OpenSFS / OpenSFS Lustre Development / Layout Enhancement Design

Layout Enhancement High Level Design

Added by John Hammond, last edited by Richard Henwood on Feb 07, 2014

1. Introduction

The following design applies to the Layout Enhancement project within the Technical Proposal by High Performance Data Division of Intel for OpenSFS Contract SFS-DEV-003 signed Friday 23rd August, 2013.

In the Lustre* file system the data of a file is striped over one or more objects each residing on an OST. The *layout* of a file is an attribute of the file which describes the mapping of file data ranges to object data ranges. The layout is stored on the MDT as a trusted extended attribute (`trusted.lov`) of the file and are sent to clients as needed. The layout of a file is often simply referred to as its *striping*, since in the current (2.5) implementation of the Lustre file system only non-redundant striped (RAID0) layouts are permitted. This project will design enhancements to the representation and handling of layouts to support features such as Data On MDT, File-Level Replication, live data migration (via File-Level Replication), and RAID1/5/6 or erasure coding. The Layout Enhancement (LE) project is therefore a prerequisite for the Data on MDS and File-Level Replication projects, described in separate documents.

In this document we present several new layout types: composite layouts to support replication and layouts extents (sections 2.1 and 2.2), RAID layouts (section 3), compact layouts for widely striped files (section 4), and large layouts (section 5).

2.1. Composite Layouts

In order to support file-level replication and extent based layouts we define a new *composite layout* type which comprises several *simple* (non-composite) layouts designating (partial) mirrors of the file data. This type generalizes RAID0+1 to allow heterogeneous stripe sizes and counts among mirrors and to allow for the possibility of partial mirroring schemes, potentially with each replica on different OST storage pools with different performance and capacity characteristics.

Composite layouts are described by `struct lov_comp_md_v1` which is defined below.

```
struct lu_extent {
       __u64 e_begin;
       __u64 e_end;
};

/* These enum constants are virtual entry ids and are to be used in
 * struct lov_comp_md_op and in composite layout API functions. They
 * specify any or all entries in a composite layout matching a certain
 * condition. Together with a flag that negates the sense of matching
 * they allow us to specify multiple entries in a single operation,
 * for example: delete all entries but the primary from a layout, or
 * get any stale entry. */
enum lcme_id {
        LCME_ID_NONE    = -1,
        LCME_ID_ALL     = -2,
        LCME_ID_ANY     = -3,
        LCME_ID_PRIMARY = -4,
        LCME_ID_STALE   = -5,
};

enum lov_comp_md_entry_flags {
        LCME_FL_PRIMARY   = 1 << 0,
        LCME_FL_STALE     = 1 << 1,
        LCME_FL_OFFLINE   = 1 << 2,
        LCME_FL_PREFERRED = 1 << 3,
};

struct lov_comp_md_entry_v1 {
        __u32 lcme_id;                  /* unique identifier of component */
        __u32 lcme_flags;               /* LCME_FL_XXX */
        struct lu_extent lcme_extent;   /* file extent for component */
        __u32 lcme_offset;              /* offset of component blob in layout */
        __u32 lcme_size;                /* size of component blob data */
        union {
                __u64 lcme_padding;
        } u;
};

enum {
        LOV_COMP_MD_V1_MAGIC = 0x0BD40BD0,
};

enum lov_comp_md_flags {
        /* Replication states, use explained in replication HLD. */
        LCM_FL_RS_READ_ONLY     = 0,
        LCM_FL_RS_WRITE_PENDING = 1,
        LCM_FL_RS_WRITABLE      = 2,
        LCM_FL_RS_SYNC_PENDING  = 3,
        LCM_FL_RS_MASK          = 0xff,

        LCM_FL_PRIMARY_SET      = 1 << 15,
```

A composite layout begins with a `struct lov_comp_md_v1` followed by `lcm_entry_count` entries (instances of `struct lov_comp_md_entry_v1`), followed by `lcm_entry_count` simple layouts (`struct lov_mds_md_v3`, or any other non composite layouts).

