Detailed Level Design Specification for Index Api Module

Danilov Nikita <nikita@clusterfs.com>

2006.02.25–2006.06.15

Contents

1 Introduction

iam functionality is described in iam hld, which see for reference. This document assumes familiarity with ext3/htree implementation, and tree search/update algorithms.

2 Functional Specification

2.1 General description

iam provides an abstraction of transactional persistent container, where user-supplied records, identified by keys, are stored and retrieved. Implementation is based on ext3/htree: iam container structure is built on top of the underlying file (either regular of directory), divided into block-sized *nodes*. Nodes are addressed by their logical offset within file. This means, that iam employs two-level indexing: first, node address is obtained, then this address is resolved into physical storage block number through file-system mechanism, e.g., direct/indirect/double-indirect pointers in case of ext3.

Different iam containers vary in the attributes of keys and records they support:

- key size;
- record size and whether variable sized records are supported;
- whether records with duplicate keys are supported.

Internally, iam implementation introduces additional abstraction layer, referred to as a *container format*. Format defines on-disk layout of tree nodes and provides low-level access methods to manipulate node contents. On top of format interface generic iam code implements complex container operations (lookup, insertion, iteration) that are used by the iam clients. Formats are necessary, because one of iam goals is the support of containers with different on-disk layouts, specifically, legacy ext3/htree containers.

Container format together with other container attributes mentioned above is termed a *container description* (or *container parameters*). Container description is a complete set of data necessary to manipulate container.

As in ext3/htree, nodes in iam container are of two different flavors:

- index nodes. Index nodes contain pointers to other nodes;
- leaf nodes (leaves). Leaves contain actual records.

Index nodes have regular structure: a header, followed by an array of *index entries*. Index entry is a (key, address) pair, describing sub-tree rooted at the child of this index node. Key is the least key in this sub-tree, and address is logical block number of the child node. Index entries are sorted in the key order. The format of the header may be different for the root of the tree and for other index nodes. This is done to make tree self-contained: its root header contains full description of tree format, while headers for non-root index nodes contain only necessary information, like number of entries.

While keys and, therefore, index entries can be of different size in different containers, overall structure of index nodes is the same for all containers. Technically speaking, this means that index nodes are manipulated by the generic iam code rather than by the container format. The only part of index node that is manipulated through format interface is node header.

Leaf nodes, on the other hand, have quite different structure for different containers: for ext3/htree leaves are filled with ext2 compatible directory entries, while for object index and fld containers leaves are arrays of fixed size records. Providing uniform interface to access leaf nodes is main function of format interface.

Following formats are currently provided:

- iam_htree: this format is binary compatible with ext3/htree directory. By using this format, ext3/htree directories can be manipulated through iam interface (both read and write access).

- iam_lfix: a format for container with fixed size records (and fixed size keys). This is to be used for object index, fld, and other data-base-like indices.

iam module provides two interfaces: "simple" interface and "cursor" interface. Simple interface allows its users to perform most common index operations: lookup, insert, delete and update of a record with a given key. This interface should be sufficient for majority of iam clients (object index, fld). Cursor interface is based on a notion of *iterator* (cursor). Iterator is a particular position in a container. In the first approximation, iterator is just a key, identifying some record in the container. In fact, some containers support duplicate records with the same key, so, technically, speaking, iterator is some sort of pointer to a particular record within particular leaf node. Iterator can be positioned to the first record with a given key, and them moved forward one record at a time. Clients of cursor interface inspect key and record that are currently "under" cursor, and may update record contents, or insert/delete records, subject to key ordering constraints. One final refinement of the concept of iterator is necessary: in fact iterator is not a pointer to a record, but record insertion point. The difference between these two notions is almost non-existent except for few subtle corner cases:

- usually we can identify insertion point for a record with the record after which new record will be inserted. This works most of the time, except for the case when insertion has to be done *before* first record in the node. This curious insertion mode is unusual and is not present in the normal B-trees, but it does happen in htrees sometimes;

- another htree-specific complication arise from the possibility that insertion point falls into empty leaf node (this is possible, because in htree a range of keys assigned to the leaf node never changes once assigned).

These two exceptions are handled through special iterator state, see `enum iam_it_state` below.

Cursor interface is more complex than simple interface, but provides wider access to the container internals (e.g., containers with duplicate keys can be properly accessed only through cursor interface). Internally simple interface is implemented on top of cursor interface.

All invariants below are specified under assumption of no concurrent container updates.

2.2 Format-specific data

This subsection describes one particular important optimization implemented by iam. As this optimization affects client-visible interfaces it is described in Functional Specification.

