LAID5/LAID3 DLD

February 8, 2008

1 Functional specifiation

Laid5 code consists of four parts:

- Calculation for parity updating and data reconstructing.
- Stripe cache manager. It contains the operations such as add/ remove/ lookup stripe cache.
- IO in case of LAID5. It contains special process for file data IO of LAID5 such as prepare _async_page for parity, mark dirty flag for the stripe units to guide how to update parity, do batched parity updating and sync.
- Truncate handling.

First part contains the following two functions:

1.1 Parity Calculation

1.1.1 Prototype

```
static void ld_compute_parity(struct stripe_head *sh, int method)
```

1.1.2 Parameters

- Input: sh - *stripe_head* want to update
- Input: method - parity calculation method: READ_MODIRY_WRITE, RECONSTRUCT_WRITE.

1.1.3 Return value

1.1.4 Description

This method is called when do parity updating.

1.2 Data unit reconstructing

1.2.1 Prototype

```
static void ld_reconstruct_unit(struct stripe_head *sh, int ds_idx)
```

1.2.2 Parameters

- Input: sh - pointer to *stripe_head* want to do reconstructing operation.
- Input: ds_idx - stripe unit index in the *stripe_head*.

1.2.3 Return value

1.2.4 Description

In degraded mode, after we preread all other stripe units in the stripe_head, we call this funtion to do data unit reconstructing.

Second part: Stripe Cache Manager contains the following functions.

1.3 Get LAID5 private data

1.3.1 Prototype

```
struct ld_private_data *ld_from_lsm(
                struct lov_stripe_md *lsm)
```

1.3.2 Parameters

- Input: lsm - pointer to file stripping meatdata
- Output: pointer of *ld_private_data*

1.3.3 Return value

The method returns a pointer of *ld_private_data* in successful case, error code otherwise.

1.3.4 Description

This method is called when we want to get the laid5 private information from lsm. If the *lsm->lsm_private* is *NULL*, we will allocate and initialize laid5 private data for it. And before release the *lsm* of a file inode , we must first free the *ld_private_date*.

1.4 Add stripe_head to stripecache

1.4.1 Prototype

```
void add_to_stripe_cache(
       struct stripe_head     *sh,
       struct ld_private_data *ld,
       unsigned long offset,
       int ps_idx
       )
```

1.4.2 Parameters

- Input: sh - the *stripe_head* want to add into stripe cache.
- Input: ld - pointer to ld_private_data
- Input: offset - offset in the object, modulo PAGE_SIZE
- Input: ps_idx - stripe unit index of the parity

1.4.3 Return value

1.4.4 Description

This function is used to add newly allocated *stripe_head* into stripe cache.

1.5 Remove *stripe_head* form stripe cache.

1.5.1 Prototype

```
void remove_from_stripe_cache(
          struct stripe_head *sh
          )
```

1.5.2 Parameters

- Input: sh - *stripe_head* want to remove from stripe cache.

1.5.3 Return value

1.5.4 Description

This method is used to remove a *stripe_head* from the stripe cache. Before do this, we should teardown the *_async_page* for parity at first.

1.6 Find stripe head

1.6.1 Prototype

```
static struct stripe_head *find_get_stripe(
                struct ld_private_data *ld,
                 unsigned long index)
```

1.6.2 Parameters

- Input: ld - pointer to *ld_private_data*
- Input: index - index in the object, which is offset in object modulo PAGE_SIZE.

1.6.3 Return value

Retuen the pointer of the *stripe_head* if find, *NULL* otherwise.

1.6.4 Description

This method is called when we want to find a cache *stripe_head* by the index in the object.

1.7 Get *stripe_head*

1.7.1 Prototype

```
static void stripe_cache_get(struct stripe_head *sh)
```

1.7.2 Parameters

- Input: sh - *stripe_head* want to add the reference.

1.7.3 Return value

1.7.4 Description

This method will add the reference of the *stripe_head*.

1.8 Release *stripe_head*

1.8.1 Prototype

```
static void release_stripe(struct stripe_head *sh)
```

1.8.2 Parameters

- Input: sh - the stripe head want to release.

1.8.3 Return value

1.8.4 Description

This method is called when release a stripe head. If the reference count of the stripe head become zero, we will finially free it.

1.9 grab cache stripe

1.9.1 Prototype

```
static struct stripe_head *grab_cache_stripe(
             struct lov_stripe_md *lsm,
             unsigned long index,
             int ps_idx)
```

1.9.2 Parameters

- Input: lsm - file stripping information
- Input: index - index in the object.
- Input: ps_idx - parity stripe number

1.9.3 Return value

The method returns a *stripe_head* pointer in successful case, error code othewise.

1.9.4 Description

We use this function to get a *stripe_head*, its function is like *grab_cache_page* : If the stripe head with *index* offset is found in the stripe cache, we will return it; Otherwise we will allocate a new one , initilize and add it to the cache.

1.10 Gang stripecache lookup

1.10.1 Prototype

```
unsigned shvec_lookup(struct shvec *shvec,
        struct ld_private_date *ld,
        pgoff_t start, unsigned nr_shs
        );
```

1.10.2 Parameters

- Input: shvec - Where the resulting stripe_heads are placed.
- Input: ld - The *ld_private_date* to search.

- Input: start - The starting stripe_head index
- Input: nr_shs - The maximum number of stripe_heads.

1.10.3 Return value

Return the number of stripe_heads which were found.

1.10.4 Description

This method will search for and return a group of up to @nr_shs stripe_heads in the stripe cache. The stripe_heads are placed in @shvec.

1.11 Gang stripecache lookup with tag

1.11.1 Prototype

```
static void shvec_lookup_tag(struct shvec *shvec,
        strcut ld_private_data *ld, pageoff_t *index,
        int tag, unsigned nr_shs);
```

1.11.2 Parameters

- Input: shvec - Where the resulting stripe_heads are placed.
- Input: ld - The ld_private_date to search.
- Input: start - The starting stripe_head index
- Input: tag - Radix-tree tags
- Input: nr_shs - The maximum number of stripe_heads.

1.11.3 Return value

Return number of stripe_head which were found.

1.11.4 Description

This method will search for and return a group of up to @nr_shs stripe_heads in the stripe cache with tag @tag. The stripe_heads are placed in @shvec.

Third Part: IO in case of LAID5.

1.12 Lock stripe_head

1.12.1 Prototype

```
static void lock_stripe_head(
        struct stripe_head *sh
        )
```

1.12.2 Parameters

- Input: sh - stripe_head want to lock.

1.12.3 Return value

1.12.4 Description

This function is used to lock the entire stripe_head.

1.13 Unlock stripe_head

1.13.1 Prototype

```
static void unlock_stripe_head(
        struct stripe_head *sh
        )
```

1.13.2 Parameters

- Input:sh - the stripe_head want to unlock

1.13.3 Return value

1.13.4 Description

This function is used to unlock the entire stripe_head.

1.14 Get parity stripe number

1.14.1 Prototype

```
int lov_ps_stripe(
            struct lov_stripe_md *lsm,
            obd_off lov_off
            )
```

1.14.2 Parameters

- Input: lsm - file stripping metedata.
- Input: lov_off - offset in the file.

1.14.3 Return value

Return the parity stripe number the file offset @lov_off will be written into.

1.14.4 Description

This function is usually used to get the corresponding parity stripe number of data stripe unit with file offset @lov_off.

1.15 Extend the lock extent

1.15.1 Prototype

```
int lov_extend_lock_extent(
            struct lov_stripe_md *lsm,
            obd_off lov_off,
            obd_off *obd_off, int start
            )
```

1.15.2 Parameters

- Input: lsm - pointer to lov_stripe_md
- Input: lov_off - offset in the file.
- Output: obd_off - offset in the object.
- Input: start - indicate it is start or end of the lock extent.

