LFSCK 1.5 High Level Design

1 Introduction

This document describes the agreed design decisions that will be implemented to achieve the goals of the Lustre File System ChecK (LFSCK)1.5 FID-in-dirent and linkEA consistency checking/repairing. The scope of the project is presented in the LFSCK 1.5 Scope Statement. The LFSCK 1.5 Solution Architecture describes the important components of a successful implementation. A familiarity with the Solution Architecture is necessary before embarking on this document.

2 Scanning the MDT OSD device

LFSCK 1.5 FID-in-dirent and linkEA consistency checking will combine the object table-based iteration and incremental namespace-based directory traversal to scan the whole MDT device to discover inconsistent FID-in-dirent and linkEA entries.

LFSCK is driven by the object table-based iterator concurrently with OI Scrub. The OSD layer object table-based iteration will scan and return each object in sequential order to the MDD layer index iterator.

If the object returned by the object table-based iteration is a directory, then LFSCK will traverse entries in the directory in index order. Directories renamed during directory traversal will not cause subtrees to be missed as would be the case during pure namespace traversal, because the object table-based iteration will guarantee that all the directory objects are processed.

Before LFSCK begins namespace-based directory traversal, directory object client visibility is verified. FID-in-dirent and linkEA are needed only on the client-visible objects.

2.1 Upgrade Lustre 1.8 objects

For filesystems formatted with Lustre 1.8, objects are identified with Inode and Generation in FID (IGIF FID) identifiers from the underlying filesystem. IGIF FID is mapped into a reserved subset of the FID namespace. This is inconsistent with Lustre 2.x filesystem FIDs which are abstracted from the underlying filesystem for portability. The first step for upgrading such filesystems is to add the IGIF FID into the LMA xattr on each inode and add the IGIF FID into the Object Index (OI) files. This task is performed during normal OI Scrub processing that will ensure that both normal FID objects and IGIF FID objects are handled uniformly.

If an upgraded 1.8 filesystem that has had LFSCK 1.5 store the IGIF FID in the LMA xattr and OI is downgraded, there will not be any problems as a result. Since the IGIF FID is a 1:1 mapping to the underlying inode and generation numbers used by the 1.8 MDS there will not be duplicate identifiers for any object.

2.2 Filter out objects not visible to the client

For a 2.x formatted device it is simple to distinguish if the given object is visible to client by checking linkEA. linkEA are only generated for client visible objects. The Lustre “`ROOT`” object is an exception. The "`ROOT`" is visible to client but does not have a linkEA. In this case the Lustre “`ROOT`” object can be identified as it is pinned in RAM.

Complexity increases when considering the case of an MDT upgraded from Lustre 1.8. Lustre 1.8 objects do not have the linkEA regardless of whether it is visible to client or not. The OSD cannot itself distinguish whether an object is visible to the client. As a consequence, the object table-based iteration will scan the entire device and return all the objects in flat order irrespective of client visibility. The LFSCK will filter out the internal objects at the MDD layer. The logic for filter out client invisible objects is as follows:

1. For the object returned by object table-based iteration, the LFSCK 1.5 needs only to process directory

objects, because the non-directory object has the same visibility as its parent directory.

2. If the directory object returned by the object table-based iteration has a linkEA, or the object is the Lustre “`ROOT`”, then it is visible to client and the same for all child objects.

3. If the directory returned by the object table-based iteration does not have a linkEA and is not the Lustre “`ROOT`”, then it is either hidden from the client (i.e. Lustre internal metadata), or the object is of 1.8-format. It is also possible that the object is a 2.x object that has a corrupted linkEA that was removed by `e2fsck`. For all these cases, the LFSCK will check the parent visibility by `lookup("..")`. If the parent is also missing linkEA and is not the Lustre "`ROOT`", then the LFSCK will repeat `lookup("..")` on each parent directory until an ancestor directory that has linkEA or the Lustre “`ROOT`” is discovered. If this is the case, the object is visible to client; otherwise the OSD root directory will be reached and the object is a hidden internal object.

For most of 2.x objects, the step 1) and 2) are enough. The step 3) is mainly for 1.8-IGIF objects being upgraded. Since upgrading is a 'once-only' operation that can terminate as soon as the any parent is found with a linkEA xattr, the overhead introduced by `lookup("..")` recursively for object visibility is judged to be acceptable.