Sometimes during tree manipulations generic iam code has to store temporary values of some keys. Typical example is a node split, where value of pivot key has to be remembered to be later inserted into parent index node. On the other hand, key size depends on the particular container, so necessary temporary storage cannot be statically allocated on the stack. To avoid an overhead and scalability constraints of dynamic memory allocation in the hot paths, a notion of *description-specific area* is introduced (see above for the definition of container description). Description-specific area is a structure containing pointers to the preallocated keys (and possibly some other fields in the future), to be used as a temporary storage by iam generic code. Pointer to description-specific area is passed by clients to the iam entry-points. Idea behind this is that upper level client knows actual size of keys supported by container in advance . For example, fld module assumes that keys in its iam container are fid sequence numbers (64 bits wide), and object index knows that its container uses fids as keys. Due to this, caller can preallocate keys efficiently (either on the stack, or, preferably, through lu_context_key interface).

```
/*
 * description-specific part of iam_path. This is usually embedded into larger
 * structure.
 */
struct iam_path_descr {
        /*
         * Scratch-pad area for temporary keys.
         */
        struct iam_key        *ipd_key_scratch[DX_SCRATCH_KEYS];
};
```

2.3 General container types and functions

2.3.1 type definitions

```
/*
```

```
 * Incomplete type used to refer to keys in iam container.
 *
 * As key size can be different from container to container, iam has to use
 * incomplete type. Clients cast pointer to iam_key to real key type and back.
 */
struct iam_key;
/* Incomplete type use to refer to the records stored in iam containers. */
struct iam_rec;
/*
 * An instance of iam container.
 */
struct iam_container;
```

2.3.2 prototypes

```
/*
 * Initialize container @c.
 */
int iam_container_init(struct iam_container *c,
                       struct iam_descr *descr, struct inode *inode);
/*
 * Finalize container @c, release all resources.
 */
void iam_container_fini(struct iam_container *c);
/*
 * Determine container format.
 */
int iam_container_setup(struct iam_container *c);
```

2.4 Simple iam interface

```
/*
 * Search container @c for record with key @k. If record is found, its data
 * are moved into @r.
 *
 * Return values: +ve: found, 0: not-found, -ve: error
 */
int iam_lookup(struct iam_container *c, const struct iam_key *k,
               struct iam_rec *r, struct iam_path_descr *pd);
/*
 * Insert new record @r with key @k into container @c (within context of
 * transaction @h.
 *
 * Return values: 0: success, -ve: error, including -EEXIST when record with
 * given key is already present.
 *
 * postcondition: ergo(result == 0 || result == -EEXIST,
 *                                   iam_lookup(c, k, r2) > 0 &&
 *                                   !memcmp(r, r2, c->ic_descr->id_rec_size));
 */
```

```
int iam_insert(handle_t *h, struct iam_container *c, const struct iam_key *k,
               struct iam_rec *r, struct iam_path_descr *pd);
/*
 * Update record with the key @k in container @c (within context of
 * transaction @h), new record is given by @r.
 *
 * Return values: 0: success, -ve: error, including -ENOENT if no record with
 * the given key found.
 */
int iam_update(handle_t *h, struct iam_container *c, const struct iam_key *k,
               struct iam_rec *r, struct iam_path_descr *pd);
/*
 * Delete existing record with key @k.
 *
 * Return values: 0: success, -ENOENT: not-found, -ve: other error.
 *
 * postcondition: ergo(result == 0 || result == -ENOENT,
 *                                  !iam_lookup(c, k, *));
 */
int iam_delete(handle_t *h, struct iam_container *c, const struct iam_key *k,
               struct iam_path_descr *pd)
```