1.15.3 Return value

Return the extended lock extent.

1.15.4 Description

This function is used to get extended lock extent with group-size-granularity.

1.16 Remove stripe cache

1.16.1 Prototype

```
static int o_remove_stripe_cache(
        struct obd_export *exp,
        struct lov_stripe_md *lsm,
        obd_off start, obd_off end,
        struct obd_trans_info *oti
        )
```

1.16.2 Parameters

- Input: exp - next stack obd export.
- Input: lsm - metadata information of the file.
- Input: start- start in the object.
- Input: end - end in the object.

1.16.3 Return value

Return 0 in successful case, error code otherwise.

1.16.4 Description

We use this function to remove the cached *stripe_head* from stripe cache in the range: [start, end].

1.17 lov_prep_preread_set

1.17.1 Prototype

```
static int lov_prep_preread_set(
      struct obd_export *exp,
      struct lov_stripe_md *lsm,
      struct obdo *src_oa,
      struct laid_update_group *lug,
      struct obd_trans_info *oti,
      struct lov_request_set **request)
```

1.17.2 Parameters

- Input: exp - next stack obd export.
- Input: src_oa - src object information.
- Input: lsm - file stripping metedata.
- Input: lug - pointer of *laid_update_group* which contains the list of *stripe_heads* need to preread data.
- Input: oti - pointer to *obd_trans_info*
- Outptu: request - return request set

1.17.3 Return value

The method returns 0 in successful case, error code otherwise.

1.17.4 Description

We use this function to prepare preread request set for preread operation in case of LAID5 and LAID3.

1.18 lov_stripe_preread

1.18.1 Prototype

```
static int lov_stripe_preread(
      struct obd_export *exp,
      struct lov_stripe_md *lsm,
      struct obdo *src_oa,
      struct laid_update_group *lug,
      struct obd_trans_info *oti)
```

1.18.2 Parameters

- Input: exp - next stack obd export.
- Input: src_oa - src object information.
- Input: lsm - pointer to lov_stripe_md.
- Input: lug - pointer of laid_update_group, which contains the list of *stripe_heads* need to preread.
- Input: oti - pointer to obd_trans_info.

1.18.3 Return value

The method returns 0 in successful case, error code otherwise.

1.18.4 Description

we use this function to do batched preread of stripe unit data in case of LAID5 and LAID3 in synchronous way.

1.19 initialize *laid_update_group*

1.19.1 Prototype

```
static void lug_init(
      struct laid_update_group **lug,
      struct lov_stripe_md *lsm,
      int cmd, int stripeno
      )
```

1.19.2 Parameters

- Output: lug - pointer to laid_update_group.
- Input: lsm - file stripping information
- Input: cmd - r/w command.
- Input: stripeno - number of the stripe want to do parity update.

1.19.3 Return value

1.19.4 Description

This funcation is used to initialize a *laid_update_group*.

1.20 Add a *stripe_head* to *laid_update_group*

1.20.1 Prototype

```
static void lug_add(
      struct laid_update_group *lug,
      struct stripe_head *sh)
```

1.20.2 Parameters

- Input: lug - pointer to laid_update_group.
- Input: sh - pointer to *stripe_head* want to add to the *lug*.

1.20.3 Return value

1.20.4 Description

This funcation is used to add *stripe_head* to *laid_update_group*.

1.21 queue the updating *stripe_head*

1.21.1 Prototype

```
static int .ap_handle_stripe(
             void *data,
             int cmd,
             obd_flags flags,
             struct laid_update_group **lug)
```

1.21.2 Parameters

- Input: data - pointer to lov_async_page
- Input: cmd - read/write command.
- Input: flags - flags for update.
- Input/Output: lug - *laid_update_group* used to do batched parity updating or data reconstructing.

1.21.3 Return value

The method returns 0 in successful case, error code otherwise.

1.21.4 Description

This is a upcall from OSC layer to LOV layer,which is mainly used by LAID5/LAID3 case. In the read case, it is used to check whether it can get data on client via XORing calculation of all other stripe unit data if cache enough stripe unit data. In the write case, it is used to queue group of preread set for doing batched parity updating in future; In the degraded mode of read request, we also use it to queue set of preread stripe unit for later reconstructing.

1.22 Trigger Batched parity updating

1.22.1 Prototype

```
static int .ap_trigger_update(
             void *data,
             int cmd,
             obd_flags flags,
             struct laid_update_group *lug)
```

1.22.2 Parameters

- Input: data - pointer to lov_async_page
- Input: cmd - read/write command.
- Input: flags - flag for update.
- Input/Output: lug - laid_update_group used to do batched parity updating or data reconstructing.

1.22.3 Return value

The method returns 0 in successful case, error code otherwise.

1.22.4 Description

This is a upcall from OSC layer to LOV layer,which is mainly used by LAID5/LAID3 case. Via this funcation we will trigger the updating, do the batched preread and then batched parity calculation or data reconstructing. After the parity update, we queue the parity asynchronous page, trigger the batched sync for pairy asynchronous pages.

1.23 prepare parity _async_page

1.23.1 Prototype

```
static int lov_prep_ps_async_page(struct obd_export *exp,
             struct lov_stripe_md *lsm,
             obd_off lov_off, obd_off sub_off,
             struct page *page,
             struct lov_async_page *ds_lap)
```

1.23.2 Parameters

- Input: exp - next stack obd export.
- Input: lsm - file stripping information.
- Input: lov_off - offset in the file.
- Input: sub_off- offset in the object.
- Input: page - file cache page.
- Input: ds_lap - lov_async_page of data stripe unit.

1.23.3 Return value

The method returns 0 in successful case, error code otherwise.

1.23.4 Description

We use this function to prepare the _async_page for parity.

1.24 Grab cache space for parity

1.24.1 Prototype

```
static int lov_ps_grab_cache(struct obd_export *exp,
             struct lov_stripe_md *lsm,
             struct lov_async_page *ds_lap)
```

1.24.2 Parameters

- Input: exp - next stack obd export.
- Input: lsm - file stripping information.
- Input: ds_lap - *lov_async_page* of data stripe unit.

1.24.3 Return value

The method returns 0 in successful case, error code otherwise.

1.24.4 Description

We use this function to grab cache for parity. If the stripe_head is mark as SH_GRANT_SPACE, we just return success.

1.25 queue async io for parity

1.25.1 Prototype

```
static int lov_queue_ps_async_io(struct obd_export *exp,
            struct lov_stripe_md *lsm,
            struct lov_async_page *ds_lap,
            int cmd, obd_off off, int count,
            obd_flag brw_flags, obd_flag async_flags)
```

1.25.2 Parameters

- Input: exp - next stack obd export.
- Input: lsm - pointer to lov_stripe_md
- Input: ds_lap- *lov_async_page* of data stripe unit.
- Input: cmd - read/write command.
- Input: off - offset in the page.
- Input: count - read/write count.

1.25.3 Return value

The method returns 0 in successful case, error code otherwise.

1.25.4 Description

This function is mainly used in write case. After queue async page for IO data, we use this funcation to mark the corresponding stripe_head and stripe_unit as DIRTY.

1.26 Set async flags for parity

1.26.1 Prototype

```
static int lov_set_ps_async_flags(struct obd_export *exp,
             struct lov_stripe_md *lsm,
             struct lov_oinfo *loi,
             struct lov_async-page *ds_lap,
             obd_flag async_flags)
```

1.26.2 Parameters

- Input: exp - next stack obd .
- Input: lsm - file stripping information.
- Input: loi - object information.
- Input: ds_lap- *lov_async_page* of data stripe unit.
- Input: async_flags - asynchronous IO flags.

