FID to Inode Translation HLD

Yury Umanets

12th March 2007

Contents

1 Introduction

There is the task to generate lustre client inode numbers. In former versions of lustre MDS server backing store FS inode numbers and generations were used. This solution was found to be not suitable for many reasons (collisions in CMD, backing store inodes exposition, many others). Additionally, FIDs were introduced as main and unique object identifier in lustre.

All these reasons were enough to consider about new client inode numbers generation solutions, which could possibly fix all flaws about former solution. The prototype of new generation policy was created in CMD3 project. It is based on FID, as the unique object id and may be considered as simple flatten

of two dimensional FIDs array (seq and oid inside seq). However, it has some downsides which should be fixed in this work and thus prepare it for production.

Downsides are the following:

- Compatibility is not supported, new solution should handle igifs (igif is FID which contains inode number of generation of backing store FS instead of seq and oid in its fields. Read FIDs HLD for details);
- Overflowing and thus re-using already existing inode numbers;
- In some working patterns, for instance, where recovery is involved and thus, sequence switch is performed quite often, number of generated inodes is limited. Week of running test 11 could exhaust it. This is completely not acceptable for production and should be fixed/improved.

2 Requirements

In this work we should fix downsides described in 1. This is supposed to be done be means of using the following ideas:

- Igifs support should be added;
- Minimize possible collisions and overflows;
- Use FID as a base for generating client inode numbers;
- Generate 32 or 64-bit inode numbers (not all platforms and applications support 64-bit inode numbers).

3 Functional specification

To meet requirements we should do the following:

- Add igif support by simple check in inode mapping function. If passed FID is igif - put its components (may be only inode number) to result value;
- FID's components (seq and oid) are used as base for generating inode number;
- All overflows (not all clients support 64-bit also there may exist such a big sequences, that using them for generating numbers causes overflow of 64-bit value) may be fixed by using convenient hash function with small number of collisions. All clients should use same algorithm for inode generation. This is important, because some clients may support 64-bit inodes and others may not. Until all clients support 64-bit - we use 32-bit inode numbers;

Read section 5 for details.

4 Use cases

The following use cases should be worked out. If they do not pass - we can use another hashes and find one which is optimal from point of view of using in production.

4.1 Checking level of collisions

In this case we would like to check if new algorithm generates unique inode numbers and how often it does generate collisions.

1. Mount lustre;
2. Create sub-directory "1a" in root directory;
3. Create 10M files (using mknod) in directory "1a" with names equal inode numbers;
4. If there is the collision - we will not be able to add new entry - test will not pass;

4.2 Checking level of collisions in recovery

This should check if collisions come more often if do recovery and thus loss sequences.

1. Mount lustre;
2. Create sub-directory "1b" in root directory;
3. Run the loop in which we do one recovery and creating 1000 files in directory "1b" (using mknod) after recovery and so on until 10M files are creates in "1b". File names should be same as inode numbers;
4. The loop should survive such a many counts that enough to create 10M files. If it does not - test fails.

4.3 Number of unique inodes

This test should check how many unique inode numbers may be created in simplest working pattern - create files in directory.

1. Mount lustre;
2. Create sub-directory "1c" in root directory;
3. Run infinite loop with creating files in directory "1c" (using mknod). File names should be same as inode numbers;

4. Test is finished when new file will not be able to create due to -EEXIST error, which should indicate that dentry with such a name already exists in directory "1c";

5. Calculate number of created files.

4.4 User-space applications may work with inode generation policy

This test should check if current inode generation policy is compatible with user space applications which usually use inode numbers. They are tar, rsync, ls, cpio, find. May be some else.

1. Mount lustre;

2. Run "tar" on lustre in most of modes (tar, untar, etc) to check if it works well;

3. Run "rsync" on lustre;

4. Run "ls" on lustre;

5. Run "cpio" on lustre;

6. Run "find" on lustre;

7. Run something else if any. Test fails if any of above fails.

5 Logical specification

There are few ways to achieve requirements:

1. Use 32-bit inode numbers if not all clients in cluster support 64-bit. This is going to be controlled by configure key and thus, is chosen in compilation time - disabled version is not going to even compile. In this case we have to check if all applications may work with 32-bit inodes - see section 4.4;

2. Use 64-bit inode numbers if all clients in cluster and their user-space applications in supported kernels may use 64-bit inode numbers. We need to check if tar/rsync/ls/cpio/find work with 64-bit in this case - see section 4.4;

Here is the main algorithm which we are going to use in both cases with only disabling some part of it if needed. In this algorithm we always work with 64-bit numbers internally, regardless what does client support (32 or 64-bit inode numbers). And later algorithm may yeild 32-bit values (if it is enabled), so that, 64-bit value will be converted to 32-bit one by means of using appropriate hash function.

Algorithm looks like the following:

- First of all check for if passed FID igif or not. If it is igif - we put its components as inode and generation without any conversions;
- Generate object presentation number (OPN) from FID's components. This may be done by using this approach: *if (seq < 2^47) opn = seq << 16 | oid; else opn = tea_hash(fid, 64).* As it may be seen, *opn* never overflows 64-bit. In the case, seq number is such a big that may cause overflow - we use *tea_hash(fid)* for generating *opn*. Function *tea_hash(fid)* is slow, though yields minimal number of collisions;
- Use *tea_hash()* function against *opn* like the following*: ino = tea_hash(opn, 32).* This generates 32-bit inode number with minimal number of collisions;

Function prototype may look like the following:

```
ino_t ll_fid_build_ino(struct lu_fid *fid)
{
    if (fid_is_igif(fid))
        return fid_pack_igif(fid);

#if BITS_PER_LONG == 64
    return (fid_seq(fid) < ((__u64)2 << 47) ?
        fid_seq(fid) << 16 | fid_oid(fid) : tea_hash(fid, 64));
#else
    return tea_hash(opn, 32);
#endif
}
```

Note the following points about algorithm above:

1. Collisions in use-space are still possible, we find it difficult (impossible?) to avoid them. But in kernel we can handle them by means of additional FIDs comparison in *iget5_locked()*;
2. Igifs check should additionally make sure what is current compatibility mode, for some versions all inode numbers may be igifs and other parts of algorithm may be disabled;
3. We track is sequence is < than *(2^47) - 1* because we want to reserve 16-bit for oid component and sequences > than *(2^47) - 1* would give overflow of 64-bit value.

6 State management

6.1 Scalability & Performance

No performance changes expected.

As for scalability, the follow improvements are expected:

- Inode numbers generation will not depend directly on working pattern;
- More inode numbers may be generated;
- Overflows and re-using is minimized.

6.2 Recovery changes

No recovery changes are expected.

6.3 Wire format changes

No changes in wire format.

6.4 Protocol changes

No changes in protocol.

6.5 API changes

No changes in API.