LFSCK2 High Level Design

1 Introduction

On a Lustre* file system a normal non-directory file is composed of one MDT-object (called a **parent**) and several (0 - N) OST-objects (called **children**). The parent resides on the MDT, and records the file layout information for every child belonging to the file. With the file layout information a client can locate the specified OST-object. To guarantee the data integrity each child on related OST also records its parent MDT-object FID information to indicate which file the OST-object belongs to. Typically the file layout information stored in the parent should be consistent with the parent MDT-object FID information stored in its children. On a production system however, inconsistencies may occur caused by some system errors and failures. Possible inconsistencies include:

- MDT-object with dangling reference
  The MDT-object1 claims that the OST-object1 is its child, but on the OST, the OST-object1 does not exist, or it is not materialized (so does not recognize the MDT-object1 as its parent).
- Unmatched referenced MDT-object/OST-object pair
  The MDT-object1 claims that the OST-object1 is its child, but the OST-object1 claims that its parent is the MDT-object2 rather than the MDT-object1. On the MDT, the MDT-object2 does not exist, or not recognize the OST-object1 as its child. An additional case exists where the child index stored in the parent layout information does not match the index information stored in the child itself.
- Multiple referenced OST-object
  The MDT-object1 claims that the OST-object1 is its child, but the OST-object1 claims that its parent is the MDT-object2 rather than the MDT-object1. On the other hand, the MDT-object2 recognizes the OST-object1 as its child.
- Unreferenced OST-object
  The OST-object1 claims that the MDT-object1 is its parent, but on the MDT, the MDT-object1 does not exist yet, or it does not recognize the OST-object1 as its child.

In fact, the layout information stored in the parent layout EA contains self-check information, the `lov_mds_md.lmm_oi`, which stores the MDT-object FID, with them we can know which file the layout EA belongs to. In LFSCK phase II, we should verify whether such information is self-consistent or not, since it may have become stale over backup/restore. Similarly, the OST object also has self-check information (`filter_fid.ff_objid`/`ff_group` or LMA) stored together with the parent information that needs to be verified also.

Another kind of filesystem consistency between MDT and OST is related with quota. On a Lustre file system, each object, is spite of MDT-object or OST-object, has the owner attribute: UID and GID, to indicate which user the object belongs to. If the owner attributes for the objects belonging to the same file are not consistent, then quota will be incorrectly reported to the system administrators.

As the second phase of the new LFSCK, it needs to find out and repair above inconsistent cases during the system running with normal external services. The design and implementation will be done against master branch and should be OSD neutral, though it will only be tested against osd-ldiskfs.

2 Find out MDT-OST inconsistency

The LFSCK for MDT-OST consistency needs to scan the whole system both MDT(s) and OST(s) to discover inconsistencies. Completeness and efficiency are the main requirements for the LFSCK system scanning. To achieve comprehensive scanning of the whole system, we will introduce a two-stage scanning method.

2.1 First-stage scanning - Verify layout and OST object state

The first-stage system scanning is driven by the LFSCK on MDT side.

The master MDT first sends an RPC to all of the OSTs start their local otable-based iteration of objects in order to perform their OI Scrub in parallel. This will allow the OSTs to verify or rebuild their local OI files (the `O/{seq}/dnn` directories and `LAST_ID` file) and the "self FID" for each object (stored in either `lma` xattr for 2.4+ OSTs or `fid` xattr for older OSTs since Lustre software 1.6). At this time LFSCK on the OST will also generate a local in-memory orphan object index (described in more detail below) containing the FID of all objects returned from its otable-based iteration (excluding `FID_SEQ_LOCAL_FILE` and `FID_SEQ_LOCAL_NAME` and similar internal objects). This orphan object index initially contains all of the object FIDs that exist on the OST at the start of LFSCK.

The master MDT also scans the local MDT device sequentially by reusing the existing otable-based iteration that has been implemented in LFSCK Phase I. For each striped file, it checks the child objects in the layout on their OSTs. The MDT otable-based iteration is sequential from the MDTs point of view, but from the OST side the MDT's first-stage scanning of OST objects is in random order because the MDT on-disk object order is not related to the OST on-disk object order.

As each OST-object is accessed (whether by the MDT LFSCK, or by normal object access from the client) the corresponding object FID is removed from the orphan object index. The simplest approach to maintaining consistency between the OST's orphan object index and the MDT's object access is to complete the OST first-stage processing *before* the the MDT first-stage processing is *started*. This would not increase the work done by LFSCK (still a single stage on the OST and MDT), but would increase the wallclock time for a full scan somewhat. It is expected that the OST first-stage processing to take significantly less time than that of the MDT(s), since the OSTs have fewer objects and would only be doing local operations, while the MDT(s) contain many more objects and will need to do network operations. An optimization to allow parallel first-stage scanning of the MDTs and OSTs (possibly for LFSCK Phase IV Performance Optimization) is if the OSTs keep a separate list of unique object FIDs accessed from the start to the finish of the first-stage OST-object iteration, and process them locally at the end of the OST first-stage iteration. This would remove any OST-objects from the orphan object index that were not in the index at the time that the MDT processed them.

The MDT-driven OST-object access is not expected to be a significant performance impact for LFSCK because there are typically many more OSTs with correspondingly fewer objects and can do object lookups in parallel if the objects are not already in the correspondingly larger aggregate OST cache. The OST OI Scrub will also serve to pre-fetch the OST-objects into cache with optimized disk accesses (if they fit) to reduce the chance that the random access from the MDT becomes seek bound.

After the MDT first-stage system scanning has completed the following file inconsistencies are identified:

- MDT-object with dangling reference
- Unmatched referenced MDT-object/OST-object pair
- Multiple referenced OST-object
- MDT-object with self-inconsistent layout EA
- OST-object with wrong owner

The OST orphan object index will contain only OST-objects that existed at the start of the first-stage scanning, but were not referenced during the entire LFSCK run by either the MDT LFSCK or normal client accesses.

2.2 Second-stage scanning - Find unreferenced OST-objects

The second-stage scanning is only started once the MDT and OST first-stage scanning has completed.

To identify unreferenced OST-objects, we need to know if the OST-object is referenced by some MDT-object or not. How can we know? The simplest way would be for the OST to query the MDT with the parent FID for each OST-object as it is seen during its first-stage iteration. This would be extremely undesirable if all the MDT-objects would be accessed again during the second-stage scan, since the scanning time would increase significantly due to many random MDT lookups from multiple OSTs, and would make a full LFSCK take much more than double the processing time. The LFSCK first-stage MDT scanning has already accessed all of the OST-objects that are referenced by related MDT-object layouts. Similarly, there may be OST-objects that have been accessed by normal RPCs from clients during the LFSCK first-stage scanning, they are or were recently also referenced by related MDT-objects (otherwise, clients cannot know how to access them). It is important to handle the OST-objects accessed by normal client RPCs, since they may have been unlinked from the MDT before LFSCK could process them or migrated to another MDT during the first-stage processing.

