inode = mapping->host;
- ssize_t ret;
-
- BUG_ON(iocb->ki_pos != pos);
-
- ret = __generic_file_aio_write_nolock(iocb, iov, nr_segs,
- &iocb->ki_pos);
+ ret = written ? written : err;

    if (ret > 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode))) {
- ssize_t err;
-
- err = sync_page_range_nolock(inode, mapping, pos, ret);
- if (err < 0)
-     ret = err;
- }
- return ret;
- }
-EXPORT_SYMBOL(generic_file_aio_write_nolock);
+EXPORT_SYMBOL(__generic_file_aio_write);

    ssize_t generic_file_aio_write(struct kiocb *iocb, const struct iovec *iov,
        unsigned long nr_segs, loff_t pos)
@@ -2370,8 +2354,7 @@ ssize_t generic_file_aio_write(struct ki
    BUG_ON(iocb->ki_pos != pos);

    mutex_lock(&inode->i_mutex);
- ret = __generic_file_aio_write_nolock(iocb, iov, nr_segs,
- &iocb->ki_pos);
+ ret = __generic_file_aio_write(iocb, iov, nr_segs, pos);
    mutex_unlock(&inode->i_mutex);

    if (ret > 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode))) {

```

Subject: Re: [PATCH] incorrect error handling inside generic_file_direct_write
Posted by [Christoph Hellwig](#) on Fri, 15 Dec 2006 10:43:41 GMT
[View Forum Message](#) <> [Reply to Message](#)

```
> +ssize_t  
> +__generic_file_aio_write(struct kiocb *iocb, const struct iovec *iov,  
> +    unsigned long nr_segs, loff_t pos)
```

I'd still call this generic_file_aio_write_nolock.

```
> + loff_t *ppos = &iocb->ki_pos;
```

I'd rather use iocb->ki_pos directly in the few places ppos is referenced currently.

```
> if (ret > 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode))) {  
> - ssize_t err;  
> -  
>   err = sync_page_range_nolock(inode, mapping, pos, ret);  
>   if (err < 0)  
>     ret = err;  
> }
```

So we're doing the sync_page_range once in __generic_file_aio_write with i_mutex held.

```
> mutex_lock(&inode->i_mutex);  
> - ret = __generic_file_aio_write_nolock(iocb, iov, nr_segs,  
> -    &iocb->ki_pos);  
> + ret = __generic_file_aio_write(iocb, iov, nr_segs, pos);  
> mutex_unlock(&inode->i_mutex);  
>  
> if (ret > 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode))) {
```

And then another time after it's unlocked, this seems wrong.

Subject: RE: [PATCH] incorrect error handling inside generic_file_direct_write
Posted by [kenneth.w.chen](#) on Fri, 15 Dec 2006 18:53:18 GMT
[View Forum Message](#) <> [Reply to Message](#)

Christoph Hellwig wrote on Friday, December 15, 2006 2:44 AM

```
> So we're doing the sync_page_range once in __generic_file_aio_write  
> with i_mutex held.  
>  
>  
> > mutex_lock(&inode->i_mutex);
```

```
> > - ret = __generic_file_aio_write_nolock(iocb, iov, nr_segs,
> > - &iocb->ki_pos);
> > + ret = __generic_file_aio_write(iocb, iov, nr_segs, pos);
> > mutex_unlock(&inode->i_mutex);
> >
> > if (ret > 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode))) {
>
> And then another time after it's unlocked, this seems wrong.
```

I didn't invent that mess though.

I should've ask the question first: in 2.6.20-rc1, generic_file_aio_write will call sync_page_range twice, once from __generic_file_aio_write_nolock and once within the function itself. Is it redundant? Can we delete the one in the top level function? Like the following?

```
--- ./mm/filemap.c.orig 2006-12-15 09:02:58.000000000 -0800
+++ ./mm/filemap.c 2006-12-15 09:03:19.000000000 -0800
@@ -2370,14 +2370,6 @@ ssize_t generic_file_aio_write(struct ki
     ret = __generic_file_aio_write_nolock(iocb, iov, nr_segs,
        &iocb->ki_pos);
     mutex_unlock(&inode->i_mutex);
-
- if (ret > 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode))) {
-     ssize_t err;
-
-     err = sync_page_range(inode, mapping, pos, ret);
-     if (err < 0)
-         ret = err;
- }
    return ret;
}
EXPORT_SYMBOL(generic_file_aio_write);
```

Subject: Re: [PATCH] incorrect error handling inside generic_file_direct_write
 Posted by [Christoph Hellwig](#) on Tue, 02 Jan 2007 11:17:46 GMT
[View Forum Message](#) <> [Reply to Message](#)

On Fri, Dec 15, 2006 at 10:53:18AM -0800, Chen, Kenneth W wrote:
 > Christoph Hellwig wrote on Friday, December 15, 2006 2:44 AM
 > > So we're doing the sync_page_range once in __generic_file_aio_write
 > > with i_mutex held.
 > >
 > >
 > > > mutex_lock(&inode->i_mutex);

```

>>> - ret = __generic_file_aio_write_nolock(iocb, iov, nr_segs,
>>> - &iocb->ki_pos);
>>> + ret = __generic_file_aio_write(iocb, iov, nr_segs, pos);
>>> mutex_unlock(&inode->i_mutex);
>>>
>>> if (ret > 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode))) {
>>
>> And then another time after it's unlocked, this seems wrong.
>
>
> I didn't invent that mess though.
>
> I should've ask the question first: in 2.6.20-rc1, generic_file_aio_write
> will call sync_page_range twice, once from __generic_file_aio_write_nolock
> and once within the function itself. Is it redundant? Can we delete the
> one in the top level function? Like the following?

```

Really? I'm looking at -rc3 now as -rc1 is rather old and it's definitely not the case there. I also can't remember ever doing this - when I started the generic read/write path untangling I had exactly the same situation that's now in -rc3:

- generic_file_aio_write_nolock calls sync_page_range_nolock
 - generic_file_aio_write calls sync_page_range
 - __generic_file_aio_write_nolock doesn't call any sync_page_range variant
-