2.3 Find missing or inconsistent FID-in-dirent

For the client visible directory object returned by the OSD object table-based iteration LFSCK will traverse the directory in namespace-based order. For each child object the processing in the OSD layer is as following:

1. If the child object has neither FID-in-LMA nor FID-in-Dirent, it is a 1.8-format IGIF object. Insert the IGIF FID into the dirent to repair it.
2. Else if the child has no FID-in-dirent then it is restored from file-level backup or half-processed 1.8-format IGIF object. Insert the LMA FID into the dirent to repair it.
3. Else if the FID-in-dirent does not match the FID-in-LMA, then it is an inconsistent FID-in-dirent. If LMA exists, store it into FID-in-dirent. If LMA is missing (it was lost or removed by `e2fsck`,) store FID-in-dirent into LMA.

2.4 Find missing or inconsistent linkEA

For the client visible directory object returned by the low layer object table-based iteration, LFSCK will traverse the directory with namespace-based order. For each name entry, LFSCK will verify whether there is valid linkEA entry on the target object corresponding to the "parent FID + child name". In the design document, the character 'L' is used to identify object linkEA entries count, 'D' is the count of the directory entries pointing to the target object, 'N' is for the object nlinks count. For most, but not all cases, "L == D == N". There is an important condition for LFSCK 1.5: whether we should trust that "D == N" or not, it will much affect LFSCK 1.5 complexity.

1. The case of "D == N" is trusted.
   It is relative easy. For example, the object_A, assume N = 1, when the LFSCK finds some name entry name_A pointing to the object_A under the parent FID_PA, then we can exactly know that the unique valid linkEA entry for the object_A is the "name_A + FID_PA", any other linkEA entries on the object_A are invalid or redundant, and should be removed. So the single-linked object_A can be completely processed when the first name entry is found during the first cycle LFSCK scanning.

2. The case of "D == N" is not trusted.
   Things become more complex. The same example as above, when the LFSCK finds some name entry name_A pointing to the object_A under the parent FID_PA, we only can know that the "name_A + FID_PA" is a valid linkEA entry for the object_A, but we do not know whether it is the unique valid linkEA entry for the object_A. Because 'D' may be larger than 'N', and before the LFSCK finish to scan the whole device, nobody knows the 'D'. Although we know the 'L' at the beginning, but 'L' may increases if found some other directory entries pointing to the object_A but related linkEA entries on the object_A do not exist originally.

On the other hand, for the case of "L >= 1" at the beginning of the LFSCK, to verify whether the object_A contains some invalid linkEA entries or not, the LFSCK needs to record the verification history for each linkEA entry on the object_A in RAM (preferred). The LFSCK will keep such in-RAM records until all the known linkEA entries have been verified or the whole device scanning finished. Further more, the MDT may crash during the LFSCK and lose in-RAM linkEA entries verification history, so the LFSCK needs to store the object_A in some on-disk file (called "`lfsck_namespace`", will be described in more detail later in this document), then if the MDT crashed during the LFSCK, it can re-scan the objects stored in the "`lfsck_namespace`".

Originally, we trended to use option 1). But it is depends on the offline e2fsck tool to guarantee the "D == N". As an online system consistency tool, it is impossible to run e2fsck offline before each LFSCK run. So over the lifetime of an active Lustre system, 'D' may become different from 'N', because of some partial operations (such as link/unlink/rename) or some system failures/errors. So now, we will design and implement the LFSCK 1.5 based on the assumption 2): 'D' may be different from 'N'.

Make the linkEA entries consistent with the directory entries is one of the LFSCK 1.5 goal, so the LFSCK will trust 'D' if there is an inconsistency. But there is an exception: we will ignore the case of 'D == 0', that means if we cannot find any name entry pointing to the object_A during the namespace-based directory traversal, even though the object_A may has non-zero 'L' and 'N', we will not repair it. It is the e2fsck's duty to fix such inconsistencies. Since client cannot find the object_A through namespace, such inconsistency is harmless for the client until it is repaired by e2fsck.

2.4.1 Store objects in the "`lfsck_namespace`" for double scanning

Assume that the number of verified linkEA entries on the object_A is 'V', it is an in-RAM variable, and will be reset to 0 if LFSCK is restarted in the case of resume from a MDT crash. For a detailed description of `lfsck_namespace` see "section 3 LFSCK tracing" below. LFSCK needs to store the object in the "`lfsck_namespace`" only when "(V == 1) && (N > 1 || L > 1)". Consider the following cases:

1. L > 1 || N > 1
   Typically, this is for multiple-linked object. If the object_A contains more than one linkEA entries at the beginning of LFSCK, then it will be stored in the "`lfsck_namespace`" when the first name entry pointing to the object_A is found.