This is what the OST orphan object index is for. At the end of the MDT and OST first-stage scanning the OST orphan object index will contain zero or more OST-objects that were never accessed during the first-stage system scanning. The orphan object index will not include any objects that were created after LFSCK started processing, nor will it contain objects that were destroyed (since this would also be considered an object access). Through the second-stage system scanning, we can iterate over the orphan object index to find out all the unreferenced OST-objects that existed at the start of LFSCK, and need to be repaired. LFSCK verifies whether its parent MDT-object references it or not. From the MDT side view, the second-stage system scanning is in random order, though it is expected that only a small number of such orphaned OST objects would be found during any scan. If there are no inconsistencies in the distributed filesystem (hopefully the normal state) the second-stage processing will not need to do any work.

To maintain the orphan object index, on each OST has a logical in-memory index of existing OST FIDs. The index will use the OST-object FID as the key, and will return the MDT parent FID and stripe index as the value (the `fid` xattr on the OST). LFSCK on the MDT can iterate over this index on the OST. Each MDT parent FID will be verified, and either the OST object will be re-linked to an existing file or a new file in `lost+found` by the MDT (default action), or destroyed, depending on administrator policy.

Usually, the OST-object is identified by FID (or IDIF), and the FID space is huge and sparse, and we need to maintain one FID for each object that exists on the OST, which may be tens or hundreds of millions of objects. It may be undesirable to use a full index in RAM on the OST for

LFSCK directly. Instead, an rbtree-based sparse bitmap will be used. According to current FID allocation policy, there are at most (2^{32} - 1) FIDs that can be allocated for a FID sequence, but typically it will contain far fewer. Also, only one FID sequence will normally be in use by any MDT. It needs 512M Bytes RAM to trace all the single sequence FIDs with single bitmap. Since FID space is clustered (many objects typically are created and destroyed together), we prefer to use some separated and small bitmaps to trace these sparse FIDs belonging to the single sequence. On the other hand, smaller bitmap means more leaves in the rbtree, and the leaf descriptors also occupy some RAM. And more leaves also means more deep rbtree, which will affect the operations efficiency on the rbtree. So it is not always true that smaller bitmap is better. Currently, we plan to use up to 32768 bits (4k Bytes) in each leaf block to match the `PAGE_SIZE` for easy allocation, but the actual bitmap for each leaf could be dynamically allocated to save memory if only a few objects are allocated in that range. The actual implementation of the orphan object index is internal to the OSD, and presents the normal OSD index API to users so can be changed in the future as necessary.

3 LFSCK tracing

The LFSCK phase II will share the same on-disk file `lfsck_bookmark` to store some parameters. It will introduce a new local file, named **`lfsck_layout`** on the MDT to trace the LFSCK phase II processing. The LFSCK phase II status, statistics, checkpoint, and so on, will be recorded in the new file. It can be used for querying from user space, and also for resuming the LFSCK phase II from breakpoint. From a performance perspective it is undesirable to update the file `lfsck_layout` for each object processed. Instead, we cache the update in RAM, and write the updating to disk periodically. Such write will be processed through backend journal asynchronously.

On the other hand, the second-stage system scanning will be driven by the LFSCK on OST. It also needs a local `lfsck_layout` file on the OST to trace OST side LFSCK processing. It will use similar on-disk layout and update/write mechanism as the “lfsck_layout” file on the MDT side does.

- **`lfsck_layout`**
  This is the `lfsck_layout` file on-disk structure:

```
struct lfsck_layout {
        lfl_magic;
        lfl_status;
        /* Time for the last LFSCK completed. */
        lfl_time_last_complete;

        /* Time for the latest LFSCK ran. */
        lfl_time_latest_start;

        /* Time for the last LFSCK checkpoint. */
        lfl_time_last_checkpoint;

        /* Position for the latest LFSCK started from. */
        lfl_pos_latest_start;
        /* Position for the last LFSCK checkpoint. */
        lfl_pos_last_checkpoint;

        /* Position for the first should be updated object. */
        lfl_pos_first_inconsistent;
        /* How long the LFSCK has run. */
        lfl_run_time;

        /* How many completed LFSCK ran on the system. */
        lfl_success_count;

 ...
        /* OST/MDT count */
        lfl_target_count;

        /* Which OSTs/MDTs completed the LFSCK. */
        lfl_target_bitmap;
};
```

This is a special tracing file MDT-OST consistency check/repair, independent from other LFSCK phases.

- **`lfsck_layout.status`**
  For LFSCK current status, as following:

```
enum lfsck_status {
 LS_INIT,

 /* first-stage system scanning driven by MDT */
 LS_FIRST_STAGE,

 /* second-stage system scanning driven by OST */
 LS_SECOND_STAGE,

  LS_COMPLETED,

 /* Some OST (or MDT for DNE mode) failed during the LFSCK, or not join the
LFSCK. */
 LS_PARTIAL,

 /* The LFSCK exited automatically for some failure, will not auto restart. */
 LS_FAILED,

 /* The LFSCK is stopped manually, will not auto restart. */
 LS_STOPPED,

 /* The LFSCK is auto paused when umount, can be auto restarted when remount. */
 LS_PAUSED,

 /* System crashed during the LFSCK, can be auto restarted after recovery. */
 LS_CRASHED,
};
```

LS_PAUSED: The administrator did not stop the LFSCK by "lctl lfsck_stop" explicitly. Instead, the LFSCK has to be paused automatically when the device umounted.

LS_CRASHED: There is system crash (can be caused by LFSCK, and can be NOT) during the LFSCK processing, and cause the LFSCK status LS_FIRST_STAGE or LS_SECOND_STAGE cannot be updated. After the system restart, it will detect that the former LFSCK status is in LS_FIRST_STAGE or LS_SECOND_STAGE, then it will set the LFSCK status as LS_CRASHED.

Each time when the MDT mounts up, it will auto check whether need to restart the LFSCK. If in the status of LS_PAUSED or LS_CRASHED , then the LFSCK will be restarted from the breakpoint.

- **lfsck_layout.lfl_pos_last_checkpoint**

  It is used for recording the position corresponding to the oldest to be committed transaction for LFSCK repairing when making the checkpoint, including both OST side transactions and MDT side transactions. If there is no more LFSCK repairing transaction to be committed, then the position corresponding to the latest processed MDT-object (for MDT side `lfsck_layout`) or OST-object (for OST side `lfsck_layout`) will be recorded. For MDT `lfsck_layout`, it is only useful during the first-stage system scanning driven by the MDT; for OST side `lfsck_layout`, it is only useful during the second-stage system scanning driven by the OST.