1.26.3 Return value

The method returns 0 in successful case, error code otherwise.

1.26.4 Description

After set async flags and trigger the sync for the data stripe unit, we use this function to set async flags for parity *_async_page*, and trigger the update and sync for parity.

1.27 queue group IO for parity

1.27.1 Prototype

```
static int lov_queue_ps_group_io(struct obd_export *exp,
             struct lov_stripe_md *lsm,
             struct lov_oinfo *loi
             struct obd_io_group *oig,
             strcut lov_async_page *ds_lap,
             int cmd, obd_off off, int count,
             obd_flag brw_flags, obd_flag async_flags)
```

1.27.2 Parameters

- Input: exp - next stack obd export.
- Input: lsm - file stripping information.

- Input: loi - object informaion.
- Input: oig - pointer to obd_io_group.
- Input: ds_lap - *lov_async_page* of data stripe unit.
- Input: cmd - read/write command.
- Input: off - offset in the page.
- Input: count- read/write count.

1.27.3 Return value

The method returns 0 in successful case, error code otherwise.

1.27.4 Description

The function is similar with *lov_queue_ps_async_io*. In this function, we first lock the stripe, and then mark the *stripe_head* and stripe unit corresponded to the asynchronous page as dirty, after that unlock the stripe.

1.28 Teardown parity _async_page

1.28.1 Prototype

```
static int lov_teardown_ps_async_page(struct obd_export *exp,
            struct lov_stripe_md *lsm,
            struct lov_async_page *ds_lap)
```

1.28.2 Parameters

- Input: exp - next stack obd export.
- Input: lsm - file stripping information.
- Input: ds_lap - *lov_async_page* of data stripe unit.

1.28.3 Return value

The method returns 0 in successful case, error code otherwise.

1.28.4 Description

In case of LAID5/LAID3, we will call this function to decrease the reference of *stripe_head* in function *lov_teardown_async_page*. If the reference count of *stripe_head* become zero, we will also need to teardown the *_async_page* of parity stripe unit.

1.29 Prepare brw set in case of LAID5/LAID3

1.29.1 Prototype

```
static int lov_r5_prep_brw_set(
        int cmd,
        struct obd_export *exp,
        struct obdo *src_oa,
        struct lov_stripe_md *lsm,
        obd_count oa_bufs,
        struct brw_page *pga,
        struct obd_trans_info *oti,
        struct lov_request_set **reqset)
```

1.29.2 Parameters

- Input: cmd - read/write command.
- Input: exp - next stack obd export.
- Input:src_oa - object information.
- Input: lsm - file stripping information.
- Input: pga - brw page array.
- Input: oa_bufs - count of brw pages in the array.
- Input: oti - pointer to obd_trans_info.
- Output: reqset - return requset set.

1.29.3 Return value

The method returns 0 in successful case, error code otherwise.

1.29.4 Description

In case of LAID5/LAID3, we will call this function to prepare brw set for synchronous IO in case of LAID5/LAID3.

Four part: truncate handling.

1.30 Truncate stripe cache

1.30.1 Prototype

```
static int o_truncate_stripe_cache(
             struct obd_export *exp,
             struct lov_stripe_md *lsm,
             obd_off start, obd_off end,
             struct obd_trans_info *oti)
```

1.30.2 Parameters

- Input: exp - next stack obd.
- Input: lsm - file stripe information.
- Input: start - start in the file.
- Input: end - end in the file.
- Input: oti - pointer to obd_trans_info.

1.30.3 Return value

The method returns 0 in successful case, error code otherwise; Otherwise return error code .

1.30.4 Description

We use this method to truncate the stripe cache in the range [start, end], which function is similar with *vmtruncate*.

1.31 Update parity for unaligned truncate

1.31.1 Prototype

```
static int o_truncate_update(
             struct obd_export *exp,struct obdo *oa,
             struct lov_stripe_md *lsm,
             obd_off new_size, obd_off orig_size,
             struct obd_trans_info *oti)
```

1.31.2 Parameters

- Input: exp - next stack obd.
- Input: oa - object information.
- Input: lsm - file stripe information.
- Input: start - size after truncate.

- Input: orig_size - size before truncate.
- Input: oti - pointer to obd_trans_info.

1.31.3 Return value

The method returns 0 in successful case, error code otherwise; Otherwise return error code .

1.31.4 Description

We use this method to update the parity in the last unaligned stripe group for shrink truncate.

2 Use case

2.1 File IO case

The call chain of file write in case of LAID5/LAID3 looks like the following:

```
ll_commit_write()
  |
  |->llap_from_page
  |       |->lov_prep_async_page -> lov_prep_ps_async_page
  |
  |  /* if (!PageDirty(page))*/
  |->queue_or_sync_write
  |       |->lov_grab_cache ->lov_ps_grab_cache
  |       |->lov_queue_async_io -> lov_queue_ps_async_io
  |       |->oig_init
  |       |->lov_queue_group_io -> lov_queue_ps_group_io
  |       |->lov_trigger_group_io
  |       |->oig_wait
  |
  |   /*else*/
  |->obd_hit_dirty -> lov_queue_ps_async_io

ll_writepage()
  |
  |->llap_from_page
  |
  |  /*if (llap->llap_write_queued)
  |->lov_set_async_flags ->lov_set_ps_async_flags
  |  /*else*/
  |->queue_or_sync_write
```

For file IO in case of LAID5/LAID3, we must first grant the extended extent lock with stripe-group-size granularity. As parity stripe unit is not part of file data, so we must do special process for parity, for example:

- When prepare *_async_page* for data stripe unit, we should also prepare *_async_page* for parity stripe unit if it hasn't created and initialized.
- After grab cache space for data stripe unit, we should also grab cache space for parity. Neither of them fails, we must do the write in synchronous way.
- After queue async/group io for the data stripe unit, we should mark the stripe unit and stripe head as dirty, which can guide how to preread and update parity.

2.2 Truncate case

The call chain of truncate in case of LAID5/LAID3 looks like the following:

```
ll_setattr_raw()
   |->vmtruncate()
        |->ll_truncate()
ll_truncate()
   |->obd_truncate_stripe_cache()
   |->ll_inode_size_unlock()
   |->obd_truncate_update()
   |->obd_punch()
```

In case of LAID5/LAID3, after truncate handling for file cache mapping in *vmtruncate*, we should remove the stripe cache in the truncate range via function *obd_truncate_stripe_cache*. If it is a partial truncate, that is to say the file size after truncate is not stripe-group-size aligned, we should first zero out the stripe unit in the left of stripe group out of the file size , and then compute parity and sync it to OST.

3 Logic specification

3.1 Data structure definition

Each cached stripe unit express as the following tuple:

```
struct stripe_unit {
    struct list_head   su_item;
    struct page       *su_page;
    struct page       *su_cache;
    unsigned long      su_flags;
};
```

- su_item: list_head used for parity updating.

- su_page: pointer to the file cache page associated the data stripe unit.
- su_cache: cache of the old stripe unit data.
- su_flags: flags such as SU_INIT, SU_DIRTY, SU_UPDATE, SU_PREREAD.

We use data structure *stripe_head* to manage the strie units in the stripe row. The structure is as following:

```
struct stripe_head {
    struct lov_stripe_md       *sh_lsm;
    struct list_head            sh_item;
    struct list_head            sh_lock_item;
    struct list_head            sh_ready_item;
    struct lov_async_page      *sh_pslap;
    struct page                *sh_parity;
    spinlock_t                  sh_lock;
    unsigned long               sh_index;
    int                         sh_psidx;
    int                         sh_nrcached;
    atomic_t                    sh_count;
    unsigned long               sh_flags;
    struct stripe_unit          sh_unit[1];
};
```

- sh_lsm: file stripping information
- sh_itme: *list_head* used for parity upating.
- sh_pslap: the *lov_async_page* for parity stripe unit.
- sh_parity: pointer of page of parity.
- sh_lock: spinlock protecting to change the flags.
- sh_index: index in the object, which is offset in object modulo STRIPE_SIZE(4k).
- sh_psidx: parity index in all stripe units.
- sh_nrcached: count of stripe units already cached.
- sh_count: reference count of *stripe_head*.
- sh_flags: various flags such as SH_QUEUED, SH_GRANT_SPACE.
- sh_unit: array of stripe units.