2. L == 1 && N == 1
   Typically, this is for singly-linked object. If LFSCK finds the directory entry pointing to the object_A that matches the unique linkEA entry, then processing is complete. Otherwise if a name entry pointing to the object_A does not match the unique linkEA entry, then a new linkEA entry will be added, and 'L' will increase ('N' will not increase, become the case 1). object_A would then be stored in the "`lfsck_namespace`" file.

3. L == 0
   It is usually for IGIF object. When new linkEA entries are added, it becomes the case 2 or the case 1.

As LFSCK executes, object_A can be removed from the "`lfsck_namespace`" when "L == V == N > 1" is detected to avoid unnecessary double scanning.

2.4.2 Record linkEA verification history in RAM

To know which linkEA entries on the object_A have been verified, LFSCK must pin object_A in RAM and record the linkEA entries verification history. To avoid exhausting available memory, not all objects are pinned in RAM. LFSCK permanently pins the object in RAM only when "(V == 1) && (N > 1 || L > 1)". Consider the following cases:

1. L > 1 || N > 1
   LFSCK treats the linkEA entry as unverified as the in-RAM verification history is absent. It is necessary to pin

the object in RAM for this case until all of the linkEA entries are verified.

2. L == 1 && N == 1
   Typically, this is for singly-linked object. If LFSCK finds the directory entry pointing to the object_A that matches the unique linkEA entry, then processing is complete. Otherwise if a name entry pointing to the object_A does not match the unique linkEA entry, then a new linkEA entry will be added, and 'L' will increase ('N' will not increase, become the case 1). object_A and its linkEA verification history will be pinned in RAM.

   It is possible that, the first found name entry matches the unique linkEA entry, then L == V == N == 1, we neither record the object in the "lfsck_namespace" or pin the object in RAM, but as the LFSCK scanning, more name entries pointing to the same object may be found, at that time, with those new linkEA entries added, the object will be pinned in RAM and recorded in the "lfsck_namespace" file, and will be double scanned later. For a large system, this kind "upgrading" is very rare. We prefer to double scan these objects instead of pinning most unnecessary objects in RAM.

3. L == 0
   It is usually for IGIF object. When new linkEA entries are added, it becomes the case 2 or the case 1.

If too many objects are pinned in RAM, it may cause server memory pressure. To avoid exhausting memory, LFSCK needs to unpin objects from RAM. The following conditions to un-pinning are applied:

1. L == V
   All the known linkEA entries on the object are valid. Although there may be other directory entries pointing to the object will be found as the LFSCK scanning. It is unnecessary to maintain the linkEA entries verification history, instead, add some on-disk flag (**VERIFIED**) on the object in the "lfsck_namespace" file. If more directory entries pointing to the object are found, the LFSCK can detect this flag and just adding new linkEA entries without maintaining the verification history.

2. Memory pressure
   All the objects with "L == V" have been unpinned from RAM but there still is memory pressure. LFSCK will unpin some half-verified objects from RAM. Since these objects have been stored in the “`lfsck_namespace`” when they pinned in RAM, the possible invalid linkEA entries on these unpinned half-processed objects can be handled during the double scan.

2.4.3 Post-processing after the LFSCK first cycle device scanning

After the first cycle object table-based iteration, the LFSCK will try to remove the linkEA entries that reference non-existent link names, as following:

1. For the objects still pinned in RAM, if the LFSCK has ever resumed from some checkpoint (for MDT crash or sysadmin pause/stop), then check the non-verified linkEA entries one-by-one; otherwise remove those non-verified linkEA entries directly.

2. Rescan the objects stored in the “`lfsck_namespace`” to check if some linkEA entries reference non-existent link names.

3. L == N > D
   The LFSCK will add the missing name entry back to the namespace.

4. L > N > D || N > L > D
   'L' is trusted, means adjust 'N' to match 'L', then become the case 3.

5. N > D >= L
   'D' is trusted, means adjust 'N' to match 'D'.

When an object is post-processed, then remove it from the "`lfsck_namespace`" to avoid repeatedly processing it if LFSCK restarts (due to MDT crash or sysadmin pause/stop) during the post-processing.