- **lfsck_layout.lfl_target_bitmap**
  As distributed consistency check/repair, it is normal that some components cannot join the LFSCK, or failed during the LFSCK. The whole LFSCK process should not be blocked for them. So when LFSCK is finished, the cases may be like that: some OSTs participated in the whole LFSCK from the beginning to the end; some OSTs participated in the LFSCK but failed for system crash or network broken, which may re-joined the LFSCK after recovery, but may not; some OSTs totally missed the LFSCK. So although the LFSCK finished, it cannot guarantee the system consistency, because it was not completed. We need to run another LFSCK some time later. Nobody can guarantee no OST(s) failed when next LFSCK run, but it is unlikely that the same OST would fail in subsequent LFSCK runs. On the other hand, it is meaningless to run the LFSCK on the OSTs that participated in former LFSCK successfully; we should skip such OSTs when next LFSCK run.
    - For MDT, the `lfsck_layout.lfl_target_bitmap` is used to record on which OSTs the LFSCK for MDT-OST consistency check/repair has run completely. Before all the OSTs ran the LFSCK completely, the LFSCK status on the MDT cannot be LS_COMPLETED, may be LS_PARTIAL.
    - For OST, the `lfsck_layout.lfl_target_bitmap` has some different purpose. It will record which MDTs have

successfully completed the LFSCK for related MDT-OST consistency check/repair. Before the MDT(s) run the LFSCK completely, the LFSCK status on the OST cannot be `LS_COMPLETED`, may be `LS_PARTIAL`. Under non-DNE mode, it is useless, because only the single MDT can start the LFSCK for MDT-OST consistency check/repair. In DNE mode, the OST-objects on the OST may belong to the files on different MDTs, and only some MDTs complete the LFSCK for MDT-OST consistency check/repair. Without a complete MDT first-stage scan, the OST cannot guarantee that all the OST-objects are in consistent status, so it will skip any objects belonging to an MDT that has not completed its first-stage scanning.

4 LFSCK user space control

We will try to reuse the existing LFSCK user space tools to control the LFSCK for MDT-OST check/repair. In fact, the existing LFSCK user space framework supports to specify MDT-OST consistency check/repair by specifying the `-t layout` option. For example:

```
lctl lfsck_start –M lustre-MDT0000 –t layout
```

The needed action is to implement kernel space logic to start/stop the LFSCK for MDT-OST consistency check/repair. Each MDT can start and stop verification independently and in parallel. There is no automatic coordination between MDTs and each instance of LFSCK must be started manually on each MDT.

4.1 Query LFSCK processing

We will not introduce special tools for querying the LFSCK processing; instead, it can be done by some new lproc interfaces:

- **`mdd.$fsname-MDTnnnn.lfsck_layout`**
  MDT side lproc interface for querying MDT-OST consistency check/repair processing. Mainly dump the local `lfsck_layout` file on this MDT device.
- **`ofd.$fsname-OSTnnnn.lfsck_layout`**
  OST side lproc interface for querying MDT-OST consistency check/repair processing. Mainly dump the local `lfsck_layout` file on this OST device.

4.2 Auto detect MDT-OST inconsistency and trigger LFSCK

The object-based RPCs to OST, like read/write/punch, already contain both the OST-object FID and related parent MDT-object FID. On OST side, the RPC service threads can optionally check whether the parent MDT-object FID stored in the found OST-object is consistent with the given MDT-object FID or not. This will add an extra layer of integrity checking that the client RPCs are operating on the right OST objects. This checking will be enabled according to a new OFD layer parameter **`fail_on_inconsistency`**, which can be operated by `"lctl {get,set}_param"` command or through the OST side lproc interface **`ofd.$fsname-OSTnnnn.fail_on_inconsistency`**. If set, each client RPC will be checked against the OST object's parent FID for consistency. If unset, the client RPCs will not be verified (current behaviour). This is added checking is not expected to add a significant overhead, since this information is stored directly in every OST inode since Lustre software version 1.6, but will add protection against data corruption caused by misbehaving clients, corruption of data structures, etc. On newly-formatted OSTs, or after an upgraded filesystem has completed its first full LFSCK the **`fail_on_inconsistency`** checking will be enabled by default.

If checking is enabled and parent FIDs match the RPC will be allowed to continue. If the parent FIDs do not match, then the RPC service thread will give it to a dedicated LFSCK thread to handle, and return `-EINPROGRESS` to the client (the client will retry after some time). The dedicated thread will query the MDT whether the client given MDT-object's FID is trusted or not. If the client is incorrect, then the object will be flagged in memory as being verified and the client RPC will fail with `-EIO`. If the client is right, then the LFSCK thread will fix the the OST object's parent FID. If the OST finds a large number of layout inconsistencies, it can ask the MDT to trigger a full LFSCK for the whole system MDT-OST consistency check/repair. Because such check/repair is time-consuming work, it is better to not run the LFSCK often. We can define some thresholds, and only trigger the LFSCK on the whole system when the inconsistency instance exceeds the thresholds. The LFSCK thresholds will be implemented as a new OFD layer parameter **`lfsck_threshold`**, which can be operated by `"lctl {get,set,conf}_param"` command or through the OST side lproc interface **`ofd.$fsname-OSTnnnn.lfsck_threshold`**. In this phase, the **`lfsck_threshold`** will be an wrong/total ratio and/or an absolute number. As long as one of the two facets is exceed, then will trigger the whole system LFSCK. In order to improve the consistency checking, the parent MDT-object FID should be sent by the client in all object-based OST RPCs.

4.3 Speed control

In LFSCK Phase I, we have implemented a basic LFSCK speed control mechanism. The administrator can specify the max speed for the LFSCK to scan the device with the format “N objs/sec” through the MDD layer lproc interface **`mdd.$fsname-MDTnnnn.lfsck_speed_limit`**. We prefer to reuse such mechanism in LFSCK Phase II for controlling the LFSCK speed. Such speed limit will affect not only the MDT but also related OSTs. (**Note:** under DNE mode, the speed limit on the OST specified for one LFSCK instance started on an MDT may be over-written by another LFSCK instance started on another MDT.) Consider the OST device may have different performance from MDT, and the OST may has quite different system load, so there will be separated lproc interface on the OST **`ofd.$fsname-OSTnnnn.lfsck_speed_limit`** to allow the administrator to specify the LFSCK speed on the OST via "`lctl {get,set,conf}_param`" command.

