Subject: [PATCH 0/1][RFC] prepare_write positive return value V(2)
Posted by Dmitriy Monakhov on Mon, 12 Feb 2007 09:45:45 GMT
View Forum Message <> Reply to Message

Changes from ver(1)
 - __page_symlink(): In order to be on a safe side add explicit zeroing content
                before fail (just in case).
 -do_lo_send_aops(): If prepare_write can't handle total size of bytes requested
                from loop_dev it is safer to fail.
 - pipe_to_file(): Limit the size of the copy to size wich prepare_write ready
                to handle. instead of failing.
 - ecryptfs: was't changed in this version, because where is no even
                AOP_TRUNCATED_PAGE handling code where.

Almost all read/write operation handles data with chunks(segments or pages)
and result has integral behaviour for folowing scenario:
for_each_chunk() {
    res = op(....);
    if(IS_ERROR(res))
        return progress ? progress : res;
    progress += res;
}
prepare_write may has integral behaviour in case of blksize < pgsize,
for example we successfully allocated/read some blocks, but not all of them,
and than some error happend. Currently we eliminate this progress by doing
vmtrunate() after prepare_has failed.
It is good to have ability to signal about this progress. Interprete positive
prepare_write() ret code as bytes num that fs ready to handle at this moment.

BTH: This approach was used in OpenVZ 2.6.9 kernel in order to make FS with
delayed allocation more correct, and its works well.
I think not everybody will happy about this, but let's discuss all advantages
and disadvantages of this change.

fixes not dirrectly connected with proposed prepare_write semantic changes:
 __page_symlink : If find_or_create_page has failed on second retry attempt
                function will exit with wrong error code.
 __generic_cont_expand: Add correct AOP_TRUNCATED_PAGE handling.