3 LFSCK tracing

The LFSCK will introduce a new local file named "`lfsck_namespace`" on the MDT to track the processing for FID-in-dirent and linkEA consistency check/repair. The LFSCK parameters including status, statistics, checkpoint, and the FIDs for multiple-linked objects will be recorded in the "`lfsck_namespace`". This file can be used for querying from user space and for providing the LFSCK resume functionality.

Updating "`lfsck_namespace`" for each object processed will significantly impact performance. Instead, the update will be cached in RAM and written to disk periodically. Writes to disk will be processed asynchronously through the server journal. The default write interval is 60 seconds. Write to the file "`lfsck_namespace`" will create a new checkpoint. If the system crashes before the write, at most 60 seconds (plus journal commit interval) work will be lost. When the system recovered, the LFSCK can restart (resume) from the position of the last checkpoint in the "`lfsck_namespace`" file.

- **lfsck_namespace**

  This is the "`lfsck_namespace`" file on-disk structure:

```
struct lfsck_namespace {
/** Magic number to detect that this struct contains valid data. */
__u64 lfm_magic;
/** Parameters controlling LFSCK runtime behaviour. */
__u64 lfm_param;
__u64 lfm_flags;
/** Current LFSCK status from enum lfsck_status */
__u64 lfm_status;
/** Time for the last LFSCK completed in seconds since epoch. */
__u64 lfm_time_last_complete;
/** Time for the latest LFSCK ran in seconds since epoch. */
__u64 lfm_time_latest_start;
/** Time for the last LFSCK checkpoint in seconds since epoch. */
__u64 lfm_time_last_checkpoint;
/** Position for the latest LFSCK started from. */
struct lfsck_position lfm_pos_latest_start;
/** Position for the last LFSCK checkpoint. */
struct lfsck_position lfm_pos_last_checkpoint;
/** Position for the first should be updated object. */
struct lfsck_position lfm_pos_first_inconsistent;
/** How long the LFSCK has run in seconds. */
__u32 lfm_run_time;
/** How many completed LFSCK runs on the system. */
__u32 lfm_success_count;
/** How many objects (including dir) have been checked. */
__u64 lfm_objs_checked;
/** How many objects have been repaired. */
__u64 lfm_objs_repaired;
/** How many objects failed to be processed. */
__u64 lfm_objs_failed;
/** How many directories have been traversed. */
__u64 lfm_dirs_checked;
/** How many multiple-linked objects have been checked. */
__u64 lfm_nlink_checked;
__u64 padding[M];
};
```

The position for the LFSCK 1.5 is composed of two parts as following:

```
struct lfsck_positionition {
/** local layer object table-based iteration position. */
__u64 lp_oit_cookie;

/** parent FID for directory traversal. */
__u64 lp_dir_parent;

/** namespace-based directory traversal position. */
__u64 lp_dir_cookie;
};
```

This file description is to support LFSCK for FID-in-dirent and linkEA consistency check/repair. There may be other tracing files for other LFSCK components in the future. It is undesirable for not want multiple LFSCK components shares the same trace file because it will require unnecessary complexity to the tracing file structure and potentially introduce compatibility issues.

- **lfsck_namespace::status**

  For LFSCK current status, as following:

```
enum lfsck_status {
/** Initial state, no LFSCK has been run on this filesystem */
 LS_INIT,
 /** first-step system scanning. */
 LS_FIRST-SCANNING,
 /** second-step system scanning for multiple-linked objects. */
LS_SECOND-SCANNING,
/** LFSCK processing has completed for all objects */
 LS_COMPLETED,
/** 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 paused automatically when umount, can be restarted automatically
when remount. */
 LS_PAUSED,
/* System crashed during the LFSCK, can be restarted automatically after recovery. */
 LS_CRASHED,
};
```

  Each time the MDT is mounted it will check if a LFSCK restart is required. If in the status is `LS_PAUSED` or `LS_CRASHED`, then LFSCK will be restarted from the breakpoint automatically.