> This design does not support nested composite layouts (i.e. components which are themselves composites) to avoid complexity and recursion in the implementation of layout handling. We believe that non-nested layouts provide sufficient flexibility for current projects and anticipated future uses.

Each entry has an identifier (`lcme_id`) which is unique within the lifetime of the composite layout. This is the only way to identify an entry or component within a composite layout. In contrast to existing layout types, composite are intended to updated as components (file replicas) are added and removed. Therefore the particular index on an entry in `lcm_entries` is not a suitable identifier. Entry identifiers are selected by the server side layout handling code as new components are added to a composite layout.

The offset and size members of the entry (`lcme_offset` and `lcme_size`) describe the location of the contained simple layout. We do not assume that the component offsets of the entries are in any particular order nor do we assume any relationship among the component sizes. The offsets are chosen to be multiples of 8 so that components will be suitably aligned in memory. Assume `lcm` points to an in-memory instance of `struct lov_comp_md_v1` which contains three entries (`lcm->lcm_entry_count == 3`).

```
+-----------+ lcm
|           |
|           |
|           |
|           |
+-----------+ &lcm->lcm_entries[0]
|           |
|           |
+-----------+ &lcm->lcm_entries[1]
|           |
|           |
+-----------+ &lcm->lcm_entries[2]
|           |
|           |
+-----------+ (char *)lmm2 == (char *)lcm + lcm->lcm_entries[2].lcme_offset
|           |
|           |
|           | /* lmm2 ends at (char *)lcm + lcm->lcm_entries[2].lcme_size */
+-----------+ (char *)lmm0 == (char *)lcm + lcm->lcm_entries[0].lcme_offset
|           |
|           |
|           |
|           | /* lmm0 ends at (char *)lcm + lcm->lcm_entries[0].lcme_size */
+-----------+ (char *)lmm1 == (char *)lcm + &lcm->lcm_entries[1].lcme_offset
|           |
|           |
|           |
|           |
|           | /* lmm1 ends at (char *)lcm + lcm->lcm_entries[1].lcme_size */
+-----------+ (char *)lcm + lcm->lcm_size
```

Recall that extent based layouts permit different layouts to be used in different extents of a file. They may be used to set progressively wider striping as a file grows in size, to prevent inconsistent out of space errors as individual OSTs become full, and to enable incremental migration, replication, and HSM restore. The extent member (`lcme_extent`) of a composite layout entry describes the range of the file to which the component layout applies. Note that in the simplest case of replication each component would have the same extent of [0, ∞). For a more interesting use of component extents, consider a file *F* which cannot grow in size because one of its objects belongs to an OST with no free space. To handle this the layout L_0 of *F* is converted to a composite layout *C* which contains L_0 as a component with extent [0, *s*) where *s* is the size of *F*. Then a new simple layout L_1 is allocated (with objects on other OSTs) and added to *C* but with extent [*s*, ∞). Now data appended to *F* goes to objects belonging to L_1.

In the design we normally assume that the component extents in a composite layout have the same starting offset at byte 0 of the file. The extents may each form a non-overlapping a subset of [0, ∞), or the may all start at file offset 0, or there may be some other overlap. However, we should try not to use the component extent start as an offset when accessing the component objects. That is, if a component has a single object *O* and extent [*s*, ∞) then the file byte at position *p* should be found at offset *p* of *O* and not at *p* - *s*. In this way an extent with non-zero start can be converted to one which starts at 0. Similarly assume that file data is safely mirrored to another component, a component whose extent starts at zero can be figuratively *punched* to have some positive start without remapping the objects, followed by a punch of corresponding objects data.