We use data structure *ld_private_data* to manager all cached *stripe_heads* of the file. It is as following:

```
struct ld_private_data {
    int                     ld_magic;
    struct lov_stripe_md   *ld_lsm;
    struct obd_export      *ld_exp;
    struct radix_tree_root  ld_stripe_tree;
    spinlock_t              ld_lock;
    unsigned long           ld_nrstripes;
};
```

- ld_magic: magic number.
- ld_lsm: pointer to file's *lov_stripe_md.*
- ld_exp: pointer to the lov layer obd.
- ld_stripe_tree: radix tree of all *stripe_heads.*
- ld_lock: spinlock protecting the radix tree.
- ld_nrstripes: count of *stripe_heads* in cache.

In some places such as truncate operation, we need to batch an operation up against multiple *stripe_heads.* we use data structure *shvec* as a muli-stripehead container (it is similar with *pagevec*):

```
sttuct shvec {
    unsigned nr;
    int cold;
    struct stripe_head *stripes[SH_VEC_SIZE];
};
```

We use data struct laid_update_group to batch prereading and updating for parity.

```
struct laid_update_group {
   spinlock_t   lug_lock;
   atomic_t     lug_count;
   int          lug_pending;
   int          lug_rc;
   int          lug_cmd;
   int         *lug_info;
   int          lug_stripeno;
   struct lov_stripe_md *lug_lsm;
   struct list_head  lug_locked_list;
   struct list_head  lug_preread_list;
   struct list_head  lug_update_list;
   struct list_head  lug_rcw_list;
   struct list_head  lug_rmw_list;
};
```

- lug_count: reference count.
- lug_pending: number of *stripe_heads* in this group.
- lug_cmd: indicate that update the stripe_heads in the group for parity write or read in degraed read.
- lug_lsm: file stripping information.
- lug_info: information of number of brw pages issue to various objects.
- lug_stripeno: the number of stripe doing sync and needing to update parity.
- lug_locked_list: list maintains the stripe_heads locked in this group.
- lug_preread_list: list maintains the *stripe_heads* need to preread uncached data.
- lug_update_list: list maintains the *stripe_heads* need to do parity calculation.
- lug_rcw_list: list maintains the *stripe_heads* want to update via method RECONSTRUCT_WRITE.
- lug_rmw_list: list maintains the *stripe_heads* want to update via method READ_MODIFY_WRITE.

We need also do some modification of the following data structure:

```
struct lov_stripe_md {
   spinlock_t     lsm_lock;
   ....
   void          *lsm_private;
   struct lov_oinfo lsm_oinfo[0];
};
```

- lsm_private: pointer to *ld_private_data* in case of LAID5/LAID3.

```
struct lov_async_page {
   int             lap_magic;
   struct obd_async_page_ops  *lap_caller_ops;
   void           *lap_private;
   obd_off         lap_off;
   obd_flag        lap_flags;
   atomic_t        lap_remaining;
   int             lap_rc;
   int             lap_nrsub;
   int             lap_type;
   int             lap_first_stripe;
```

```
    struct list_head   lap_queued;
    struct list_head   lap_grabd;
    struct list_head   lap_subs;
};
```

- lap_private: pointer to the corresponding *stripe_head* in case of LAID5/LAID3.
- lap_off: offset in the file.
- lap_remaining: count of lov_async_page_sub need to sync.
- lap_rc: finial return code. Just when all sub mirrors fail, *lap_rc* is error code; Otherwise, *lap_rc* is 0.
- lap_type: type of the lov_async_page: LAP_TYPE_DATA, LAP_TYPE_PARITY.
- lap_first_stripe: first stripe number of all mirrors.
- lap_queued: list maintains the mirror sub async page have already queed.
- lap_grabed: list maintains the morror sub async page have already grabed cache space.
- lap_subs: used for linking all *lov_async_page_subs*.

```
struct lov_request {
    struct list_head   rq_link;
    ...
    struct brw_page   *rq_pga;
    ...
};
```

- rq_pga: *brw_page* array for write request in case of LAID5/LAID3 as parity is not part of file data, we can not place them togerther in *lov_request_set.set_pga*, so we add a member *rq_pga* in *lov_request*.

```
struct lov_request_set {
    atomic_t   set_refcount;
    ...
    struct laid_update_group *set_lug;
}
```

- set_lug: For read request in degraded mode, we use set_lug to gather the stripe_heads need to reconstrut.

```
struct loi_oap_pages {
    struct list_head  lop_pending;
    int               lop_num_pending;
    struct list_head  lop_urgent;
    struct list_head  lop_pending_group;
    struct list_head  lop_parity_group;
};
```

- lop_parity_group: list maintains the pending parity asynchronous page.

```
struct lov_oinfo {
   __u64  loi_id;
   __u64  loi_gr;
   ...
   struct loi_oap_pages loi_read_lop;
   struct loi_oap_pages loi_write_lop;
   sttuct loi_oap_pages loi_parity_lop;
   ...
};
```

- loi_parity_lop: It is used to maintain the parity asynchronous page.

3.2 ld_from_lsm implementation

The purpose of this method it to get the *ld_private_data* of the file. If it is *NULL*, we should also create and initialize it.

```
struct ld_private_data *ld_from_lsm(
                        struct lov_stripe_md *lsm)
{
    struct ld_private_date *ld;

    LASSERT(PATTERN_PARITY(lsm));
    ld = (struct ld_private_data *)lsm->lsm_private;
    if (ld != NULL)
       RETURN(ld);

    OBD_ALLOC(ld, sizeof(*ld));
    if (ld == NULL)
       RETURN(ERR_PTR(-ENOMEM));

    INIT_TADIX_TREE(&ld->ld_stripe_tree, GFP_ATOMIC);
    spin_lock_init(&ld->ld_lock);
    ld->ld_lsm = lsm;
    lsm->lsm_private = ld;

    RETURN(ld);
}
```

3.3 Lock operation of *stripe_head*

When modify the content of spare page or update the parity, we must lock the *stripe_head* to block any change for the stripe units. we achieve this purpose via method *lock_stripe*, *unlock_stripe* which borrows the *page* locking operation.

```
#define lock_stripe(sh)   lock_page((sh)->sh_parity)
#define unlock_stripe(sh) unlock_page((sh)->sh_parity)
#define StripeDirty(sh)   PageDirty((sh)->sh_parity)
#define TryLockStripe(sh) TryLockPage((sh)->sh_parity)
#define clear_stripe_dirty(sh) clear_page_dirty((sh)->sh_parity)
int cache_has_stripes(struct ld_private_data *ld)
{
   return (ld->ld_nrstripes > 0);
}
```