- lfsck_nlink_record

  As described in former sections, the "lfsck_namespace" file is also used for storing the multiple-linked objects for double scanning. To simplify the operations (insert/delete/lookup/iteration) on the file, it will be implemented as an index file. Above the on-disk "struct lfsck_namespace" will be stored as a new extended attribute (named **XATTR_NAME_LFSCK**) in the "lfsck_namespace" file. And for each multiple-linked object to be double scanned, it will stored as an fixed-length record as following:

```
struct lfsck_nlink_record {
/** KEY: the object's FID. */
struct lu_fid FID;
/** REC: some flags, such as "VERIFIED". */
__u8 flags;
};
```

  The operations against the **lfsck_nlink_record** will use the existing fixed-length IAM APIs (osd_index_iam_ops) in the OSD layer.

4 LFSCK user space control

Where prudent, the existing LFSCK user space tools will be reused to control the LFSCK for FID-in-dirent and linkEA consistency check/repair. A new LFSCK type “`namespace`” will be introduced.

```
lctl lfsck_start -M lustre-MDT0000 -t namespace
```

Kernel space logic will be implemented to start/stop the LFSCK for FID-in-dirent and linkEA consistency check/repair.

4.1 Query LFSCK processing

New, special tools for querying the LFSCK processing will not be created. Instead, querying will be performed via new lproc interface:

```
lctl get_param mdd.${fsname}-MDT${idx}.lfsck_namespace
```

This is a MDT-side lproc interface in YAML format for querying FID-in-dirent and linkEA consistency check/repair processing. A key feature of this interface is to provide access to the “lfsck_namespace” file on the MDT.

4.2 Speed control

LFSCK phase I implemented a basic LFSCK speed control mechanism. The administrator can specify the max speed for the LFSCK to scan the device through the MDD layer lproc interface:

```
        lctl set_param mdd.${fsname}-MDT${idx}.lfsck_speed_limit=N
```

Where N is the maximum speed in inodes per second. We prefer to reuse such mechanism in LFSCK phase 1.5 for controlling the LFSCK speed.

5 Repair the inconsistency

Since the FID-in-dirent and linkEA belong to different code layers of the metadata stack they will be repaired separately. As a result, the FID-in-dirent and linkEA will be checked and repaired in different layers by separate mechanisms. This will not result in repeat scanning or duplicate processing.

5.1 Initial OI scrub to verify server local objects

When the server mounts, it needs to access to some server local objects. These server local objects can be accessed by name or by FID. This requires the FID mappings in the OI files for these server local objects to be correct, even though the server is restored from server-side file-level backup. As a result, an initial OI scrub is required to make a local scope OI check/repair synchronously when the server mounts. Compared with the normal whole device OI scrub, the initial OI scrub has the following different behaviour or requirements:

1. The scan is not object table-based iteration. A backend local root namespace-based directory traversal is performed.
2. The scan scope is not the whole device. Only server local objects are recursively scanned, except for the subdirs:
    a. /lost+found. The sysadmin should guarantee all the useful objects have been moved out of the lost+found.
    b. /O. "LAST_ID" under the "/O" directory/sub-directly will be checked/repaired.
    c. /PENDING. The "/PENDING" directory itself will be checked/repaired.
    d. /ROOT. The "/ROOT" directory itself and its child "/ROOT/.lustre" will be checked/repaired.
3. The initial OI scrub is triggered by the mount thread directly inside the OSD at the beginning of the server mount. This guarantees the server local object FID mappings can be verified before others using them.
4. The initial OI scrub is triggered automatically when the server mounts under the conditions:
    a. The server is upgraded from old 2.x or 1.8 device.
    b. The server is restored from server-side file-level backup.
    c. OI files crash caused by known/unknown reasons.

Timing considerations: A typical server has hundreds of local objects. The performance results from LFSCK 1 a full speed OI scrub can process 50K objects per second. Hence, an initial OI scrub would expect to complete within 1 second. However, the initial OI scrub is not triggered frequently, so even though it takes a longer time it will not affect normal system availability.

If the target device is 2.4 or newer format, processing is simple. In Lustre-2.4, when the local object is created, the FID_SEQ_LOCAL_FILE FID is stored in the object LMA. With this information the related FID mapping can be directly checked and repaired in the OI files. For 2.x or 1.8 formatted device, no FID was stored in the object LMA when the local object was created. When the server upgrades from old 2.x or 1.8 device the initial OI scrub will generate IGIF mode FID for those old local objects, store the IGIF FID in local object LMA, and build related IGIF FID mappings.

