Subject: [PATCH] Consolidate sleeping routines in file locking code
Posted by Pavel Emelianov on Tue, 18 Sep 2007 13:41:08 GMT
View Forum Message <> Reply to Message

This is the next step in fs/locks.c cleanup before turning
it into using the struct pid *.

This time I found, that there are some places that do a
similar thing - they try to apply a lock on a file and go
to sleep on error till the blocker exits.

All these places can be easily consolidated, saving 28
lines of code and more than 600 bytes from the .text,
but there is one minor note.

The locks_mandatory_area() code becomes a bit different
after this patch - it no longer checks for the inode's
permissions change. Nevertheless, this check is useless
without my another patch that wakes the waiter up in the
notify_change(), which is not considered to be useful for
now.

Later, if we do need the fix with the wakeup this can be
easily merged with this patch.

Signed-off-by: Pavel Emelyanov <xemul@openvz.org>

---

 fs/locks.c | 122 ++++++++++++++++++++++++++++++++----------------------------
 1 files changed, 47 insertions(+), 75 deletions(-)

diff --git a/fs/locks.c b/fs/locks.c
index cb1c977..8e849ed 100644
--- a/fs/locks.c
+++ b/fs/locks.c
@@ -659,6 +659,29 @@ static int locks_block_on_timeout(struct
  return result;
 }

+typedef int (*lock_wait_fn)(struct file *, struct file_lock *, void *);
+
+static int do_lock_file_wait(struct file *filp, struct file_lock *fl,
+ lock_wait_fn lockfn, void *arg)
+{
+ int error;
+
+ might_sleep();

```
+ while (1) {
+  error = lockfn(filp, fl, arg);
+  if ((error != -EAGAIN) || !(fl->fl_flags & FL_SLEEP))
+   break;
+
+  error = wait_event_interruptible(fl->fl_wait, !fl->fl_next);
+  if (error) {
+   locks_delete_block(fl);
+   break;
+  }
+ }
+
+ return error;
+}
+
 void
 posix_test_lock(struct file *filp, struct file_lock *fl)
 {
@@ -720,7 +743,8 @@ next_task:
  * whether or not a lock was successfully freed by testing the return
  * value for -ENOENT.
  */
-static int flock_lock_file(struct file *filp, struct file_lock *request)
+static
+int flock_lock_file(struct file *filp, struct file_lock *request, void *x)
 {
  struct file_lock *new_fl = NULL;
  struct file_lock **before;
@@ -1029,20 +1053,7 @@ EXPORT_SYMBOL(posix_lock_file);
  */
 int posix_lock_file_wait(struct file *filp, struct file_lock *fl)
 {
- int error;
- might_sleep ();
- for (;;) {
-  error = posix_lock_file(filp, fl, NULL);
-  if ((error != -EAGAIN) || !(fl->fl_flags & FL_SLEEP))
-   break;
-  error = wait_event_interruptible(fl->fl_wait, !fl->fl_next);
-  if (!error)
-   continue;
-
-  locks_delete_block(fl);
-  break;
- }
- return error;
+ return do_lock_file_wait(filp, fl, (lock_wait_fn)posix_lock_file, NULL);
 }
```

```
 EXPORT_SYMBOL(posix_lock_file_wait);

@@ -1085,12 +1096,17 @@ int locks_mandatory_locked(struct inode
  * This function is called from rw_verify_area() and
  * locks_verify_truncate().
  */
+
+static int lock_mandatory_fn(struct file *filp, struct file_lock *fl, void *arg)
+{
+ return __posix_lock_file((struct inode *)arg, fl, NULL);
+}
+
 int locks_mandatory_area(int read_write, struct inode *inode,
     struct file *filp, loff_t offset,
     size_t count)
 {
  struct file_lock fl;
- int error;

  locks_init_lock(&fl);
  fl.fl_owner = current->files;
@@ -1103,27 +1119,12 @@ int locks_mandatory_area(int read_write,
  fl.fl_start = offset;
  fl.fl_end = offset + count - 1;

- for (;;) {
-  error = __posix_lock_file(inode, &fl, NULL);
-  if (error != -EAGAIN)
-   break;
-  if (!(fl.fl_flags & FL_SLEEP))
-   break;
-  error = wait_event_interruptible(fl.fl_wait, !fl.fl_next);
-  if (!error) {
-   /*
-    * If we've been sleeping someone might have
-    * changed the permissions behind our back.
-    */
-   if (__mandatory_lock(inode))
-    continue;
-  }
-
-  locks_delete_block(&fl);
-  break;
- }
-
- return error;
+ /*
+  * If we've been sleeping someone might have changed the permissions
```

```
+  * behind our back. However, nobody wakes us up, so go on spinning
+  * here till the blocker dies.
+  */
+ return do_lock_file_wait(filp, &fl, lock_mandatory_fn, inode);
 }

 EXPORT_SYMBOL(locks_mandatory_area);
@@ -1517,20 +1518,7 @@ out_unlock:
  */
 int flock_lock_file_wait(struct file *filp, struct file_lock *fl)
 {
- int error;
- might_sleep();
- for (;;) {
-  error = flock_lock_file(filp, fl);
-  if ((error != -EAGAIN) || !(fl->fl_flags & FL_SLEEP))
-   break;
-  error = wait_event_interruptible(fl->fl_wait, !fl->fl_next);
-  if (!error)
-   continue;
-
-  locks_delete_block(fl);
-  break;
- }
- return error;
+ return do_lock_file_wait(filp, fl, flock_lock_file, NULL);
 }

 EXPORT_SYMBOL(flock_lock_file_wait);
@@ -1728,9 +1716,15 @@ int vfs_lock_file(struct file *filp, uns
 }
 EXPORT_SYMBOL_GPL(vfs_lock_file);

+static int fcntl_lock_fn(struct file *filp, struct file_lock *fl, void *arg)
+{
+ return vfs_lock_file(filp, (unsigned int)arg, fl, NULL);
+}
+
 /* Apply the lock described by l to an open file descriptor.
  * This implements both the F_SETLK and F_SETLKW commands of fcntl().
  */
+
 int fcntl_setlk(unsigned int fd, struct file *filp, unsigned int cmd,
   struct flock __user *l)
 {
@@ -1788,18 +1782,7 @@ again:
  if (error)
   goto out;
```

```
- for (;;) {
-  error = vfs_lock_file(filp, cmd, file_lock, NULL);
-  if (error != -EAGAIN || cmd == F_SETLK)
-   break;
-  error = wait_event_interruptible(file_lock->fl_wait,
-    !file_lock->fl_next);
-  if (!error)
-   continue;
-
-  locks_delete_block(file_lock);
-  break;
- }
+ error = do_lock_file_wait(filp, file_lock, fcntl_lock_fn, (void *)cmd);

  /*
   * Attempt to detect a close/fcntl race and recover by
@@ -1912,18 +1895,7 @@ again:
  if (error)
   goto out;

- for (;;) {
-  error = vfs_lock_file(filp, cmd, file_lock, NULL);
-  if (error != -EAGAIN || cmd == F_SETLK64)
-   break;
-  error = wait_event_interruptible(file_lock->fl_wait,
-    !file_lock->fl_next);
-  if (!error)
-   continue;
-
-  locks_delete_block(file_lock);
-  break;
- }
+ error = do_lock_file_wait(filp, file_lock, fcntl_lock_fn, (void *)cmd);

  /*
   * Attempt to detect a close/fcntl race and recover by
```