3.4 File IO in case of writeback cache implementation

3.4.1 Prepare async IO for parity

```
int lov_prep_async_page (...)
{
   ....
   if (ops == NULL)
        lap->lap_type = LAP_TYPE_PARITY;
   ...
   /*after prepare _async_page for data stripe unit,
    * we should also prepare it for parity stripe unit.
    */
   if (PATTERN_PARITY(lsm) && ops != NULL) {
    /*in case of LAID5, parity has no obd_async_page_ops
     *upcall to llite layer
     */
        rc = lov_prep_ps_async_page(...);
        if (rc)
            GOTO(out_err, rc);
   }
   RETURN(0);
   ....
}
int lov_prep_ps_async_page(exp, lsm, page, lov_off, sub_off, ds_lap)
{
   struct lov_async_page *lap; /* lap for parity*/
   struct ld_private_date *ld;
   struct stripe_head *sh;
   struct stripe_unit *su;
   unsigned long index;
   int ps_idx, ds_idx;
   ...
   ld = ld_from_lsm(lsm);
   if (IS_ERR(ld))
       RETURN(PTR_ERR(ld));
```

```
    if (ld->ld_exp == NULL)
        ld->ld_exp = exp;

    index = sub_off >> STRIPE_SHIFT;
    ps_idx = lov_ps_number(lsm, lov_off);
    ds_idx = ds_lap->lap_frist_stripe;
    sh = _grab_cache_stripe(lsm, index, ps_idx);
    if (sh == NULL)
         RETURN(-ENOMEM);

    ds_lap->lap_private = sh;

    rc = init_stripe_head(sh);
    if (rc)
        GOTO(out, rc = -ENOMEM);

    /*hit the hole or this is a file cache page of append write*/
    loi = &lsm->lsm_oinfo[ds_idx];
    if (sub_off > loi->loi_rss) {
        su = &sh->sh_unit[ds_idx];
        memset(kmap(su->su_cache), 0, PAGE_SIZE);
        kunmap(su->su_cache);
        set_bit(SU_UPDATE, &sh->sh_unit[ds_idx].su_flags);
    }

    init_stripe_unit(sh, page, ds_idx);
    if (sh->sh_pslap != NULL && test_bit(SH_INIT, &sh->sh_flags))
        RETURN(0);

    /*prepare _async_page for parity stripe unit*/
    OBD_ALLOC(lap, sizeof(*lap));
    if (lap == NULL)
          GOTO(out, rc = - ENOMEM);

    rc = obd_prep_async_page(exp, lsm, NULL, sh->sh_parity,
              offset, NULL, NULL, lap);
    if (rc)
          GOTO(out, rc);
    lap->lap_private = sh;
    sh->sh_pslap = lap;
    set_bit(SH_INIT, &sh->sh_flags);
    RETURN(0);
out:
    if (lap != NULL)
        OBD_FREE(lap, sizeof(*lap));
```

```
    stripe_cache_release(sh);
    RETURN(rc);
}
```

In function *lov_prep_ps_async_page*, we call *init_stripe_head* to allocate cache pages for stripe units if the stripe_head is not marked as SU_INIT.

3.4.2 Queue async IO for parity

```
static int lov_queue_async_io()
{
    ...
    /*after queue async IO for data stripe unit, we
     * should also queue async IO for parity*/
    if (PATTERN_PARITY(lsm) && (lap->lap_type == LAP_TYPE_DATA)) {
        err = lov_queue_ps_async_io(exp, lsm, lap,,,);
        ....
    }
    ...
}
static int lov_queue_ps_async_io(exp, lsm, ds_lap, cmd, off, count,
             brw_flags, async_flags)
{
    ...
    ds_idx = ds_lap->lap_first_stripe;
    su = &sh->sh_unit[ds_idx];
    lock_stripe(sh);
    set_bit(SH_DIRTY, &sh->sh_flags);
    set_bit(SU_DIRTY, &su->su_flags);
    unlock_stripe(sh);
    REUTRN(rc);
}
```

3.4.3 Set aync flags for parity

```
int lov_set_ps_async_flags(exp, lsm, loi, ds_lap, async_flags)
{
    ...
    sh = SH_FROM_LAP(ds_lap);
    rc = obd_set_async_flags(., sh->sh_pslap, .);
    /*grace error of OSS failure*/
    if (rc == -EINVAL)
        rc = 0;
    RETURN(rc);
}
```

3.4.4 queue group IO for parity

```
static int lov_queue_group_io(...)
{
    ....
    /*After queue group IO for data stripe unit*/
    if (cmd == OBD_BRW_WRITE && PATTERN_PARITY(lsm) &&
        lap->lap_type == LAP_TYPE_DATA) {
       err = lov_queue_ps_group_io(...);
    ...
}
intlov_queue_ps_group_io(exp, lsm, loi, oig, ds_lap, cmd, off, count...)
{
   ...
   ds_idx = ds_lap->lap_first_stripe;
   sh = SH_FROM_LAP(ds_lap);
   su = &sh->sh_unit[ds_idx];
   lock_stripe(sh);
   set_bit(SH_DIRTY, &sh->sh_flags);
   set_bit(SU_DIRTY, &su->su_flags);
   unlock_stripe(sh);
   RETURN(rc);
}
```

3.4.5 Teardown async_page for parity

```
static int lov_treadown_async_page(...)
{
    ...
    /*After teardown the next layer osc_async_page*/
    if (PATTERN_PARITY(lsm)) {
        if (lap->lap_type = LAP_TYPE_DATA) {
               err = lov_teardown_ps_async_page(exp, lsm, lap);
               if (err && !rc)
                   rc = err;
        } else {  /*LAP_TYPE_PARITY*/
               OBD_FREE(lap, sizeof(*lap));
        }
    }
    RETURN(rc);
}
int lov_teardown_ps_async_page(exp, lsm, ds_lap)
{
   ...
   ds_idx = ds_lap->lap_first_stripe;
   clear_stripe_unit(sh, ds_idx);
```

```
   stripe_cache_release(sh);
   return(0);
}
void stripe_cache_release(sh)
{
  struct ld_privaet_data *ld;
  struct obd_export *exp;
  struct lov_stripe_md *lsm;
  int rc;

  ld = LD_FROM_LSM(sh->sh_lsm);
  exp = ld->ld_exp;
  if (atomic_dec_and_test(&sh->sh_count)) {
      remove_from_stripe_cache(sh);
      /*we may teardown the osc_async_page in truncate stripe cache*/
      if (!test_bit(SH_FREE_AP, &sh->sh_flags) ||
          sh->sh_pslap != NULL)
            rc = obd_teardown_async_page(exp, lsm, NULL, sh->sh_pslap);
            sh->sh_pslap = NULL;
      free_stripe_head(sh);
  }
}
```

3.5 Batch parity updating

When dirty the file cache page, we just mark the corresponding data stripe unit and parity stripe unit as SU_DIRTY, delay the updating untile *pdflush* starts batched sync, and at that time we do batched preread, parity updating and batched parity sync.

To support batched parity updating, we need to modify some methods:

```
static int lov_ap_make_ready(void *data, int cmd)
{
   ...
   ENTRY;

   if (lap->lap_type == LAP_TYPE_PARITY) {
       sh = SH_FROM_LAP(lap);
       if (TryLockStripe(sh))
            RETURN(-EAGAIN);
       clear_stripe_dirty(sh);
       return 0;
   }
   ...
}
```