2.2. RPCs for Composite Layouts

In order to construct and modify composite layouts a set of new RPCs are defined. These are sent using a single high level RPC with opcode `MDS_LOV_COMP_MD_OP` and format `RQF_MDS_LOV_COMP_MD_OP` based on the existing MDS_SWAP_LAYOUTS RPC. Depending on the operation one or two FIDs are packed in an MDT body along with capabilities and LDLM data. To this a `struct lov_comp_md_op` is added which specifies the exact operation to be performed and any operation specific data.

```
struct req_msg_field RMF_LOV_COMP_MD_OP =
        DEFINE_MSGF("lov_comp_md_op", 0, sizeof(struct lov_comp_md_op),

static const struct req_msg_field *mdt_lov_comp_md_op[] = {
        &RMF_PTLRPC_BODY,
        &RMF_MDT_BODY,
        &RMF_MDT_LOV_COMP_MD_OP,
        &RMF_CAPA1,
        &RMF_CAPA2,
        &RMF_DLM_REQ,
};

struct req_format RQF_MDS_LOV_COMP_MD_OP =
        DEFINE_REQ_FMT0("MDS_LOV_COMP_MD_OP",
                        mdt_lov_comp_md_op, empty);
```

`struct lov_comp_md_op` is defined below.

```
enum lov_comp_md_opc {
        LCMO_MOVE_ENTRY         = 0,
        LCMO_UPDATE_ENTRY       = 1,
        LCMO_SET_PRIMARY        = 2,
};

enum lov_comp_md_op_flags {
        LCMO_FL_ID_NEQ          = 1 << 0, /* negate sense of id matching */
};

struct lov_comp_md_op {
        __u32   lcmo_opc;
        __u32   lcmo_flags;
        __u32   lcmo_layout_gen[2];
        __u32   lcmo_entry_id;
        __u32   lcmo_entry_flags;
        __u32   lcmo_entry_flags_mask;
        __u32   lcmo_padding;
        struct lu_extent lcmo_entry_extent;
};
```

The file(s) to be operated on are identified by `fid1` (and `fid2`) of the MDT body. Files must reside on the MDT receiving the RPC; otherwise no operation is performed and `-EPROTO` is returned. For quota maintenance purposes they must also have the same ownership (UID and GID) or `-EPERM` is returned. If `lcmo_layout_gen[0]` is not -1 then is must be equal to the layout generation of the file identified by `fid1`; otherwise no operation is performed and `-EXXX` is returned. Similarly for `lcmo_layout_gen[1]` and the file identified by `fid2` for operations involving two files.

In each operation `lcmo_entry_id` designates one or more entries in a composite layout by using a known id or by using a virtual id from `enum lcme_id`. Setting `LCMO_FL_ID_NEQ` negates the sense of matching and may only be used with a specific id, `LCME_ID_PRIMARY`, or `LCME_ID_STALE`; otherwise no operation is performed and `-EINVAL` is returned. If either file has a non-composite layout then `LCME_ID_ANY` should be used to specify their single-component. If the designated entries do not exist then no operation is performed and `-EBADSLT` is returned.

`lcmo_extry_extent` must be a valid non-empty extent, have begin and end both 0 (empty), or have begin and end both -1 (special, see below); otherwise no operation is performed and `-EINVAL` is returned.

- `LCMO_MOVE_ENTRY` - move one or more components between the files (Fd and Fs) identified by `fid1` and `fid2`. Fd is the destination and Fs the source. The newly created entries in Fd will have flags and extents equal to `lcmo_entry_flags` and `lcmo_entry_extent`.
- `LCMO_UPDATE_ENTRY` - update the flags and/or extent of the entries in the file (F0) specified by `fid1` of the MDT body. If O is the supplied struct lov_comp_md_op then for easch such entry E we set `E.lcme_flags = (E.lcme_flags & ~O.lcmo_entry_flags_mask) | (O.lcmo_entry_flags & O.lcmo_entry_flags_mask)`. If `O.lcmo_entry_extent` is not `{ -1, -1 }` then we set `E.lcme_extent = O.lcmo_entry_extent` as well.
- `LCMO_SET_PRIMARY` - the designated entry in the file (F0) identified by `fid1` is set primary.