2.5 Cursor (iterator) interface

2.5.1 data types

```
/*
 * iam cursor (iterator) api.
 */
/*
 * Iterator.
 *
 * Immediately after call to iam_it_init() iterator is in "detached"
 * (IAM_IT_DETACHED) state: it is associated with given parent container, but
 * doesn't point to any particular record in this container.
 *
 * After successful call to iam_it_get() and until corresponding call to
 * iam_it_put() iterator is in one of "active" states: IAM_IT_ATTACHED or
 * IAM_IT_SKEWED.
 *
 * Active iterator can move through records in a container (provided
 * IAM_IT_MOVE permission) in a key order, can get record and key values as it
 * passes over them, and can modify container (provided IAM_IT_WRITE
 * permission).
 *
 * Iteration may reach the end of container, at which point iterator switches
 * into IAM_IT_DETACHED state.
 *
 * Concurrency: iterators are supposed to be local to thread. Interfaces below
 * do no internal serialization of access to the iterator fields.
```

```
 *
 * When in non-detached state, iterator keeps some container nodes pinned in
 * memory and locked (that locking may be implemented at the container
 * granularity though). In particular, clients may assume that pointers to
 * records and keys obtained through iterator interface as valid until
 * iterator is detached (except that they may be invalidated by sub-sequent
 * operations done through the same iterator).
 *
 */
struct iam_iterator {
        /*
         * iterator flags, taken from enum iam_it_flags.
         */
        __u32                ii_flags;
        enum iam_it_state    ii_state;
        /*
         * path to the record. Valid in IAM_IT_ATTACHED state.
         */
        struct iam_path      ii_path;
};
/*
 * States of iterator state machine.
 */
enum iam_it_state {
        /* initial state */
        IAM_IT_DETACHED,
        /* iterator is above particular record in the container */
        IAM_IT_ATTACHED,
        /* iterator is positioned before record  */
        IAM_IT_SKEWED
};
/*
 * Flags controlling iterator functionality.
 */
enum iam_it_flags {
        /*
         * this iterator will move (iam_it_next() will be called on it)
         */
        IAM_IT_MOVE  = (1 << 0),
        /*
         * tree can be updated through this iterator.
         */
        IAM_IT_WRITE = (1 << 1)
};
```

2.5.2 function prototypes

```
/*
 * Initialize iterator to IAM_IT_DETACHED state.
 *
```

```
 * postcondition: it_state(it) == IAM_IT_DETACHED
 */
int  iam_it_init(struct iam_iterator *it, struct iam_container *c, __u32 flags,
                 struct iam_path_descr *pd);
/*
 * Finalize iterator and release all resources.
 *
 * precondition: it_state(it) == IAM_IT_DETACHED
 */
void iam_it_fini(struct iam_iterator *it);
```

Attaching/detaching iterator:

```
/*
 * Attach iterator. After successful completion, @it points to record with
 * least key not larger than @k.
 *
 * Return value: 0: positioned on existing record,
 *             -ve: error.
 *
 * precondition:  it_state(it) == IAM_IT_DETACHED
 * postcondition: ergo(result == 0 && it_state(it) == IAM_IT_ATTACHED,
 *                     it_keycmp(it, iam_it_key_get(it, *), k) <= 0)
 */
int iam_it_get(struct iam_iterator *it, const struct iam_key *k);
/*
 * Attach iterator, and assure it points to the record (not skewed).
 *
 * Return value: 0: positioned on existing record,
 *             -ve: error.
 *
 * precondition:  it_state(it) == IAM_IT_DETACHED &&
 *                !(it->ii_flags&IAM_IT_WRITE)
 * postcondition: ergo(result == 0, it_state(it) == IAM_IT_ATTACHED)
 */
int iam_it_get_at(struct iam_iterator *it, const struct iam_key *k);
/*
 * Detach iterator. Does nothing it detached state.
 *
 * postcondition: it_state(it) == IAM_IT_DETACHED
 */
void iam_it_put(struct iam_iterator *it);
```

Iteration:

```
/*
 * Move iterator one record right.
 *
 * Return value: 0: success,
 *              +1: end of container reached
 *             -ve: error
```

```
 *
 * precondition:  (it_state(it) == IAM_IT_ATTACHED ||
 *                 it_state(it) == IAM_IT_SKEWED) && it->ii_flags&IAM_IT_MOVE
 * postcondition: ergo(result == 0, it_state(it) == IAM_IT_ATTACHED)
 */
int iam_it_next(struct iam_iterator *it);
```

Access to record and key:

```
/*
 * Return pointer to the record under iterator.
 *
 * precondition:  it_state(it) == IAM_IT_ATTACHED && it_at_rec(it)
 * postcondition: it_state(it) == IAM_IT_ATTACHED
 */
struct iam_rec *iam_it_rec_get(const struct iam_iterator *it);
/*
 * Replace contents of record under iterator.
 *
 * precondition:  it_state(it) == IAM_IT_ATTACHED &&
 *                it->ii_flags&IAM_IT_WRITE
 * postcondition: it_state(it) == IAM_IT_ATTACHED &&
 *                ergo(result == 0, !memcmp(iam_it_rec_get(it), r, ...))
 */
int iam_it_rec_set(handle_t *h, struct iam_iterator *it, struct iam_rec *r);
/*
 * Return pointer to the key under iterator.
 *
 * precondition:  it_state(it) == IAM_IT_ATTACHED
 * postcondition: it_state(it) == IAM_IT_ATTACHED
 */
struct iam_key *iam_it_key_get(const struct iam_iterator *it,
                               struct iam_key *k);
```