4.4 LFSCK sync status interval

Usually, when the first-stage system scanning is completed on the MDT, the MDT will send RPCs to related OST to notify the OSTs to start the second-stage system scanning. But under some failure cases, the notification RPCs may be not sent out or some OST failed to received such notification RPC. Then related OSTs may falls into waiting forever. On the other hand, because of stripe policy, most of the OST-objects may reside on some OSTs, and other OSTs may quite idle. Then these idle OST may not talk with the MDT for LFSCK progress for a long time. Similar case will happen at the second-stage system scanning, this time, the waiting ones are MDTs. So we need some mechanism to make the MDT and the OST can know the LFSCK progress, but not only wait to be notified. The solution will be like that:

1. In the LFSCK first-stage system scanning, if the OST does not receive RPC from the MDT for LFSCK progress for sometime, it will send RPC to the MDT to query the LFSCK status on the MDT. The status query interval will be controlled thought the new parameter **`lfsck_query_interval`**, which can be adjusted by "`lctl {get,set,conf}_param`" command or through the OFD side lproc interface **`ofd.$fsname-OSTnnnn.lfsck_query_interval`**. The default will be 180 seconds.
2. In the LFSCK second-stage system scanning, if the MDT does not receive RPC from the OST for LFSCK progress for sometime, it will send RPC to the OST to query the LFSCK status on the OST. The status query interval will be controlled thought the parameter **`lfsck_query_interval`**, which can be adjusted by "`lctl {get,set,conf}_param`" command or through the MDD side lproc interface **`mdd.$fsname-MDTnnnn.lfsck_query_interval`**. The default will be 180 seconds.

4.5 LFSCK Repair Policy

The administrator needs to be able to specify the repair policy for LFSCK. For example: Is LFSCK going to repair the filesystem, or only doing a scan? For orphan OST-objects, should they be linked into `lost+found` or deleted immediately. Should files with dangling OST-object refences be repaired by recreating the missing OST-object, or should they be deleted (since their data is missing or corrupt). The administrator can specify the LFSCK behavior via lfsck_start parameters. There will detailed description in the LFSCK UI document.

5 LFSCK engines

The MDT-OST consistency check/repair is driven by a series of kernel threads on both MDT and OSTs, including master engine on MDT and slave engines on OSTs.

5.1 LFSCK master engine

The LFSCK master engine resides on the MDT, and is implemented as a kernel thread in the LFSCK layer. In the LFSCK phase II, the master engine not only controls the slave engines on OSTs, but also drives the first-stage system scanning on the MDT.

1. When the master engine is triggered by the LFSCK user space command (`lctl lfsck_start`) or by an excessive number of MDT-OST inconsistency events, it sends some RPCs to related OSTs to trigger the slave engines.
2. Then the master engine scans the MDT device through low-layer otable-based iteration. For each striped file, it calls the registered LFSCK process handlers to perform related system consistency check/repair (we will describe the detailed processing in subsequent sections).
3. After the MDT completes first-stage system scanning, the master engine sends some RPCs to related OSTs and the master engine waits for the slave engines to complete the first-stage system scanning.
4. The MDT performs second-stage scanning to link orphan objects into the `lost+found` directory, or delete them.

5.2 LFSCK slave engine

The LFSCK slave engine reside on each OST, and will be implemented as a kernel thread in the LFSCK layer. It drives the first-stage system scanning on the OST.

1. When the slave engine is triggered by the RPC from the master engine, it scans local OST device via low layer otable-based iteration to generate an in-memory orphan object index.
2. When the first-stage system scanning (for both MDT side and OST side) finished, it will get the list of non-referenced OST-objects. All the accessed OST-objects during the first-stage LFSCK scanning, in spite of by normal RPCs or by MDT driven LFSCK checking, will be regarded as non-orphans.

Under DNE mode, many MDTs can check/repair MDT-OST consistency in parallel. To avoid multiple scanning the OST device, the slave engine on the OST will not move into the second-stage system scanning until all the master engines completed the first-stage system scanning. For each OST, there is single OST-object accessing bitmap, regardless of how many MDTs are in the MDT-OST consistency check/repair.

6 Repair MDT-OST inconsistency

NOTE: this section only describes the check/repair logic. API and RPC changes are discussed in subsequent sections.

The LFSCK is not clever enough to repair all the found inconsistencies automatically. Under some special cases, it needs some human knowledge involved to determine how to deal with the inconsistency. An interactive mode LFSCK will be inefficient for very large filesystems. Instead, we will create a special directory `lost+found/` on the MDT to hold objects that cannot be repaired automatically. The administrator can review the `lost+found/` entries, and process those sub-items anytime during or after the LFSCK with more human knowledge, like context, time-stamps, owner and some other clues.

Consider DNE mode, each MDT should have its own `lost+found/`, but since those `lost+found/` directories are under the same namespace and visible to any client, they are either under different parent directories or have different names. To simply the processing, the MDT0 will create the unique `.lustre/lost+found/MDT0000` with `FID_SEQ_DOT_LUSTRE`. For each other MDT, it will create its own sub-directory under the `.lustre/lost+found/MDTnnnn` with the MDT index as the name and the normal FID (assigned to the MDT) to guarantee that the sub-directory inode resides on the specified MDT.

6.1 Repair the file which MDT-object has dangling reference

There are two cases for dangling reference:

1. A formerly allocated OST-object is lost. The LFSCK will allocate new OST-object with the specified object external FID and initialize it with the given parent MDT-object FID and owner attributes. Although the new created OST-object is initialized, the SUID + SGID mode will be kept, which will be dropped by the first modification RPC, like write/punch/setattr. Then we can distinguish whether the new create OST-object has been modified or not.
2. The OST-object is there, but it is not initialized, and without SUID + SGID mode set. Then the LFSCK will initialize it with the given parent MDT-object FID and owner attributes.

6.2 Repair unmatched referenced MDT-object/OST-object pair

The MDT-object layout information is trusted over the OST-object back-pointer because it relates to user visible file data. The OST-object back-pointer is only used for internal recovery purposes and is not visible to the user, so does not affect proper file usage information, nor was kept consistent for Lustre software version 1.8.x MDT file-level backup/restore. The LFSCK will update the OST-object to make it recognize the new parent.