Clients can delete a given set of components from a composite layout by opening a volatile file and moving the set to the volatile file.

3. RAID Layouts

While RAID-1 and RAID-10 layouts offer degrees of replication they are not to be confused with the layouts for file-level replication described above. They are fixed layouts similar to the existing RAID-0 layouts currently used by a Lustre file system. Compared to composite layouts they are simpler and more compact but they are also less expressive and less flexible. RAID1 and RAID-10 layouts may be converted to composite layouts. Composite layouts whose components all have the same stripe size and stripe count may be expressed as RAID1 and RAID10 layouts. Full read/write support for files with RAID-4/5/6 layouts is beyond the scope of this design.

RAID layouts are specified using `struct lov_raid_md_v1` defined below. This format borrows from the SNIA Common RAID Disk Data Format Specification and allows us to specify common simple (0, 1, 4, 5, 6) and nested RAID levels (10, 40, 50, 60).

```
enum {
        LOV_RAID_MD_V1_MAGIC = 0x0A1D0A1D,
};

enum lrm_pattern {
        /* Primary RAID Levels */
        LRM_PRL_0               = 0, /* RAID-0, striped objects */
        LRM_PRL_1               = 1, /* RAID-1, mirrored objects */
        LRM_PRL_4               = 4, /* RAID-4, 0 + parity object */
        LRM_PRL_5               = 5, /* RAID-5, 0 + distributed parity */
        LRM_PRL_6               = 6, /* RAID-6, 0 + 2 distributed parity */
        LRM_PRL_MASK            = 0xff,
        /* RAID Level Qualifiers */
        LRM_56Q_RP0             = 0x000, /* Rotating Parity 0 */
        LRM_56Q_RPN             = 0x100, /* Rotating Parity N */
        LRM_56Q_DR              = 0x000, /* Data Restarts on first object */
        LRM_56Q_DC              = 0x200, /* Data Continues below parity */
        LRM_RLQ_MASK            = 0xf00,

        /* Secondary (Nested) RAID Levels */
        LRM_SRL_0               = 0x0000, /* striped primary groups */
        LRM_SRL_1               = 0x1000, /* mirrored primary groups */
        LRM_SRL_MASK            = 0xf000,

        LRM_RAID_0      = LRM_PRL_0,
        LRM_RAID_1      = LRM_PRL_1,
        LRM_RAID_4      = LRM_PRL_4,
        LRM_RAID_5      = LRM_PRL_5,
        LRM_RAID_6      = LRM_PRL_6 | LRM_56Q_RP0 | LRM_56Q_DR,

        LRM_RAID_10     = LRM_PRL_1 | LRM_SRL_0,
        LRM_RAID_40     = LRM_PRL_4 | LRM_SRL_0,
        LRM_RAID_50     = LRM_PRL_5 | LRM_56Q_RP0 | LRM_56Q_DR | LRM_SRL_0,
        LRM_RAID_60     = LRM_PRL_6 | LRM_56Q_RP0 | LRM_56Q_DR | LRM_SRL_0,

        LRM_FL_RELEASED = 0x8000000, /* LOV_PATTERN_F_RELEASED */
};

struct lov_raid_md_v1 {
        __u32           lrm_magic;
        __u32           lrm_pattern;
        __u32           lrm_layout_gen;         /* u32! */
        __u32           lrm_stripe_size;
        __u16           lrm_count[2];
        __u32           lrm_padding;
        char            lrm_pool_name[16];      /* 16 == LOV_MAXPOOLNAME */
        struct lu_fid   lrm_objects[0];
};

/* sizeof(struct lov_raid_md_v1) == 40 */
```