For upcall of *.ap_refresh_count* of parity, we just retrun PAGE_SIZE.

```
static lov_ap_refresh_count(void *data, int cmd)
{
   ...
   if (lap->lap_type == LAP_TYPE_PARITY)
        return PAGE_SIZE;
   return lap->lap_caller_ops->ap_refresh_cout(...);
}

static void lov_ap_fill_obdo(data, cmd, oa)
{
   ...
   /*FIXME: It'd better to pass the uid and gid here
    * for parity which is needed by quota*/
   if (lap->lap_type == LAP_TYPE_parity) {
       oa->o_valid = OBD_MD_FID | OBD_MD_FLMDOE |
                     OBD_MD_FLTYPE;
       return;
   }
   ...
}
static void lov_ap_completion(...)
{
  ...
  if (rc == 0)
     lap->lap_rc = 0;
  if (atomic_dec_and_test(&lap->lap_remaining)) {
     lap->lap_flags = 0;
     if (lap->lap_type == LAP_TYPE_PARITY) {
       sh = SH_FORM_LAP(lap);
       unlock_stripe(sh);
       stripe_cache_release(sh);
       return;
     }
     lap->lap_caller_ops->ap_completion(..);
  }
}
static int osc_send_oap_rpc(...)
{
   ...
   struct laid_update_group *lug = NULL;
   ...
   list_for_each_safe(pos, tmp, &lop->lop_pending) {
       ...
       if(oap->oap_count <= 0) {
         CDEBUG(D_CACHE, ...);
         osc_ap_completion(...);
```

```
            continue;
        }
        if (oap->oap_async_flags & ASYNC_STRIPE_HEANDLE) {
            if (!lug)
                  init_lug(&lug);

            /*To wrirte for parity stripe unit or read in degraded
             *case, we gather the stripe_heads need to updating
             *into @lug*/
            rc = ops->ap_handle_stripe(oap->oap_caller_data,
                 cmd, 0, &lug);
            /*in read case, if the rc == 1, that means we get the
             *data on client from cache or XORing with other cache
             *stripe unit data. */
            if (cmd = OBD_BRW_READ && rc == 1) {
                 osc_ap_completion(...);
                 continue;
            }
        }
        /*now put the page bach in our accounting*/
        ...
        spin_unlock(&cli->cl_loi_list_lock);
        if (lug != NULL && lug->lug_pending > 0) {
             /*trigger the batched updating for parity*/
             rc = ops->ap_trigger_update(oap, cmd, 0, lug);
             /*read in degraed case*/
             if (!rc && cmd == OBD_BRW_READ) {
               /*call osc_ap_completion to finish the read
                 in degraded case*/
             }
        }
        request = osc_build_req(cli, ...);
        ...
}
```

In *lov_ap_queue_stripe*, we will gather stripe_heads upcall from parity _async_page or data _async_page in the case of degraded read into lug for latter batched preread uncached stripe units and parity updating.

```
int lov_ap_handle_stripe(data, cmd, flags, lug_in)
{
   struct lov_async_page *lap = LAP_FROM_COOKIE(data);
   struct lov_stripe_md *lsm;
   struct stripe_head *sh;
   struct stripe_unit *su;
   struct ld_private_data *ld;
```

```
struct laid_update_group *lug;
struct lov_oinfo *loi;
struct lov_obd *lov;
int stripeno;
sh = SH_FROM_LAP(lap);
lsm = sh->sh_lsm;
LASSERT(PATTERN_PARITY(lsm));

ld = LD_FROM_LSM(lsm);
lov = &(ld->ld_exp->exp_obd->u.lov);
stripeno = lap->lap_first_stripe;
su = &sh->sh_unit[stripeno];
lug = *lug_in;
if (lug == NULL) {
    rc = lug_init(&lug, lsm, cmd, stripeno);
    if (rc)
        return rc;
    *lug_in = lug;
}
if (cmd == OBD_BRW_READ) {
   LASSERT(lap->lap_type == LAP_TYPE_DATA);
   /*the old data is in the cache*/
   if (test_bit(SU_UPDATE, &su->su_flags)) {
       copy_data(su->su_page, su->su_cache);
       return 1;
   }
   /*it has cached all other stripe units*/
   if (sh->sh_nrcached == lsm->lsm_stripe_count - 1) {
        ld_reconstruct_unit(sh, stripeno);
        return 1;
   }
   /*FIXME: in case of LAID15 (LAID1+LAID5, mirror LAID5),
    *we must get the sub osc_async_page's stripeno,
    *not the stripe number of the first
    *mirror: lap_first_stripe. */
   loi = &lsm->lsm_oinfo[stripeno];
   ost = loi->loi_ost_idx;
   /*read in degraded case*/
   if (lov->tgts[ost].active == 0) {
       lug_add(sh);
   }
   return 0;
}
/*write case*/
if (lap->lap_type == LAP_TYPE_DATA) {
    /*lock the stripe_head here*/
```

```
        lug_add(lug, sh);
        return 0;
    }
    /*LAP_TYPE_parity, do nothing for parity
      asynchronous page*/
    return 0;
}
void lug_add(lug, sh)
{
     int rcw = 0, rmw = 0, dirtys = 0;
     ...
     lock_stripe(sh);
     list_add_tail(&sh->sh_locked_item, &lug->lug_locked_list);

     /*we need to preread some old data to reconstruct
      *the failed stripe.*/
     if (lug->lug_cmd == OBD_BRW_READ) {
        int idx = lug->lug_stripeno;
        for (i = 0; i < lsm->lsm_stripe_count; i++) {
           struct stripe_unit *su;

           su = &sh->sh_unit[i];
           if (!test_bit(SU_UPDATE, &su->su_flags))
                set_bit(SU_PREREAD, &su->su_flags);
       }
       lug->lug_pending++;
       return;
     }

      /*scan all stripe units and collect the information*/
     for (i = 0; i < lsm->lsm_stripe_count; i++) {
           struct stripe_unit *su;

           su = &sh->sh_unit[i];
           if (i == sh->sh_psidx) {
               if (!test_bit(SU_UPDATE, &su->su_flags))
                      rmw++;
                continue;
           }
           if (test_bit(SU_DIRTY, &su->su_flags))
                dirtys++;
           if (!test_bit(SU_DIRTY, &su->su_flags) &&
               !test_bit(SU_UPDATE, &su->su_flags)) {
                set_bit(SU_PREREAD, &su->su_flags);
                rcw++;
           }
```

```
    if (test_bit(SU_DIRTY, &su->su_flags) &&
        !test_bit(SU_UPDATE, &su->su_flags))
         rmw++;
}

/*full write covered the stripe row, we don't add
 * it to the update list until sync the last dirty page*/
if (dirtys == lsm->lsm_stripe_count - 1 ) {
     LASSERT(rcw == 0);
     set_bit(SH_FULL_WRITE, &sh->sh_flags);
     su = &sh->sh_unit[lug->lug_stripeno];
     copy_data(su->su_cache, su->su_page);
     set_bit(SU_UPDATE, &su->su_flags);
     return;
}

/*we want to update parity via method RCW, and
 *It is the last dirty page and it is being sync */
if (test_bit(SH_FULL_WRITE, &sh->sh_flags)) {
     su = &sh->sh_unit[lug->lug_stripeno];
     copy_data(su->cache, su->su_page);
     set_bit(SU_UPDATE, &su->su_flags);
     if (dirtys == 1) {
        lug->lug_pending++;
        list_add_tail(&sh->sh_item, &lug->lug_update_list);
        list_add_tail(&sh->sh_ready_item, &lug->lug_rcw_list);
     }
     return;
}
lug->lug_pending++;
if (rmw == 0) {
     list_add_tail(&sh->sh_item, &lug->lug_update_list);
     list_add_tail(&sh->sh_ready_item, &lug->lug_rmw_list);
     retrun;
}
/*set SU_PREREAD flag for the stripe unit need to preread,
  and we prefer to the method RCW*/
for (i = 0; i < lsm->lsm_stripe_count; i++) {
    struct stripe_unit *su;

    su = &sh->sh_unit[i];
    if ((rmw < rcw) && i == sh->sh_psidx) {
        if (!test_bit(SU_UPDATE, &su->su_flags)){
            set_bit(SU_PREREAD, &su->su_flags);
            lug->lug_info[i]++;
        }
```

```
                continue;
            }
            if ((rmw >= rcw) &&
                (!test_bit(SU_DIRTY, &su->su_flags) &&
                !test_bit(SU_UPDATE, &su->su_flags)) {
                 set_bit(SU_PREREAD, &su->su_flags);
                   lug->lug_info[i]++;
            }
            if ((rmw < rcw) && test_bit(SU_DIRTY, &su->su_flags) &&
                !test_bit(SU_UPDATE, &su->su_flags)) {
                 set_bit(SU_PREREAD, &su->su_flags);
                 lug->lug_info[i]++;
            }
        }
        list_add_tail(&sh->sh_item, &lug->lug_preread_list);
}
```