6.3 Repair multiple referenced OST-object

There are several options to repair the multiple referenced OST-object:

1. Duplicate the multiple referenced OST-object. It is intuitive solution, but may cause some issues:
    a. Data leak. A malicious user may cheat the MDT to create the file with specified but invalid stripe information. If the specified stripe contains the OST-object FID that belongs to other file, then duplication for repairing the multiple referenced OST-object by the LFSCK will cause data leaking from the victims file, especially when the victims file contains some sensitive information.
    b. Waste resource. Depends on the duplication source size and where to put the target OST-object. The malicious user may use such mechanism to consume system space and network bandwidth.
2. Remove the unrecognized file (MDT-object). To avoid above issues caused by OST-object duplication, we can destroy the unrecognized file. But since we do not know whether such file is still useful or not, simply removing is potentially dangerous. For a multiple striped file,

may be only some of its OST-objects conflict with other file(s). Under such case, the removing may cause normal data loss.

3. Create new empty OST-object for the unrecognized MDT-object. It is simple and reasonable. It will not cause data leaked, and will not waste system resource (only one OST object). And as the LFSCK processing, we may found another unreferenced OST-object, which may claims parent as current unrecognized MDT-object. It is quite possible that the current unrecognized MDT-object and the later unreferenced OST-object belong to the same file. So we prefer to use this solution and process as following:
   Similar as repairing the MDT-object with dangling reference, the LFSCK will allocate new OST-object with the specified object external FID and initialize it with the given parent MDT-object FID and owner attributes. Although the new created OST-object is initialized, the SUID + SGID mode will be kept until some RPC modified (write/punch/setattr) the OST-object. So we can distinguish whether the new create OST-object has been modified or not. The unrecognized MDT-object will reference the empty OST-object. It is not important on which OST the new empty OST-object will be created. It is the MDT's duty to make the decision according to the system configuration and system space balance case. Generally, it is better to create the new OST-object on the OST that the multiple referenced OST-object resides on.

6.4 Repair unreferenced OST-object

During the second-stage scanning, the orphan OST-objects will be repaired. The MDT will iterate over all of the unreferenced OST-object FIDs and verify that the corresponding parent MDT-object FID does not exist. By this time, the first-stage MDT scanning will have registered the parent FID for any MDT-objects that still exist, so OST-objects without a parent MDT-object FID could be cleaned up immediately or a new FID allocated for them. If the parent MDT-object FID does not exist, then depending on administrator policy for LFSCK the MDT-object will either be recreated in `lost+found` with a default layout, or the OST object will be destroyed.

LFSCK will create a new file (MDT-object) as `lost+found/MDTnnnn/[parent MDT-object FID]` and using the same UID/GID as the orphan OST-object. If the MDT-object FID does exist (or was just created) and there is an empty slot (or a newly-created object) at the OST-object index in the layout of the parent MDT object, the OST-object will be inserted into the layout. The logic will be as follows:

1. If the MDT-object exists, but related layout EA slot is occupied by another OST-object, then check whether it is new created OST-object for fixing dangling reference or for fixing multiple referenced OST-object case.
   a. If yes and nobody modified the new created OST-object, then destroy the new created OST-object on the OST, and update the MDT-object layout EA with the given unreferenced OST-object.
   b. Otherwise, it will be kept under `lost+found/MDTnnnn`. It is the administrator's duty to process it manually.
2. The MDT-object is there, but related stripe information is lost. The LFSCK will update the MDT-object layout EA with the specified stripe information.
   a. If the given stripe offset exceeds current layout EA tail, then needs to extend the layout EA, and put the given stripe information at specified slot. If there are gaps in front of the new stripe slot, then fill them will NULL entries.
   b. If related slot in the layout EA has been filled with a NULL entry, that means the MDT-object layout EA has been extended by the LFSCK in former repairing, then just replace the NULL entry with the given stripe information.
3. Former allocated MDT-object is lost. The LFSCK will generate the layout EA according to the index of the orphan OST-objects. If the layout EA is not completed, means some stripe slots may be empty, then fill as dummy entries. And then create new file with the given MDT-object FID (indicated by the orphan OST-object's file name). Because we only know the MDT-object's FID, but not the file name, it will be created under `lost+found/MDTnnnn/[parent MDT-object FID]`. If multiple orphan OST-objects claim the same MDT-object and the same stripe index, then only one will be merged, the others will be kept under `lost+found/MDTnnnn/`. The administrator can process the others later manually.

To re-generate MDT-object layout, the LFSCK needs to know the stripe-size/stripe-pattern. Currently, only the stripe index is stored in the OST-object. So the LFSCK will use the default values (currently 1MB/ `LOV_PATTERN_RAID0`) to re-generate the MDT-object layout. It may be not correct, but better than do nothing. For the files with replica copies, it is out the scope of LFSCK phase II, and will be handled in the scope of those projects.

6.5 Fix inconsistent layout EA

The layout EA storing on the MDT-object records not only the file layout but also some information which indicates the layout EA owner, such as `lov_mds_md.lmm_oi`. They are generated from MDT-object FID, with them we can know which file the layout EA belongs to. In the LFSCK phase II, we need to verify whether such information in the layout EA is correct or not by re-caculating from the MDT-object FID. If inconsistency is found, trust the MDT-object FID rather than the FID information in the layout EA, which has not been maintained over backup/restore and is mostly used for informational purposes until now.

6.6 Repair inconsistent file owner

The MDT-object owner information is trusted over the OST-object's. Because the chown/chgrp processing order is: client => MDT => OST, it is

quite possible that the OST-object owner information is stale rather than the MDT-object's. Also, the MDT-object's owner information is visible to users and can be directly repaired by the system administrator, while the OST-object's owner information is only used internally by quota. So the LFSCK will update the OST-object owner information according to the MDT-object's owner.

7 Changelog for repairing MDT-OST inconsistency

If the LFSCK repairing changed the normal namespace or file layout, then Lustre file system Changelog should record related changes.

7.1 Changelog for moving file out of the `lost+found/`

For repairing unreferenced OST-object, the LFSCK may create some file under the `.lustre/lost+found/MDTnnnn/`. Although it changes the namespace, the `lost+found/` is a temporary directory for LFSCK repairing. The files under such directory may be changed often, and eventually, the administrator will either move them to other normal directories in the namespace or unlink them (may after some data copy and merge).

Usually, the client-side applications (except for such LFSCK tools) should not use the files under `lost+found/`, because they are in inconsistent status, they should not care the changes under such directory. So it is unnecessary to record those intermediate changes in Changelog, including creating files under the `lost+found/` and removing files from the `lost+found/`.

Moving file from the `lost+found/` to other normal directory is equal to creating new file under the target directory because we do not record former operations under the `lost+found/`. So we will record such moving in Changelog, otherwise, Changelog for further operations against such file will lose the source/target and cause Changelog based applications to be failed, like `lustre_rsync`.