5.2 Handle IGIF objects

If a MDT is upgraded from Lustre 1.8 then the object has IGIF mode FID: it has no FID in LMA, no OI mapping in the OI file, and neither the FID-in-dirent nor linkEA is available. This case needs additional logic on the MDT to process these IGIF objects. To unify and simplify the normal LFSCK and RPC processing, Lustre 1.8 IGIF objects are treated as normal 2.x FIDs. The following tasks must be completed to the inodes:

1. Add the IGIF FID in the object LMA.
   With the FID-in-LMA, the OI scrub can rebuild its OI mapping when the MDT restored from file-level backup.

2. Add the IGIF FID mapping in the OI file.
   The mapping for “IGIF <=> ino/gen” that allows the IGIF FID can be reserved as normal FID even after a MDT file-level backup/restore. The OI scrub can rebuild the OI files when needed to guarantee lookup-by-FID still completes after the MDT file-level backup/restore against IGIF files.

3. Add IGIF FID to the name entry in the parent directory.
   `readdir(3)` on the MDT can resolve all the needed information from parent directory directly without reading each child object.

4. Add linkEA with the child name and the parent FID.
   "`lfs fid2path`" will work against IGIF objects.

It is the OI scrub's responsibility to add IGIF FID in the object LMA and insert related IGIF FID mapping into the OI file. This is the case for normal 2.x FID. In Lustre-2.4, both client visible objects and server local objects have FID-in-LMA. This means the OSD will not distinguish between client visible or local objects. The OI scrub can process 1.8-IGIF objects during an upgrade. If the object has no FID-in-LMA, then an IGIF is generated with the inode ino/gen. The IGIF FID is added into the object LMA, and the related IGIF FID mapping is inserted into the OI file.

OI Scrub scans the device in a flat order. This may be somewhat different from the LFSCK piecewise directory traversal order. It is possible to the system crashes immediately after LFSCK adds an IGIF FID to the name entry the parent directory and before the OI scrub adding IGIF FID to the inode LMA has completed. When the system recovers from this situation, the FID-in-dirent exists but FID-in-LMA missed. This situation is resolved as follows:

1. In the normal case where there is no MDT file-level backup/restore after the crash, the IGIF FID in the object name entry is valid in regardless of weather the IGIF FID is stored (by the OI scrub) in the inode LMA or not because the IGIF FID is just composed of the inode number and generation, and can be regenerated as needed.

2. In the highly unlikely case there is an MDT file-level backup/restore after the crash where the FID-in-dirent cannot be backed-up, the inconsistency will be automatically resolved by LFSCK after the MDT is restored from backup.

For this reason, it is unnecessary to keep the order between the LFSCK adding IGIF FID to the parent directory name entry and the OI scrub adding IGIF FID in the inode LMA. The OI scrub logic will include a minor modification for 1.8-IGIF objects upgrading as following:

```
for_each_inode {
 fetch LMA;
 if (no FID-in-LMA) {
 fid = IGIF;
 osd_ea_fid_set(inode, fid);
 }
 verify/update OI mapping with FID-in-LMA;
 }
```

With above processing by the OI Scrub, IGIF objects can be treated as the normal FID case. This has important impact on the higher LFSCK layers that no longer need to distinguish 1.8-IGIF objects. For the MDT upgraded from old 2.x-format device, the local objects have no FID-in-LMA and the OI Scrub will treat them as 1.8-IGIF objects: generate IGIF FID with the inode ino/gen, add the IGIF FID in the object LMA, and insert related IGIF FID mapping in the OI file. This method is robust even the system is downgraded back to the old 2.x format.

5.3 Repair inconsistent FID-in-dirent

The FID-in-dirent consistency check/repair will be processed in OSD layer when the higher layer LFSCK traverses the parent directory with namespace-based order.

1. If both the FID-in-dirent and the FID-in-LMA exist, but they do not match and FID-in-LMA is not an IGIF FID, then replace the FID-in-dirent with the FID in the LMA directly.