In *lov_ap_trigger_update*, we will start to preread stripe unit data uncached, and then do batch parity updating.

```
int lov_ap_trigger_update(data, cmd, flags, lug)
{
    ...

    LASSERT(lug->lug_pending > 0);
    rc = lov_stripe_preread(..., lug, ...);
    if (!rc) {
       ld_preread_to_ready(lsm, lug);
       /*in ld_update_lug, we will do batched parity updating
        *in case of parity write, and batched data reconstructing
        *in case of read in degraded case*/
       rc = ld_update_lug(lsm,lug);
    }
    /* queue asynchronous page for parity*/
    lov_queue_parity_io(..., lug, ...);
    /*unlock the stripe in lug->lug_locked_list*/
    ld_unlock_stripes(lug);
    REUTRN(rc);
}
```

In *lov_queue_parity_io*, we queue all parity in the lug->lug_update_list into *loi->loi_parity_lap*.

3.6 Preread uncached stripe unit data

After gather the stripe_heads need to update, we will do batched preread via synchronous way.

```
int lov_prep_preread_set(
    exp,lsm, src_oa, lug, oti, reqset)
{
   struct {
     obd_count count;
     obd_count off;
     struct brw_page *pga;
   } *info = NULL;
   struct list_head *tmp, *pos;
   ...
   OBD_ALLOC(info, sizeof(*info) * lsm->lsm_stripe_count);
   if (info == NULL)
      GOTO(out, rc = -ENOMEM);

   for (i = 0, loi = lsm->lsm_oinfo; i <
    lsm->lsm_stripe_count; i++, loi++) {
      struct lov_request *req;

      if(lug->lug_info[i] == 0)
        continue;
      ...
      OBD_ALLOC(req, sizeof(*req));
      ...
      info[i].count = lug->lug_info[i];
      req->rq_oabufs = info[i].count;
      /*alloc memory for member field pga of lov_request , we
       * wil free it in finish function*/
      OBD_ALLOC(req->rq_pag, (info[i].count) *
            sizeof(*req->rq_pga));
      ...
      info[i].pga = req->rq_pga;
      lov_set_add_req(req, set);
   }
   ...
   /*fill the brw_page array*/
   list_for_each_safe(pos,tmp,&lug->lug_list) {
      sh = list_entry(pos, struct stripe_head, sh_item);
      for (i = 0; i < lsm->lsm_stripe_count; i++) {
        struct stripe_unit *su;
        struct brw_page *pga = info[i].pga;

        su = &sh->sh_unit[i];
        if (test_bit(SU_PREREAD, &su->su_flags)) {
          int n = info[i].off ++;
          pga[n].pg = su->su_cache;
          pga[n].off = sh->sh_idx << PAGE_CACHE_SHIFT;
```

```
          pga[n].count = PAGE_SIZE;
        }
     }
     ...
}
static lov_stripe_preread
  (exp, lsm, src_oa, lug, oti)
{
  ...
  if(list_empty(&lug->lug_preread_list))
     RETURN(0);
  rc = lov_prep_preread_set(exp,lsm, NULL, lug, oti, &set);
  if (rc)
     RETURN(rc);
  list_for_each(pos, &set->set_list) {
    struct obd_export *sub_exp;
    req = list_entry(pos, struct lov_request, rq_link);

    sub_exp = lov->lov_tgts[req->rq_idx].ltd_exp;
    /*FIXME: It'd better implement via obd_brw_async*/
    rc = obd_brw(OBD_BRW_READ, sub_exp, req->ra_oa,
      req->rq_md, req->ra_oabufs, req->ra_pga, oti);
    if (rc)
        break;
    ...
  }
  ...
```

3.7 Callback locking

Our lock extent is stripe-group-size granularity. when flush page cache for the lock extent as it canceled, It also flushes the parity stripe cache in the lock extent.

```
void ll_pgcache_remove_extent(...)
{
  ...
  end = tmpex.l_extent.end >> PAGE_CACHE_SHIFT;
  if (lsm->lsm_stripe_count > 1) {
     int stripe_count = lsm->lsm_stripe_count;
     switch(lsm->lsm_pattern) {
     case LOV_PATTERN_RAID01:
        LASSERT(lsm->lsm_stripe_count % 2 == 0);
        stripe_count /= 2;
        stripe %= stripe_count;
     case LOV_PATTERN_RAID0:
```

```
         count = lsm->lsm_stripe_size >> PAGE_CACHE_SHIFT;
         skip = (stripe_count - 1) *count;
         start += start/count * skip +stripe *count;
         if (edn != ~0)
            end += end/count *skip +stripe *count;
         break;
      case LOV_PATTERN_RAID1:
         break;
      case LOV_PATTERN_RAID3:
       /*data layout
        *  D  D  D  P
        *  D  D  D  P
        *  D  D  D  P
        */
       LASSERT(lsm->lsm_stripe_count >= 3);
       if (stripe == lsm->lsm_stripe_count - 1)
           /*parity object*/
           GOTO(out, rc);
       /*similar with LAID0*/
       skip = (stripe_count - 2) *count;
       start += start/count *skip +stripe *count;
       if (end != ~0)
         end += end/count*skip +stripe*count;
       break;
case LOV_PATTERN_RAID5: {
      /*data layout
       * D  D  D  P  stripe group
       * D  D  P  D  stripe group
       * D  P  D  P  ...
       * P  D  D  D  ...
       */
     /*As in case of RAID5, the skip is not a fixed value,
      * It make the code very complex. If we use the master object
      * strategy and grant extent lock with stripe-group-size
      * granularity in HLD, it will make thing more simple as the skip
      * is zero.*/
      LASSERT(stripe_count > 3);
      LASSERT(start % count == 0);
      skip = 0;
      start = start * (stripe_count - 1);
      if (end != ~0) {
         LASSERT(end % count == 0);
         end = end * (stripe_count - 1);
      }
      break;
default:
```

```
    ...
}
...
*batching writeback under the lock explicitly.*/
for (i = start, j=start %count; i <= end;
 ...
}
out:
if (PATTERN_PARITY(lsm)) {
   struct ll_sb_info *sbi = ll_i2sbi(inode);
   rc = obd_remove_stripe_cache(sbi->ll_osc_exp, lsm,
       lock->l_policy_data.l_extent.start,
       lock->l_policy_data.l_extent.end);
}
}

static int lov_remover_stripe_cache
    (exp, lsm, start, end)
{
   ...
   ld = LD_FROM_LSM(lsm);
   start_idx = start >> STRIPE_CACHE_SHIFT;
   end_idx = end >> STRIPE_CACHE_SHIFT;
   for (i = start; i <= end, i++) {
      struct stripe_head *sh;
      if (!cache_has_stripes(ld))
         break;

      sh = find_get_stripe(ld, i);
      if (sh == NULL)
         continue;
      lock_stripe(sh);
      if (StripeDirty(sh)) {
         LASSERT(test_bit(SH_QUEUED, &sh->sh_flags));
         rc = lov_set_async_flags(exp, lsm, NULL,
            sh->sh_pslap, ASYNC_READY|ASYNC_URGENT);
         if (rc)
            CERROR("write stripe error!");
         lock_stripe(sh);
     }
     unlock_stripe(sh);
     stripe_cache_release(sh);
   }
   EXIT;
}
```