To guarantee old Changelog tools, like `lustre_rsync`, can use new Changelog, the LFSCK will reuse **`CL_CREATE`** for Changelog moving file from the `lost+found/` to other normal directory. On the other hand, to make it distinguishable from other normal create operation, the LFSCK will introduce new `changelog_rec.cr_flags` for `CL_CREATE` type Changelog record: **`CLF_LFSCK_RENAME`**. The `lustre_rsync` will process the **`CL_CREATE`** with **`CLF_LFSCK_RENAME`** as normal **`CL_CREATE`** cases.

```
#define CLF_LFSCK_RENAME 0x0001
```

8 Recovery process

The basic policies are as following:

1. If some servers crashed during the LFSCK, then the LFSCK on other servers should go ahead if they can skip the objects on the crashed servers.
2. The uncommitted repairing should be replayed (via normal replay mechanism) after the crash to guarantee that the fixed objects should be in consistent status.
3. The LFSCK statistics may be not accurate because replay or redo the uncommitted repairing, we do not expect too much on such statistics.

8.1 Recovery from OST failure

The LFSCK master engine on the MDT will not wait for the crashed OST to recover. If an OST id deactivated on the MDS, or is not connected when LFSCK starts, the OST is ignored and the LFSCK status will be marked as partially complete. The recovery process is as follows:

Clear the bit for the crashed OST in the `lfsck_layout.target_bitmap` on the MDT.

- If the LFSCK is in the first-stage system scanning when the crashed OST mounts, the MDT will re-establish the connection to the OST, and all uncommitted RPCs to the OST will be replayed (via normal replay mechanism), including the LFSCK related RPCs for repairing dangling reference, for unmatched referenced MDT-object and OST-object pairs, and for inconsistent owner. The OST lost the OST-object accessing bitmap because of the crash, so the OST will re-join the LFSCK only for the first-stage system scanning, and skip the second-stage system scanning. So the LFSCK status (for both the MDT `lfsck_layout` and the failed OST `lfsck_layout`) will become **`LS_COMPLETED_PARTLY`** finally if there are no other failures or crash. The skipped OST-objects check/repair during the crash and the processing for unreferenced OST-objects will be done by another LFSCK running in the future.

- If the LFSCK is in the second-stage system scanning when the crashed OST mounts, it will not re-join the LFSCK because of lost the OST-object accessing bitmap. The skipped OST-objects check/repair during the crash and the processing for unreferenced OST-objects will be done by another LFSCK running in the future. The LFSCK status (both the MDT `lfsck_layout` and the failed OST `lfsck_layout`) will become **LS_COMPLETED_PARTLY** finally if there are no other failures or crash.

8.2 Recovery from MDT failure

1. If the LFSCK is in the first-stage system scanning when the crashed MDT mounts, then LFSCK master engine on the MDT will be auto restarted, and will resume the system scanning from the position of latest checkpoint. According to the LFSCK checkpoint policy, the checkpoint position will not exceed the position corresponding to the non-committed transactions. So after the crashed MDT mounts-up, the LFSCK master engine will send new RPCs to related OSTs again. Some of the repairing on related OSTs may have been committed during the MDT crash, so these new RPCs may be redundant, and get different results compared with the original RPCs. But it is normal and harmless expect it may misguides the LFSCK statistics. Since no object will be skipped by the LFSCK, and no non-committed repairing will be lost, then the LFSCK status (for both the MDT `lfsck_layout` and the OST `lfsck_layout`) can become **LS_COMPLETED** finally if there are no other failures or crash.
2. Under DNE mode, if there are multiple MDTs run LFSCK in parallel for MDT-OST consistency check/repair, then OSTs will not wait for the failed MDT to recover. If the crashed MDT mounts-up when the others LFSCK are still in the first-stage system scanning, then it can recover its own LFSCK as above. Otherwise, it may missed for some unreferenced OST-objects check/repair during its crash, so even thought it re-join the LFSCK, related bit for the crashed MDT in the `lfsck_layout.target_bitmap` on related OSTs will be cleared, and their status (both the failed MDT `lfsck_layout` and the OST `lfsck_layout`) will become `LS_COMPLETED_PARTLY` finally if there are no other failures or crash. The possible missed unreferenced OST-objects check/repair will be done by another LFSCK running in the future.

9 Wire protocol changes

There will be some wire protocol changes (new RPCs, new flags) for the MDT-OST consistency check/repair.

9.1 Extend `OST_SET_INFO` RPC

MDT uses `OST_SET_INFO` RPC to control the LFSCK on OST with new keys: start, stop, internal synchronization, and so on.

- **`KEY_LFSCK_LAYOUT_START`**
  Start the LFSCK on OST. Corresponding value for the key is the parameters for the LFSCK:

```
struct lfsck_param_val {
 __u32 flags;
 __u32 speed_limit;
};
```

- **`KEY_LFSCK_LAYOUT_STOP`**
  Stop the LFSCK on the OST. Corresponding value for the key is the LFSCK status:

```
__u32 status; /* paused, failed */
```

- **`KEY_LFSCK_LAYOUT_NEXT`**
  Notify the OST that the LFSCK first-stage system scanning is finished. This will return `-EINPROGRESS` if the OST has not yet completed its first-stage scanning. No value for the key.

- **`KEY_LFSCK_LAYOUT_SET`**
  Set parameter for the LFSCK on OST during the LFSCK running. Corresponding value for the key is the same as **`KEY_LFSCK_LAYOUT_START`**.

- **`KEY_LFSCK_LAYOUT_JOIN`**
  Notify the OST that the MDT will re-join the LFSCK after the crash. No value for the key.

9.3 Extend OST_GET_INFO RPC

MDT uses OST_GET_INFO RPC to query the LFSCK progress on the OST with new key(s):

- **KEY_LFSCK_LAYOUT_QUERY**
  Query the OST how the progress of the LFSCK on the OST. Corresponding (reply) value for the key is the LFSCK progress:

```
struct lfsck_progress {
 __u64 lp_objects_processed;
    __u64 lp_objects_total;
    __u32 lp_status; /* running, completed, failed */
};
```

9.5 Extend OST_DESTROY RPC

MDT uses the OST_DESTROY RPC to check whether the specified OST-object has been modified or not, it not, then destroy it. This RPC will be called when repairs the unreferenced OST-object to check and destroy the former created empty OST-object for dangling reference or multiple referenced cases. We will introduce new obd_flag to indicate the special destroy RPC.