   ```
   handle = start_trans(update);
   bh = find_entry(dir, dentry, &de);
   journal_get_write_access(handle, bh);
   update_entry(handle, dir, de);
   journal_dirty_metadata(handle, bh);
   brelse(bh);
   stop_trans(handle);
   ```

2. If the FID-in-dirent does not exist for the child object, then the object is either restored from file-level backup or upgraded from 1.8-formatted device. Ideally, the the FID (or IGIF) should be appended to the name entry in the parent directory directly. However, there may not be not sufficient space in the name entry to hold the FID. In this case the name entry is removed from the parent directory and the name entry with the FID is reinserted into the parent directory.

```
if (no FID-in-LMA) {
 /* For 1.8-IGIF object, needs LMA and OI initialization by OI scrub. */
 add to OI scrub pending list;
 handle = start_trans(delete + insert);
 bh = find_entry(dir, dentry, &de);
 journal_get_write_access(handle, bh);
 delete_entry(handle, dir, bh, de);
 journal_dirty_metadata(handle, bh);
 brelse(bh);
 pack_FID(de, FID-in-dirent);
 insert_entry(handle, dir, de);
 stop_trans(handle);
 }
```

5.4 Repair inconsistent linkEA

The linkEA consistency check/repair will be processed in MDD layer when traversing the parent directory in namespace-based order.

1. For single-linked child object, if it has single linkEA entry but not point back to the given parent directory, then the old linkEA will be replaced by the new one:
   ```
   if (unmatched) {
           handle = start_trans(delete);
           remove_linkEA(handle, child, old_linkEA_entry);
           add_linkEA(handle, child, child_name, parent_FID);
           stop_trans(handle);
   }
   ```

2. For a multiple-linked child object, if it has no valid linkEA pointing back to the parent directory, then the valid linkEA entry is added and the child object is pinned in RAM. Other linkEA entries cannot be removed because it is impossible to know if they are valid linkEA entries for other link names. It is also not possible verify other linkEA entries at current time point because the low layer OI scrub may be rebuilding OI files, and the OI mapping for some parent FIDs in the linkEA entries may be not rebuilt yet. Under such case, we cannot locate related parent object according to the FID in the child linkEA entry.

```
handle = start_trans(insert);
 add_linkEA(handle, child, child_name, parent_FID);
 stop_trans(handle);

if (the object is marked as "VERIFIED") {
 return directly; /* do nothing. */
}
if (the object is not pinned in RAM) {
 store the object in the "lfsck_namespace" file;
 pin the object in RAM;
 }

insert new linkEA entry verification record in RAM;
the object verified linkEA entries count ++;

if (the object linkEA entries have all been verified) {
mark the object as "VERIFIED";
remove object linkEA verification history;
}
```

When the first cycle of the object table-based iteration is completed, the LFSCK will rescan the multiple-linkEA objects according to the FIDs in the “`lfsck_namespace`” file. If the object is not mark as "**VERIFIED**", then verify every linkEA entry and remove the invalid linkEA entries as follows:

```
if (the object is marked as "VERIFIED") {
 remove the object from the "lfsck_namespace" file;
 continue for next object;
 }

if (the object is pinned in RAM and the LFSCK has never been restarted) {
 remove all the remaining non-verified linkEA entries directly;
 remove the object from the "lfsck_namespace" file;
 continue for next object;
 }
```

```
for_each_linkEA_entries {
 if (invalid) {
 handle = start_trans(delete);
 remove_linkEA(handle, child, linkEA_entry);
 stop_trans(handle);
 }
 }

 remove the object from the "lfsck_namespace" file;
 continue for next object;
```

5.5 Concurrent link operation during the LFSCK

The current implementation for link operation has already processed the FID-in-dirent and linkEA. Additional work is to handle the linkEA verification statistics correctly.

```
if (target is marked as "VERIFIED") {
return directly; /** do nothing. */
}
if (target was multiple-linked object before the link, and not pinned in RAM) {
store the target in the "lfsck_namespace" file;
pin the target in RAM;
}
insert new linkEA verification record in RAM;
target verified linkEA entries count ++;
```

5.6 Concurrent unlink operation during the LFSCK

The current implementation for unlink operation has already processed the FID-in-dirent and linkEA. Additional work is needed to handle the linkEA verification statistics correctly for multiple-linked objects.

```
if (target is not pinned in RAM) {
return directly;/** do nothing. */
}
if (linkEA entry corresponding to the target name has been verified) {
remove target linkEA verification record from RAM;
target verified linkEA entries count --;
} else if (target linkEA entries have all been verified) {
mark the target as "VERIFIED";
remove target linkEA verification history;
}
if (target becomes single-linked object after the unlink) {
unpin the target from RAM;
remove the target from the "lfsck_namespace" file;
}
```