Non-nested RAID (levels 0, 1, 4, 5, and 6) layouts have a `lrm_count[1]` value of 1. Nested RAID (levels 10, 40, 50, and 60) have a `lrm_count[1]` value greater than 1. In the nested case, the file objects are arranged in to `lrm_group_count` *primary groups* each of which is a RAID set for the primary RAID level (0, 1, 4, 5, 6) with `lrm_count[0]` objects. Then depending on the secondary RAID level these primary groups are mirrored or striped to create the file. For levels 5 and 6 `lrm_count[0]` includes the parity objects (and hence must be at least 3 for RAID-5 and 4 for RAID-6. The number of objects in is RAID layout is always equal to the product `lrm_count[0] * lrm_count[1]`.

4. Compact Layouts for Widely Striped Files

Existing RAID-0 layout formats (`struct lov_mds_md_v1` and `struct lov_mds_md_v3`) use an explicit array of object identifiers to map each stripe index to a specific OST object. When using FIDs alone to identify objects this approach requires 16 bytes per stripe. The current implementation packs an OST index together with the object identifier and needs 24 bytes per stripe. The allocation of memory buffers to transmit, receive, and handle these layouts for very widely striped files (over 160 stripes) can be costly. To avoid this cost we define a compact layout (`struct lov_wide_md_v1`) based on a bitmask of OST indices which reduces memory consumption for widely stripe files by a factor of 192 for the current maximally-striped layout of 2000 stripes.

To make use of this layout type we need to assume that a given MDT can construct a vector `seq` of FID sequences together with a positive integer `stride` such that

- for each valid OST index `i`, either `seq[i] == 0` or `seq[i]` belongs to OST `i` and is allocated to the MDT,
- for each invalid OST index `i`, we have `seq[i] == 0`,
- for all non-negative `integers i` and `j`, if `seq[i] != 0` and `seq[j] != 0` then `seq[i] - seq[j] = (i - j) * stride`.

To be useful the vector `seq` must have many non zero slots. The exact means by which such a vector may be constructed are beyond the scope of this design.

`struct lov_wide_md_v1` is defined below.

```
enum {
        LOV_WIDE_MD_V1_MAGIC = 0x0B160B16,
};

struct lov_wide_md_v1 {
        __u32 lwm_magic;
        __u32 lwm_layout_gen;
        __u32 lwm_stripe_size;
        __u32 lwm_stripe_count;
        char  lwm_pool_name[16];      /* 16 == LOV_MAXPOOLNAME */
        __u64 lwm_obj_seq_begin;      /* used with stride (below) and OST index to
        __u64 lwm_obj_seq_stride;      * compute FID sequence (f_seq) of objects */
        __u32 lwm_obj_oid;            /* common FID OID (f_oid) of all objects */
        __u32 lwm_obj_ver;            /* common FID version (f_ver) of all objects
*/
        __u32 lwm_pattern;
        __u32 lwm_ost_idx_begin;
        __u32 lwm_ost_idx_rotate;
        __u32 lwm_ost_idx_mask_width; /* mask width in bits */
        __u64 lwm_ost_idx_mask[];
};

/* sizeof(struct lov_wide_md_v1) == 72 */
```

Every lov_wide_md_v1 may be converted into an equivalent lov_mds_md_v1 as sketched below. There are three ideas at work:

- The `lwm_ost_idx_*` members describe the set of OSTs which contain objects for the file
- The `lwm_obj_*` specify the FID of an object in the file given its OST index.
- `lwm_ost_idx_rotate` is used to rotate the array of used OST indexes to ensure that the placement of initial stripes is not biased toward lower index OSTs.

```
int lmm_v1_from_lwm_v1(struct lov_user_md_v1 *m, const struct lov_wide_md_v1 *w)
{
        unsigned int stripe_index;
        unsigned int i;
        int rc;

        if (w->lwm_ost_seq_begin == 0 ||
            w->lwm_ost_seq_stride == 0 ||
            w->lwm_ost_idx_mask_width == 0)
                return -EINVAL;

        /* The stripe count must equal the number of bits set in the ost index
mask. */
        if (w->lwm_stripe_count != bit_set_count(w->lwm_ost_idx_mask,
w->lwm_ost_idx_mask_width))
                return -EINVAL;

        /* Ensure m is properly sized for stripe_count objects. */
        rc = lmm_v1_resize_magically(m, LOV_MAGIC_V3, stripe_count);
        if (rc < 0)
                return rc;

        /* m->lmm_oi must be set by caller. */
        m->lmm_magic = LOV_MAGIC_V3;
        m->lmm_pattern = w->lwm_pattern;
        m->lmm_stripe_size = w->lwm_stripe_size;
        m->lmm_stripe_count = w->lwm_stripe_count;
        memcpy(m->lmm_pool_name, w->lwm->pool_name, sizeof(m->lmm_pool_name));

        stripe_index = 0;

        for (i = 0; i < w->lwm_ost_idx_mask_width; i++) {
                const struct lov_ost_data_v1 *o;
                struct lu_fid fid;
                unsigned int ost_idx;

                if (!bit_is_set(w->lwm_ost_idx_mask, i))
                        continue;

                ost_idx = w->lwm_ost_idx_begin + (w->lwm_ost_idx_rotate + i) %
                        w->lwm_ost_idx_mask_width;

                fid.f_seq = w->lwm_obj_seq_begin + ost_idx * w->lwm_obj_seq_stride;
                fid.f_oid = w->lwm_obj_oid;
                fid.f_ver = w->lwm_obj_ver;

                o = &m->lmm_objects[stripe_index];
                fid_to_ostid(&o->l_ost_oi, &fid);
                o->l_ost_idx = ost_idx;

                stripe_index++;
```

5. Handling of Large Layouts

This layout type and handling strategy described below depends on future work related to the Data on MDT project.

There are (at least) two issues with handing large layouts:

- Storing the layout as an extended attribute on an MDT inode limits its size to 64 KB.
- Retrieval of the layout in the initial open RPC requires a large request/reply buffer allocation.
- Layout sizes are constrained by single transfer size limits from lower layers (LNet).

To address these issues we define an indirect layout type which specifies that the file layout is stored as the data of a second *layout* file.

```
enum {
        LOV_IND_MD_V1_MAGIC = 0x0F1DF1D,
};

struct lov_ind_md_v1 {
        __u32 lim_magic;       /* LOV_IND_MD_V1_MAGIC */
        __u32 lim_layout_size;
        struct lu_fid lim_fid; /* FID of the MDT file containing layout */
};
```

Clients encountering this layout type may send OST_READ requests to the MDT using the supplied FID (`lim_fid`) to retrieve the true file layout data. The layout file must have a name (derived from the FID of the original file) in a directory (`LAYOUTS` or similar) on the MDT but this name will not exist in the client visible namespace.

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

Like   Be the first to like this

2 Comments

Nathan Rutman

Should these be __u64?

```
        __u32 lcme_offset;              /* offset of component blob in layout */
        __u32 lcme_size;                /* size of component blob data */
```

Sorry, not clear to me how lwm_ost_idx_rotate differs from lwm_ost_idx_begin?

Where will the cutover from EA to file be? Should probably be before 64K for the large repbuf problem.

`MDT inode limits its size to 64 KB`

Does the MDT support this today, or will this take new development work?

OST_READ requests to the MDT

John Hammond

No. I do not believe that we need to use 64-bit integers for component entry offsets and sizes. Do you?

Without lwm_ost_idx_rotate the first stripe of a widely striped file would always be placed on the OST that has the lowest index among all of the OSTs used. Using lwm_ost_idx_rotate we rotate the array of OST indexes to avoid this.

MDTs do not support OST_READ today. As noted indirect layouts depend on the data on MDT project.