Ext3 Inode Versioning

Andreas Dilger

2006-12-13

1 Introduction

This design is focussed on the on-disk implementation of inode versions for the ext3 filesystem used by Lustre. Some parts of the design are mandatory (specifically those in the Requirements section) while others are included in order to make the implementation more generally useful and acceptable to the ext3 community at large. Prototype implementations of parts of this design have previously been circulated on the ext2-devel mailing list, though no full implementation exists.

This design document was based on the vanilla 2.6.18 kernel sources and patches for particular kernels may vary in the functions where the inode versions are set.

2 Requirements

The **inode version** is a persistent (on-disk), monotonically increasing integer stored with each inode to track any changes made to an inode. It is not required that two changes to the same inode have sequential version numbers, only that two versions of one inode can be compared to order two or more changes to that inode relative to each other.

The inode version is also be used to compare the relative modifications between two different inodes, and as such must be based on a global value instead of a per-inode value.

3 Functional specification

The inode version can be implemented as a 64-bit integer per inode. This version needs to be stored atomically with each inode update on disk so that it is available to determine whether changes to an inode are on disk after a crash.

In memory the version is stored as a full 64-bit field in the ext3-private part of the inode `i_fs_version`. This is kept separate from the generic inode `i_version`, which is modified by the kernel and would conflict with Lustre's inode version.

The on-disk representation of the inode version is as follows:

```
struct ext3_inode {
    __u32 osd1.linux1.l_i_reserved1;  /* this is low 32 bits of version */
    :
    :
    __le16 i_extr6_isize;
    __u16 i_pad1;
    __le32 i_ctime_extra;
    __le32 i_mtime_extra;
    __le32 i_atime_extra;
    __le32 i_crtime;
    __le32 i_crtime_extra;
    __le32 i_version_hi;          /* this is the high 30 bits of version */
}
#define i_disk_version osd1.linux1.l_i_reserved1
#define EXT4_FEATURE_RO_COMPAT_EXTRA_ISIZE   0x0040
```

The `i_disk_version` field holds the low 32 bits of the version, and is always guaranteed to be present as it is in the "original" part of the ext3 inode. The `i_version_hi` field holds the high 32-bits of the version on disk and will normally be present, but in certain circumstances (e.g. old filesystems that didn't have large enough inodes, or all of the EA space is full) it is possible that this field cannot be stored on disk.

In `ext3_do_update_inode()` the version is stored to disk as follows:

```
#define EXT3_EPOCH_BITS 2
#define EXT3_EPOCH_MASK ((1 << EXT3_EPOCH_BITS) - 1)
#define EXT3_NSEC_MASK (~0UL << EXT3_EPOCH_BITS)
raw_inode->i_disk_version = cpu_to_le32(ei->i_fs_version & 0xffffffff);
if (ei->i_extra_isize) {
    :
    ext3_expand_extra_isize(raw_inode, offsetof(i_version_hi) - EXT3_GOOD_OLD_INODE_SIZ
    if (EXT3_GOOD_OLD_INODE_SIZE + ei->i_extra_isize > offsetof(i_version_hi))
        raw_inode->i_version_hi = cpu_to_le32(ei->i_fsversion >> 32);
}
```

In `ext3_read_inode()` the version is read from disk as follows:

```
ei->i_fs_version = le32_to_cpu(raw_inode->i_disk_version);
```

```
if (EXT3_INODE_SIZE(inode->i_sb) > EXT3_GOOD_OLD_INODE_SIZE) {
    :
    if (EXT3_GOOD_OLD_INODE_SIZE + ei->i_extra_isize > offsetof(i_version_hi))
        ei->i_fs_version |= (__u64)(le32_to_cpu(raw_inode->i_version_hi) << 32);
}
```

The ext3 filesystem code needs the ability to increase the extra inode size for existing inodes that may have `i_extra_isize` below the limit needed to store `i_crtime_extra` (or other extra extra inode fields), so we need a helper function to shift any EAs beyond the current `i_extra_isize` to make room for the fixed fields. This would only be necessary for files created before this feature was implemented:

```
int ext3_expand_extra_isize(struct ext3_inode_info *ei, struct ext3_inode *raw_inode,
                            int new_extra_isize)
{
    if (new_extra_isize > inode->i_extra_isize)
        return 0;
    if (no EA data)
        ei->i_extra_isize = new_extra_isize;
        memset((char *)raw_inode + EXT3_GOOD_OLD_INODE_SIZE, 0, new_extra_isize);
        return 0;
    shift EA data;
    ei->i_extra_isize = sb->s_want_extra_isize; /* >= offsetof(i_version_hi) */
}
```

A new Lustre filesystem method should be created to get the inode version:

```
__u64 fsfilt_ext3_get_version(struct inode *inode)
{
    return EXT3_I(inode)->i_fs_version;
}
```

Another Lustre filesystem method should be created to set the version of an inode:

```
void fsfilt_ext3_set_version(struct inode *inode, __u64 new_version)
{
    EXT3_I(inode)->i_fs_version = new_version;
}
```

This should only be called by the Lustre server before the inode is marked dirty so that the on-disk fields can be updated before the inode writeout. Lustre will specify the exact version number to use (possibly the transno for the current operation) to avoid dependencies in the on-disk filesystem. This also avoids issues of non-invasive operations like object defragmentation, etc from changing the inode version inadvertently.