5.7 Concurrent rename operation during the LFSCK

As an online system consistency maintaining tool, there may concurrent rename operations during LFSCK. These may cause LFSCK to miss objects when traversing directories in namespace-based order. The rename operation will process the source/target objects as LFSCK does: guarantee the consistency of the FID-in-dirent and linkEA. Additional work is needed to handle linkEA verification statistics correctly.

```
if (target object exists) {
 process it as unlink the target;
 }

if (source is single-linked object) {
return directly; /** do nothing. */
}
if (source is not pinned in RAM) {
store the source in the "lfsck_namespace" file;
pin the source in RAM;
}
insert new linkEA verification record in RAM;
if (old linkEA entry has been verified) {
remove old linkEA verification record from RAM;
} else {
source verified linkEA entries count ++;
if (source linkEA entries have all been verified) {
mark the source as "VERIFIED";
remove source linkEA verification history;
}
}
```

6 Race control between the LFSCK and other operations

The existing lock mechanism to control the race between normal object accessing for the RPC services and the LFSCK for FID-in-dirent and linkEA will be used.

6.1 Race control when check/repair FID-in-dirent

Processing the FID-in-dirent is internal to the OSD. ldiskfs PDO lock can be used on the parent object to control the race with lookup/readdir/create/unlink as following:

```
lock_handle = ldiskfs_htree_lock(parent, LDISKFS_HLOCK_ADD);
update or re-insert FID-in-dirent;
ldiskfs_htree_unlock(lock_handle);
```

6.2 Race control when check/repair linkEA

LFSCK will check/repair linkEA consistency in the MDD layer. MDD layer will use the OSD layer read-write lock through the object operations APIs to control the race with other linkEA operations as following:

```
mdd_write_lock(child);
insert or remove linkEA;
mdd_write_unlock(child);
```

7 Compatibility issues

Storing FID-in-dirent entries for 2.x directories is incompatible with Lustre 1.8 formatted filesystems. Once FID-in-dirent is enabled on a Lustre 1.8 formatted filesystem it is not safe to downgrade the MDT to Lustre 1.8. As a result, a switch is needed to control the upgrading explicitly. An ldiskfs filesystem feature “`dir_data`” controls whether FID-in-dirent is used or not.

If the administrator decides to permanently upgrade from Lustre 1.8 to Lustre 2.x then the command "`tune2fs -O dir_data ${mdtdev}`" to will throw the switch. When the LFSCK or create/link/rename operation detects this feature is enabled, it will begin storing FID-in-dirent entries in the Lustre 2.x format as described above.

8 API changes

Most of the LFSCK work for FID-in-dirent and linkEA consistency check/repair can be processed either in MDD layer or OSD layer through existing APIs. The cases where this is not possible are described here:

8.1 New param value for dt_it_ops::init

LFSCK in MDD layer uses the `dt_it_ops::init()` API to initialize the parent directory traversing with namespace-based order. It is possible to implement special iterations methods this such purpose, but reusing the existing iteration methods for MDT `readdir` is more convenient.

```
struct dt_it *(*init)(const struct lu_env *env, struct dt_object *dt,
                      __u32 attr, struct lustre_capa *capa);
```

Currently, the API implementation in OSD layer of osd_it_ea_init() does not know if the directory traversing is for MDT readdir or for LFSCK. As a result, the subsequent iteration does not know if the FID-in-dirent requires verification. A new flag **`LUDA_LFSCK`** for the parameter "`attr`" will be introduced for the intention of calling the `osd_it_ea_init()`.

```
enum lu_dirent_attrs {
 LUDA_FID = 0x0001,
 LUDA_TYPE = 0x0002,
 LUDA_64BITHASH = 0x0004,
 LUDA_LFSCK = 0x0008,
};
```

9 Split patches for code inspection

Large patch sets present reviewers with challenges. This project will be delivered in several smaller patche-sets to simply the process of code reviewing. The patch sets will include the following:

1. General LFSCK framework, including common data structures, variables, macros, tracing file, APIs, and so on.
2. Initial OI scrub to process the server local objects when the server mount.
3. OI scrub changes for process IGIF objects: storing IGIF FID in object LMA, adding IGIF FID OI mapping in OI file.
4. LFSCK main engine, such as device scanning combined otable-based iteration and namespace-based directory traversing, object visibility verification, and so on.
5. FID-in-dirent consistency check/repair.
6. LinkEA consistency check/repair.
7. User space control and interfaces.
8. Test cases.

Changes to this patch list and plan may occur once implementation is underway.