3.8 Truncate implementation

```
void ll_truncate(struct inode *inode)
{
   ...
   int shrink = 0, rc;
   __u64 orig_size;
   ...
   lov_stripe_lock(lsm);
   orig_size = lov_merge_size(lsm, 0);
   if (orig_size == inode->i_size) {
       ...
       lov_stripe_unlock(lsm);
       GOTO(out_unlock, 0);
   }
   if (orig_size > inode->i_size)
        shrink = 1;
   obd_adjust_kms(...);
   ...
   obdo_from_inode(...);
   if (shrink)
       obd_truncate_stripe_cache(...);
   ll_inode_size_unlock(inode, 0);

   if (shrink)
       rc = obd_truncate_update(exp,oa,lsm, inode->i_size, orig_size,NULL);
   rc = obd_punch(...);
   ...
}
```

3.8.1 Remove stripe cache in the truncate range

```
static int
lov_truncate_stripe_cache(
     struct obd_export *exp, struct lov_sripe_md *lsm,
     obd_off start_off, obd_off end_off, struct obd_trans_info *oti)
{
    unsigned long ssize = lsm->lsm_stripe_size;
    unsigned long swidth = ssize * (lsm->lsm_stripe_count - 1);
    unsigned long start;
    struct shvec shvec;
    struct ld_private_date *ld;
    int rc;

    if (!PATTERN_PARITY(lsm))
           RETURN(0);
```

```
    LASSERT(end_off == OBD_OBJECT_EOF);

    stripe_off = do_div(start_off, swidth);
    /*partial truncate without alignment with
     *stripe-group-size*/

    /*left size full fill the first data stipe
     *of the unaligned stripe group*/
    if (stripe_off >= ssize) {
        stripe_off = ssize;
    }
    ld = LD_FROM_LSM(lsm);
    /*get the start index in the object*/
    start =(start_off * ssize + stripe_off + STRIPE_SIZE - 1) / STRIPE_SIZE;
    shvec_init(&shvec, 0);
    next = start;
    /*Before truncate the stripe cache, the corresponding file cache
     *pages of all data stripe units have already truncated. So at this
     *we just need to teardown the _async_page of parity queued.*/
    while(shvec_lookup(&shvec, ld, next, SHVEC_SIZE)) {
        for(i = 0; i < shvec_count(&shvec); i++) {
           struct stripe_head *sh = shvec.stripes[i];
           unsigned long sh_index = sh->sh_index;

           if(sh_index > next)
                 next = sh_index;
           next++;
           lock_stripe(sh);
           LASSERT(test_bit(SH_QUEUED, &sh->sh_flags));
           rc = obd_teardown_async_page(...,sh->sh_pslap,...);
           sh->sh_pslap = NULL;
           set_bit(SH_FREE_AP, &sh->sh_flags);
           unlock_stripe(sh);
        }
        shvec_release(&shvec);
    }
}
```

3.9 Update parity for unaligned truncate

For truncate without stripe-group-size alignment, we will update the parity in aynchrounous way.

```
static int lov_truncate_update(
       exp, oa, lsm, new_size, orig_size, oti)
```

```
{
   unsigned long ssize = lsm->lsm_stripe_size;
   unsigned long swidth = ssize * (lsm->lsm_stripe_count - 1);
   unsigned long start, count;
   struct ld_private_date *ld;
   struct laid_update_group *lug;
   int i, rc;

   if (!PATTERN_PARITY(lsm))
        RETURN(0);

   if (new_size % swidth == 0)
        RETURN(0);

   /*If new_size and orig_size are all in the first data stripe,
    *needn't do parity updating*/
   if (orig_size - new_size < ssize &&
       orig_size % swidth < ssize && new_size % swidth < ssize)
        RETURN(0);

   init_lug(&lug);

   count = ssize / STRIPE_SZIE;
   start = (new_size / swidth) * ssize / STRIPE_SIZE;
   /*Scan stripe_heads in the last stripe_group, collect
    *the stripe_heads need to update into the lug.*/
   for (i = 0; i < count; i++, start++) {
      int j;
      struct stripe_head *sh;
      sh = grab_cache_stripe(i,..);

      /*check whether this stripe_head container stripe
       *units whose size is in the rang [new_size, old_size]
       * after map back onto the file. zero out the
       *stripe units' data in this rang, and mark them as
       *SU_UPDATE. Notice: Need special process for the partial stripe
       *unit in truncate rang.*/
      if (!need_update(sh, new_size, old_size)) {
           stripe_cache_release(sh);
           continue;
      }
      lug_add(lug, sh);
   }
   lov_stripe_preread(...lug..);
   ld_preread_to_update(lsm, lug);
   ld_update_lug(lsm, lug);
```

```
    /*write the updated parity to OSS.*/
    lov_brw_parity(...lug...);
}
```

3.10 File I/O in degraded case

Before do file I/O, we must first check whether it would succeed via method *lov_brw_check*. In RAID5 case, only when there are two OSTs in the stripe row occured failure, the check reports error. The IO process in degraed case is shown as follow:

- Degraded read: In case of LAID5/LAID3, every *lov_async_page* for read passes down to OSC layer will be marked as ASYNC_GRACE_ERROR. In OSC layer, we don't care about whether the OST is validate or active, just do as normal to queue async/group IO, trigger the group IO, etc. In *osc_send_oap_rpc*, we will queue the IO to failed OST and upcall to LOV layer via *.ap_handle_stripe* . After that, via *.ap_trigger_updatge* do batched preread and data reconstructing.
- Degraded write: Just need to update the corresponding stripe units in LOV layer, and grace the error of OSS failure.

3.11 Synchronal IO (DIRECT_IO)

In direct_IO, we first calls *lov_r5_prep_brw_set* to perpare for IO request set, the process is similar with *lov_prep_preread_set*. For write request , first call *lov_preread* to read necessary old stripe unit data from OSSs , and then update the parity, make the write request to OSSs; For read request in degraded case, we first add the preread stripe units to the request set, and then make read request to OSSs, in the completion handler *lov_brw_interpret*, we reconstruct the read data.

3.12 Daemon thread for syncing updated parity

Once update the parity, we will queue the asynchronous page of parity into the parity of the object. To make the parity sync to OST ASAP, we create a daemon thread *laid5d* to sync parity to OST. Every time, queue the batched group of parity into corresponding loi->loi_parity_lop, we will wake up this thread to sync the parity.

4 State management

4.1 *stripe_unit* state

- SU_INIT: This stripe unit has already initialized.
- SU_UPDATE: The *cache* contains the vaild data.

- SU_DIRTY: The file page associated with this stripe unit is dirtied, the parity stripe unit needs to update.

4.2 *stripe_head* state

- SH_INIT: The stripe_head has already initialized.
- SH_GRANT_SPACE: The parity *_async_page* has already grant the cache space.
- SH_QUEUED: The *lap* for the parity is already queued.
- SH_DIRTY: This stripe_heads need to do parity updating.
- SH_FULL_WRITE: It is large write covered the stripe row, the parity is updated in method RCW.
- SH_FREE_AP: It has already teardown the *_async_page* for parity(sh->sh_pslap).

5 Focus for inspections

- Is this a reasonable way to implement LAID5/LAID3?
- Is there any leak?
- Is it right that divide the sync into two parts: data and parity which are maintained by loi->loi_write_lop and loi->loi_parity_lop respectively? or the parity is also governed by the loi->loi_write_lop?
- Is it reasonable that do preread operation in synchronous way? or should we add a new obd stacked on OSC to manage the preread and sync of data and parity?
- Is there any performance problem?
- Should we cache the stripe cache in direct_IO? cache until the lock callbacking? Any suggestion?