4 Use cases

1. The inode version needs to be saved and restored for each operation to the filesystem.
2. The inode version must be persistent, so an umount, remount of the filesystem will produce the same inode version.
3. Verify that the inode expansion is handled correctly.

5 Logic specification

There are 3 distinct phases in implementation that can be completed in order to speed initial delivery/testing:

1. add `i_version_hi` and preceding to extra fields to `struct ext3_inode`, and for all new inodes in `ext3_new_inode()` we should always create inodes with these fixed fields already in place, assuming filesystem is formatted with large inodes. The inode's new version fields need to be written to disk in `ext3_mark_inode_dirty()`.
2. add support to fsfilt_ext3 based on changes made by Zhang in bug 10609 to get/set the 64-bit inode version in Lustre. The inode version actually needs to be updated before the transaction is committed in mds_finish_transno() otherwise the version update may not be atomic and could be lost even though the change was actually completed.
3. add support for growing the `i_extra_isize` on-the-fly if it is not large enough to hold `i_version_hi`, including pushing one or more existing EAs from the inode to an external EA block (preferrably not the lov_stripe_md).

6 State management

6.1 State invariants

It is up to the caller to ensure that the version is monotonically increasing. It is up to the caller to store the maximum fs-wide version number and is stored in a persistent manner.

6.2 Scalability & performance

The actual setting of the `i_version_hi` and `i_disk_version` field is not expected to noticably impact performance in any way. These fields are already being written to disk and the inode is in-core so no extra IO is needed.

One potential performance impact would be if the larger inode forces extended attributes out of the large inode and into a disk block. With the currently proposed extra 6 fields (`i_version_hi` in the core inode) there is not enough space in a 256-byte inode with a LOV EA for 2 stripes. Lustre has been formatting with 512-byte inodes on the MDS for some time now, so hopefully the real world impact is low.

Another potential performance impact is if the additional dirtying of the inodes in order to set the version fields. If this becomes a factor it would be possible to mitigate this by changing the `ext3_mark_inode_dirty()` codepath to avoid copying the inode into the disk buffer and instead only mark the inode dirty in memory and have a journal pre-commit callback copy the dirty inode(s) into the buffer before the buffer is written to the journal at commit time. Such a pre-commit callback mechanism would also be useful for other planned features like nanosecond timestamps, and checksums for inode, bitmap, and group descriptors.

6.3 Recovery changes

Any on-disk version recovery should be handled as part of normal ext3 journal recovery.

6.4 Locking changes

There may need to be locking of the version in the core inode if the server code drops the inode lock before the version is updated and the new version number is not yet available.

6.5 Disk format changes

The on-disk format of ext3 will be changed in order to store the extra inode fields.

The on-disk inode would get new `i_ctime_extra`, `i_mtime_extra`, `i_atime_extra`, `i_crtime`, `i_crtime_extra` and `i_version_hi` fields. These fields already have upstream approval, though no patch that uses them is yet accepted. The proposal to save `i_disk_version` into `ost1.linux1.l_i_reserved1` is also approved upstream (there is a need for a similar 64-bit version number in NFSv4).

The superblock has added `s_want_extra_isize` and `s_need_extra_isize` (all new inodes must have at least `s_need_extra_isize` bytes of `i_extra_isize`. and `s_want_extra_isize` if possible). This was added for the nanosecond timstamp patch, and should be honored for this request.

6.6 Wire format changes

This design is only concerned with the on disk format changes. The full version-based recovery design should be consulted for any wire protocol changes needed by the inode

version field. With the addition of on-disk nanosecond timestamps, there may be a desire to also export these timestamps via llite to the applications. It appears we would need to have a CONNECT flag in order to handle this properly, otherwise the clients would have the high 32 bits of the seconds set with the nanoseconds.

6.7 Protocol changes

This design is only concerned with the on disk format changes. The full version-based recovery design should be consulted for the use of the enhanced ctime/inode version during normal operations and recovery.

6.8 API changes

Addition of fsfilt_ext3_setversion() and fsfilt_ext3_getversion() methods. This does not affect the kernel API, just the lustre->{ext3,ldiskfs} API.

7 Alternatives

Using a numeric 32-bit integer (the nanosecond ctime) to disambiguate inodes with the same ctime seconds value was proposed by Bull for NFSv4 use. This doesn't allow meaningful comparisons between two inodes that have the same ctime seconds, while the nsec field allows comparisons to within a few nsec of each other (assuming the kernel clock has nanosecond accuracy). The drawback is that ctime is not guaranteed to be monotonic (e.g. if the system clock is changed to some time in the future and/or then set backward) so this breaks one of the invariants for the inode version.

8 Focus for inspections