- **OBD_FL_LFSCK**

```
obdo.obd_valid |= OBD_MD_FLFLAGS
obdo.obd_flag |= OBD_FL_LFSCK
enum obdo_flags {
 …
 OBD_FL_NOSPC_BLK    = 0x00100000,
 OBD_FL_LFSCK = 0x00200000, /* special for LFSCK. */
};
```

9.8 Extend OST_SETATTR RPC

The MDT needs to be able to repair the OST-object parent FID and UID/GID in case an inconsistency is found. Ideally, we will use OUT RPC for such purpose which allows batching multiple different object updates into a single RPC, but current OUT RPC only works for MDT operations, and related work for OUT on OST is not started yet (LU-3467 for Unified Targets/Data on MDT project). To decrease the dependency, OST_SETATTR RPC may be reused as a temporary solution before OUT on OST is available. The existing OST_SETATTR RPC can be used to batch setattr("uid", "gid") + setxattr("fid") by the MDT to repair the inconsistent OST-object, including unmatched referenced MDT-object and OST-object pairs, and inconsistent owner. When OUT on OST is ready, the LFSCK will batch normal setattr and setxattr in single OUT RPC to OST for the repair.

If the OUT is not available on the OST, the MDT may also reuses the OST_SETATTR or OST_CREATE RPC to materialize a new OST-object by the LFSCK with specified parent MDT-object FID and owner attribute, and set the “SUID + SGID” mode to make it distinguishable from other normal created OST-objects. This RPC will be called when repairs multiple referenced OST-object. We can reuse the new obd_flag OBD_FL_LFSCK (introduced by OST_DESTROY above) for the RPC to make it easy to be distinguished.

```
obdo.obd_valid |= OBD_MD_FLFLAGS
obdo.obd_flag |= OBD_FL_LFSCK
```

10 API changes

Wherever possible existing APIs will be employed for the LFSCK processing. In the LOD/OSP stack, some of the OSD APIs may have been defined but not implemented for some modules, because nobody uses them before, such as the interfaces for obd_ops.{o_get_info,o_set_

`info_async}()`. We need to implement them for the LFSCK, which will not affect the existing Lustre software code, and is the best choice.

Next, we will extend some existing APIs for enhancing some API functionality. For example, current `dt_object_operations.do_destroy()` will destroy the specified object, and all its sub-objects will be destroyed also. But sometimes, we just want to destroy some specified sub-object, then we need to call this API directly on the lower-level OSP object, or pass something into the LOD `dt_object_operations.do_destroy()` as indication. These extensions may affect the existing Lustre software code to some extent.

10.1 Implement obd_ops.o_set_info_async() for LOD/OSP

The LFSCK master engine on the MDT uses the `o_set_info_async()` to control the LFSCK slave engine on the OST.

```
int (*o_set_info_async)(const struct lu_env *, struct obd_export *,
                        __u32 keylen, void *key,
                        __u32 vallen, void *val,
                        struct ptlrpc_request_set *set);
```

- **`lod_set_info_async()`**
  Transmit the request to the specified OSP(s). The OSP(s) will be specified thought the parameter `void *val` as an index array.

  ```
  struct info_val {
   void *iv_real_val;
   int iv_array_length;
   struct iv_array {
    int iva_index;
    int iva_result;
   }[N];
  };
  ```

  The results will be saved in the `info_val.iv_array.iva_result`, then the LFSCK main engine can know which OSPs/OSTs are in the LFSCK processing. Introduce a new flags `lod_ost_desc.`**`ltd_lfsck`**. If the target OSP/OST responds **`KEY_LFSCK_LAYOUT_START`** or **`KEY_LFSCK_LAYOUT_JOIN`** successfully, then set such flag to indicate that this OSP/OST is part of the current LFSCK. And subsequent LFSCK related processing will check this flag, only process the OSP/OST with the `lod_ost_desc.`**`ltd_lfsck`** set.

- `osp_set_info_async()`
  Send `OST_SET_INFO` RPC to the specified OST.

10.2 Implement obd_ops.o_get_info() for LOD/OSP

The LFSCK master engine on the MDT uses the `o_get_info()` to sync status with the LFSCK slave engine on the OST.

```
int (*o_get_info)(const struct lu_env *env, struct obd_export *,
                  __u32 keylen, void *key, __u32 *vallen, void *val,
                  struct lov_stripe_md *lsm);
```

- **`lod_get_info()`**
  Transmit the request to the specified OSP(s). The OSP(s) will be specified thought the parameter `void *val` as an index array. See the **`info_val`** for `lod_set_info()`. The results will be saved in the info_val.array.result, then the LFSCK main engine can know each LFSCK slave engine status.

- **`osp_get_info()`**
  Send `OST_GET_INFO` RPC to the specified OST.

10.3 Reuse dt_object_operations.do_destroy for conditional destroying

specified OST-object

It is necessary to support destroy OST-object selectively for the LFSCK. The LFSCK will bypass LOD and talk with OSP directly for specified OST-object check and destroy. For conditional destroy case, the `osp_object_destroy()` will not generate llog. Instead, it will send `OST_DESTROY` RPC (with OBD_FL_LFSCK) directly (indicated by th_sync) to the OST (without involving osp_sync_thread), and tell the caller the result.

10.4 New `dt_device_operations::dt_child` for accessing child device with the given type and index

To support the LFSCK to talk with OSP directly (bypass LOD), we need a method to obtain the OSP device via LOD device. Such method is not only used for LAYOUT consistency check/repair, but also for DNE consistency in LFSCK Phase III.

```
struct dt_device *(dt_child) (struct dt_device *parent, int type, __u32 index);
```

@**`parent`**: usually, it is the dt_device for LOD

@**type**: for OST/MDT, or for others

@**`index`**: index for the target device

11 Race control between MDT-OST consistency check/repair and other operations

As an online system consistency maintaining tool, the LFSCK needs to control the concurrent object accessing during the LFSCK, such as setattr/unlink the MDT-object/OST-object during the LFSCK.

11.1 No lock on the MDT-object when related LFSCK RPC is handled on the OST

The LFSCK main engine on the MDT does not hold any lock (neither ldlm ibits lock, nor osd_{read,write}_lock) on the MDT-object when its LFSCK RPC is in-handling on the OST for related OST-object check/repair. This will avoid deadlock with other RPC(s). When the LFSCK RPC is replied, if the target OST-object is repaired, then the LFSCK needs to check whether the MDT-object is changed (unlink or setattr) or not during the LFSCK as following:

- If the MDT-object is dangling reference case and has been unlinked during the LFSCK, then the MDT will send destroy RPC to destroy the possible miss created OST-object. Consider the following sequence:
    1. On the MDT, the LFSCK scans the MDT-objectA, which is normal file, not unlinked yet, then the MDT sends RPC to OST for OST-objectA verification.
    2. Another thread unlinked the MDT-objectA, and trigger destroy RPC to OST for destroying OST-objectA.
    3. On the OST, the destroy RPC is processed earlier than the LFSCK RPC. So the OST-objectA has been destroyed before the LFSCK process it. So the LFSCK on the OST thinks that the MDT-objectA has dangling reference, and then will re-created the "missed" OST-objectA. But it fact, it should NOT do that.
    4. On the MDT, when the LFSCK main engine gets the RPC reply, it finds that the MDT-objectA has been unlinked during the LFSCK, so the MDT will send destroy RPC to the OST to destroy the new created OST-objectA.