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

diff --git a/drivers/block/loop.c b/drivers/block/loop.c
index 6b5b642..dc332dc 100644
--- a/drivers/block/loop.c
+++ b/drivers/block/loop.c
@@ -224,6 +224,7 @@ static int do_lo_send_aops(struct loop_device *lo, struct bio_vec *bvec,
   sector_t IV;

```
   unsigned size;
   int transfer_result;
+  int partial = 0;

   IV = ((sector_t)index << (PAGE_CACHE_SHIFT - 9))+(offset >> 9);
   size = PAGE_CACHE_SIZE - offset;
@@ -239,11 +240,20 @@ static int do_lo_send_aops(struct loop_device *lo, struct bio_vec
*bvec,
     page_cache_release(page);
     continue;
    }
-   goto unlock;
+   if (ret > 0 && ret < PAGE_CACHE_SIZE) {
+   /*
+      * ->prepare_write() can't handle total size of bytes
+    * requested. In fact this is not likely to happen,
+    * because we request one block only.
+      */
+    partial = 1;
+    size = ret;
+   } else
+    goto unlock;
   }
   transfer_result = lo_do_transfer(lo, WRITE, page, offset,
     bvec->bv_page, bv_offs, size, IV);
-  if (unlikely(transfer_result)) {
+  if (unlikely(transfer_result || partial)) {
    char *kaddr;

    /*
@@ -266,7 +276,7 @@ static int do_lo_send_aops(struct loop_device *lo, struct bio_vec *bvec,
    }
    goto unlock;
   }
-  if (unlikely(transfer_result))
+  if (unlikely(transfer_result || partial))
    goto unlock;
   bv_offs += size;
   len -= size;
diff --git a/fs/buffer.c b/fs/buffer.c
index 1ad674f..ed06560 100644
--- a/fs/buffer.c
+++ b/fs/buffer.c
@@ -2027,13 +2027,20 @@ static int __generic_cont_expand(struct inode *inode, loff_t size,
  if (size > inode->i_sb->s_maxbytes)
   goto out;

+retry:
```

```
  err = -ENOMEM;
  page = grab_cache_page(mapping, index);
  if (!page)
   goto out;
  err = mapping->a_ops->prepare_write(NULL, page, offset, offset);
+ if (err == AOP_TRUNCATED_PAGE) {
+  page_cache_release(page);
+  goto retry;
+ }
  if (err) {
   /*
+   * ->prepare_write() called with from==to and must not
+   * return positive size of bytes in this case.
    * ->prepare_write() may have instantiated a few blocks
    * outside i_size.  Trim these off again.
    */
@@ -2044,6 +2051,10 @@ static int __generic_cont_expand(struct inode *inode, loff_t size,
  }

  err = mapping->a_ops->commit_write(NULL, page, offset, offset);
+ if (err == AOP_TRUNCATED_PAGE) {
+  page_cache_release(page);
+  goto retry;
+ }

  unlock_page(page);
  page_cache_release(page);
diff --git a/fs/namei.c b/fs/namei.c
index e4f108f..6626277 100644
--- a/fs/namei.c
+++ b/fs/namei.c
@@ -2688,20 +2688,36 @@ int __page_symlink(struct inode *inode, const char *symname, int
len,
 {
  struct address_space *mapping = inode->i_mapping;
  struct page *page;
- int err = -ENOMEM;
+ int err;
  char *kaddr;

 retry:
+ err = -ENOMEM;
  page = find_or_create_page(mapping, 0, gfp_mask);
  if (!page)
   goto fail;
  err = mapping->a_ops->prepare_write(NULL, page, 0, len-1);
- if (err == AOP_TRUNCATED_PAGE) {
-  page_cache_release(page);
```

```
-  goto retry;
- }
- if (err)
+ if(unlikely(err)){
+  if (err == AOP_TRUNCATED_PAGE) {
+   page_cache_release(page);
+   goto retry;
+  }
+  if (err > 0 && err < PAGE_CACHE_SIZE) {
+  /*
+    * ->prepare_write() can't handle total size of bytes requested.
+    * Really this is not likely to happen, because usualy we need
+   * one block only. In order to preserve prepare/commit balanced
+   * invoke commit and fail with ENOSPC.
+    */
+   int ret;
+   kaddr = kmap_atomic(page, KM_USER0);
+   memset(kaddr, 0, err);
+   kunmap_atomic(kaddr, KM_USER0);
+   ret = mapping->a_ops->commit_write(NULL, page, 0, err);
+   err = (ret < 0) ? ret : -ENOSPC;
+  }
   goto fail_map;
+ }
  kaddr = kmap_atomic(page, KM_USER0);
  memcpy(kaddr, symname, len-1);
  kunmap_atomic(kaddr, KM_USER0);
diff --git a/fs/splice.c b/fs/splice.c
index 2fca6eb..78c3e88 100644
--- a/fs/splice.c
+++ b/fs/splice.c
@@ -649,6 +649,14 @@ find_page:
  }

  ret = mapping->a_ops->prepare_write(file, page, offset, offset+this_len);
+ if (unlikely(ret > 0 && ret < PAGE_CACHE_SIZE)) {
+ /*
+  * Limit the size of the copy to size wich prepare_write
+  * ready to handle.
+  */
+  this_len = ret;
+  ret = 0;
+ }
  if (unlikely(ret)) {
   loff_t isize = i_size_read(mapping->host);

diff --git a/mm/filemap.c b/mm/filemap.c
index 8332c77..8b4789d 100644
```

```
--- a/mm/filemap.c
+++ b/mm/filemap.c
@@ -2127,6 +2127,14 @@ generic_file_buffered_write(struct kiocb *iocb, const struct iovec *iov,
   }

   status = a_ops->prepare_write(file, page, offset, offset+bytes);
+  if (unlikely(status > 0 && status < PAGE_CACHE_SIZE)) {
+  /*
+   * Limit the size of the copy to size wich prepare_write
+   * ready to handle.
+   */
+   bytes = status;
+   status = 0;
+  }
   if (unlikely(status)) {
    loff_t isize = i_size_read(inode);
```