Directory Locking

Locking scheme used for directory operations is based on two kinds of locks - per-inode (->i_rwsem) and per-filesystem (->s_vfs_rename_mutex).

When taking the i_rwsem on multiple non-directory objects, we always acquire the locks in order by increasing address. We'll call that "inode pointer" order in the following.

For our purposes all operations fall in 5 classes:

1) read access. Locking rules: caller locks directory we are accessing. The lock is taken shared.

2) object creation. Locking rules: same as above, but the lock is taken exclusive.

3) object removal. Locking rules: caller locks parent, finds victim, locks victim and calls the method. Locks are exclusive.

4) rename() that is _not_ cross-directory. Locking rules: caller locks the parent and finds source and target. In case of exchange (with RENAME_EXCHANGE in flags argument) lock both. In any case, if the target already exists, lock it. If the source is a non-directory, lock it. If we need to lock both, lock them in inode pointer order. Then call the method. All locks are exclusive. NB: we might get away with locking the source (and target in exchange case) shared.

5. link creation. Locking rules:

    - lock parent
    - check that source is not a directory
    - lock source
    - call the method.

All locks are exclusive.

6) cross-directory rename. The trickiest in the whole bunch. Locking rules:

- lock the filesystem
- lock parents in "ancestors first" order.
- find source and target.
- if old parent is equal to or is a descendent of target fail with -ENOTEMPTY
- if new parent is equal to or is a descendent of source fail with -ELOOP
- If it's an exchange, lock both the source and the target.
- If the target exists, lock it. If the source is a non-directory, lock it. If we need to lock both, do so in inode pointer order.
- call the method.

All ->i_rwsem are taken exclusive. Again, we might get away with locking the source (and target in exchange case) shared.

The rules above obviously guarantee that all directories that are going to be read, modified or removed by method will be locked by caller.

If no directory is its own ancestor, the scheme above is deadlock-free.

Proof:

> First of all, at any moment we have a partial ordering of the objects - A < B iff A is an ancestor of B.
>
> That ordering can change. However, the following is true:

1. if object removal or non-cross-directory rename holds lock on A and attempts to acquire lock on B, A will remain the parent of B until we acquire the lock on B. (Proof: only cross-directory rename can change the parent of object and it would have to lock the parent).
2. if cross-directory rename holds the lock on filesystem, order will not change until rename acquires all locks. (Proof: other cross-directory renames will be blocked on filesystem lock and we don't start changing the order until we had acquired all locks).
3. locks on non-directory objects are acquired only after locks on directory objects, and are acquired in inode pointer order. (Proof: all operations but renames take lock on at most one non-directory object, except renames, which take locks on source and target in inode pointer order in the case they are not directories.)

Now consider the minimal deadlock. Each process is blocked on attempt to acquire some lock and already holds at least one lock. Let's consider the set of contended locks. First of all, filesystem lock is not contended, since any process blocked on it is not holding any locks. Thus all processes are blocked on ->i_rwsem.

By (3), any process holding a non-directory lock can only be waiting on another non-directory lock with a larger address. Therefore the process holding the "largest" such lock can always make progress, and non-directory objects are not included in the set of contended locks.

Thus link creation can't be a part of deadlock - it can't be blocked on source and it means that it doesn't hold any locks.

Any contended object is either held by cross-directory rename or has a child that is also contended. Indeed, suppose that it is held by operation other than cross-directory rename. Then the lock this operation is blocked on belongs to child of that object due to (1).

It means that one of the operations is cross-directory rename. Otherwise the set of contended objects would be infinite - each of them would have a contended child and we had assumed that no object is its own descendent. Moreover, there is exactly one cross-directory rename (see above).

Consider the object blocking the cross-directory rename. One of its descendents is locked by cross-directory rename (otherwise we would again have an infinite set of contended objects). But that means that cross-directory rename is taking locks out of order. Due to (2) the order hadn't changed since we had acquired filesystem lock. But locking rules for cross-directory rename guarantee that we do not try to acquire lock on descendent before the lock on ancestor. Contradiction. I.e. deadlock is impossible. Q.E.D.

These operations are guaranteed to avoid loop creation. Indeed, the only operation that could introduce loops is cross-directory rename. Since the only new (parent, child) pair added by rename() is (new parent, source), such loop would have to contain these objects and the rest of it would have to exist before rename(). I.e. at the moment of loop creation rename() responsible for that would be holding filesystem lock and new parent would have to be equal to or a descendent of source. But that means that new parent had been equal to or a descendent of source since the moment when we had acquired filesystem lock and rename() would fail with -ELOOP in that case.

While this locking scheme works for arbitrary DAGs, it relies on ability to check that directory is a descendent of another object. Current implementation assumes that directory graph is a tree. This assumption is also preserved by all operations (cross-directory rename on a tree that would not introduce a cycle will leave it a tree and link() fails for directories).

Notice that "directory" in the above == "anything that might have children", so if we are going to introduce hybrid objects we will need either to make sure that link(2) doesn't work for them or to make changes in is_subdir() that would make it work even in presence of such beasts.