  Generally, it is rare case that the destroyed OST-object will be re-created by LFSCK unexpectedly. It will NOT cause much overhead. The bad case is that: if the MDT crashed between step 3 and 4, or failed to rollback, then the new created empty OST-objectA will become unreferenced OST-object. Those empty unreferenced OST-objects will be dropped by the LFSCK when next run.

- If the OST-object owner is repaired by the LFSCK and the MDT-object owner has been updated during the LFSCK, then the owner for MDT-object and OST-object may be inconsistent. So the LFSCK needs to trigger the `OST_SETATTR` RPC according to the MDT-object current owner attributes. If the MDT crashed, then when the LFSCK resumes from the last checkpoint, such inconsistency will be repaired; if the LFSCK failed to rollback, then such inconsistency will be kept there until next LFSCK run.

11.2 Race between the unlink/destroy operations and the second-stage LFSCK

scanning

There are race conditions between the unlink/destroy operation and the LFSCK repairing unreferenced OST-object as following sequence:

1. On the MDT, the RPC service thread unlinked the striped fileA which has not been scanned by the first-stage LFSCK scanning.
2. The LFSCK master engine drives the first-stage system scanning on the MDT, but it will not find the MDT-objectA for fileA, because it is already unlinked.
3. The LFSCK master engine notifies the LFSCK slave engine to start the second-stage system scanning on the OST. At that time, the RPC service thread has not sent related RPC to the OST for destroying the fileA's OST-objectA yet.
4. The LFSCK slave engine on the OST drives the second-stage system scanning, and finds that nobody has accessed the OST-objectA (corresponding to the fileA) during the first-stage system scanning. So OST-objectA will be regarded as unreferenced OST-object and to be repaired by attaching it to the `lost+found/` directory.
5. Another RPC service thread on the MDT cannot find the MDT-objectA before repairing the unreferenced OST-objectA, so it re-created the MDT-objectA under `lost+found/`.
6. The OST received the RPC for destroying OST-objectA from the MDT (as part of unlink operation), and destroyed OST-objectA.

LFSCK will avoid this problem by calling `dt_sync()` (depends on LU-3469) at the end of the first stage scanning, which commits the fileA unlink to the local OSD firstly (this schedules the OST objects for unlink), then sends all of the pending OST object unlinks afterward. The OST will receive the destroy RPCs for each object and clear the corresponding bit from the orphan bitmap. Only after `dt_sync()` completes does LFSCK second stage scanning begin, processing only objects that definitely do not exist in the MDT namespace.

11.3 Handle empty unreferenced OST-objects

1. If the empty unreferenced OST-object is created after the LFSCK start, then it can be skipped by the LFSCK directly.
2. If the unreferenced OST-object is materialized or other OST-object in the same sequence is materialized, then we can let the LFSCK on the MDT to make a further distinction; otherwise keep it there.
3. On the MDT side, the LFSCK engine will make a further distinction: if the FID already allocated and nobody reference it, then destroy it; otherwise keep it there.
4. To avoid confusion between objects that are unreferenced by a particular MDT, but may be referenced by a second MDT in the DNE case, second-stage orphan object handling will not be done until all MDTs have completed their first-stage processing.

12 Handle layout changes

For repairing unreferenced OST-object, the MDT LFSCK engine will re-create the miss MDT-object or update/extend the existing MDT-object layout EA. Before change the MDT-object layout EA, the LFSCK thread on the MDT needs to acquire the layout lock with exclusive mode firstly to prevent the client to access OST-objects with stale layout information. Usually, it is the MDT layer to handle LDLM lock on the MDT, including the layout lock. LFSCK for repairing the unreferenced OST-object also follows such policy. On the other hand, repairing multiple referenced OST-object also needs to change MDT-object layout EA. Such layout EA changes also need to be protected by layout lock.

Currently for layout_swap case the parent information stored on the children (OST-objects) will not been updated, then after the layout_swap there may be some unmatched referenced MDT-object/OST-object pairs until the layout_swap repairs the parent FID. This could be implemented only after the similar LFSCK mechanism for parent FID repair is available. On the other hand, the LFSCK needs to check/repair those inconsistent files. Further more, the layout swap may happen during the LFSCK check/repair, which makes the situations more complex. When the OST is able to handle OUT updates for LFSCK, layout swap should be changed to update the parent FID on the objects directly.

12.1 NOT hold the `MDS_INODELOCK_LAYOUT` lock during the LFSCK RPC processing on the OST

If we do NOT hold this lock, then someone may changed the MDT-object layout EA when the LFSCK RPC handled on the OST. Then when the LFSCK RPC replied, how can the LFSCK main engine process? If the LFSCK repair nothing on the OST, then does nothing; otherwise, the LFSCK on the OST may updated the OST-object's parent information, but now the updated parent information may be wrong. Does the LFSCK needs to rollback (destroy the updated OST-object)? the answer is NO. Because after the layout changing, those updated OST-object(s) by the LFSCK become old (stale), the sponsor of the layout changing will destroy those old OST-object(s). Even though the LFSCK may re-create the OST-object(s) just after the destroy because of rare conditions, those empty unreferenced OST-objects cannot be found via namespace, and will be dropped by the LFSCK when next run.

13 Interoperability and Compatibility

Most of the changes related with the LFSCK for MDT-OST consistency check/repair only affect servers. And there are no wire protocol changes related with client side RPCs. The layout lock part is not new feature introduced by the LFSCK project, the interoperability issues caused by layout lock will not be discussed here. Generally, there will not be interoperability issues between old client and new server for the LFSCK Phase II processing. As for server side, several wire protocol changes will be made, and old server will not recognise those new RPCs or new parameters, as to fail related RPCs. So the LFSCK cannot work under the environment with old MDT/OST running.

For 1.8 format device, the MDT-object is identified with IGIF that will be changed in case of file-level backup/restore. That means the parent information previously stored in the OST-object will be invalid under such case. So for the system upgrading from 1.8 backup, the parameter **`ofd.$fsname-OSTnnnn.fail_on_inconsistency`** should be off until LFSCK has been able to repair the OST-object's parent FID.

---

*Other names and brands may be the property of others.