Container modification (record insertion and deletion):

```
/*
 * Insert new record with key @k and contents from @r, shifting records to the
 * right.
 *
 * precondition: it->ii_flags&IAM_IT_WRITE &&
 *               (it_state(it) == IAM_IT_ATTACHED ||
 *                it_state(it) == IAM_IT_SKEWED) &&
 *               ergo(it_state(it) == IAM_IT_ATTACHED,
 *                    it_keycmp(it, iam_it_key_get(it, it_scratch_key(it, 0)),
 *                              k) < 0) &&
 *               ergo(it_before(it),
 *                    it_keycmp(it, iam_it_key_get(it, it_scratch_key(it, 0)),
 *                              k) > 0));
 * postcondition: ergo(result == 0,
 *                     it_state(it) == IAM_IT_ATTACHED &&
```

```
 *                     it_keycmp(it, iam_it_key_get(it, *), k) == 0 &&
 *                     !memcmp(iam_it_rec_get(it), r, ...))
 */
int iam_it_rec_insert(handle_t *h, struct iam_iterator *it,
                      const struct iam_key *k, const struct iam_rec *r);
/*
 * Delete record under iterator.
 *
 * precondition:  it_state(it) == IAM_IT_ATTACHED &&
 *                it->ii_flags&IAM_IT_WRITE &&
 *                it_at_rec(it)
 * postcondition: it_state(it) == IAM_IT_ATTACHED ||
 *                it_state(it) == IAM_IT_DETACHED
 */
int iam_it_rec_delete(handle_t *h, struct iam_iterator *it);
```

Iterator serialization. These functions are exported to allow restartable iteration (like readdir), by first converting iterator into opaque scalar cookie of type `iam_pos_t` (64 bits currently) and later resuming iteration from cookie.

```
typedef __u64 iam_pos_t;
/*
 * Convert iterator to cookie.
 *
 * precondition:  it_state(it) == IAM_IT_ATTACHED &&
 *                iam_path_descr(it->ii_path)->id_key_size <= sizeof(iam_pos_t)
 * postcondition: it_state(it) == IAM_IT_ATTACHED
 */
iam_pos_t iam_it_store(const struct iam_iterator *it);
/*
 * Restore iterator from cookie.
 *
 * precondition:  it_state(it) == IAM_IT_DETACHED &&
 *                it->ii_flags&IAM_IT_MOVE &&
 *                iam_path_descr(it->ii_path)->id_key_size <= sizeof(iam_pos_t)
 * postcondition: ergo(result == 0, it_state(it) == IAM_IT_ATTACHED &&
 *                                  iam_it_store(it) == pos)
 */
int iam_it_load(struct iam_iterator *it, iam_pos_t pos);
```

2.6 Format interface

Format is represented by `struct iam_format`:

```
/*
 * Container format.
 */
struct iam_format {
        /*
         * Method called to recognize container format. Should return true iff
         * container @c conforms to this format. This method may do IO to read
```

```
             * container pages.
             *
             * If container is recognized, this method sets operation vectors
             * ->id_ops and ->id_leaf_ops in container description (c->ic_descr),
             * and fills other description fields.
             */
            int (*if_guess)(struct iam_container *c);
            /*
             * Linkage into global list of container formats.
             */
            struct list_head if_linkage;
};
```

Format implementation registers format by calling

```
void iam_format_register(struct iam_format *fmt);
```

This function adds format into global list of all registered formats. Generic code works with formats by calling

```
/*
 * Determine format of a given container. This is done by scanning list of
 * registered formats and calling ->if_guess() method of each in turn.
 */
int iam_format_guess(struct iam_container *c);
```

If container format is recognized, this function returns success, and, according to the definition of `->if_guess()` method, all container description fields (`->id_key_size`, `->id_rec_size`, `->id_ptr_size`, `->id_ops`, `->id_leaf_ops`, etc.) are filled with correct values. After that point, generic code interacts with format implementation by calling `->id_ops` and `->id_leaf_ops` methods from container description.

2.7 Container creation

iam containers are created by special user-level utility (lustre/lustre/utils/create_iam) that takes format name ("lfix" or "htree" currently) and a set of format-specific parameters (e.g., key size and record size). create_iam creates container that contains single record (whose content is irrelevant) with the smallest possible key.

3 Use Cases

Warning: use cases below are somewhat artificial, because both object index and fld will, in fact, use iam facilities through osd interface, rather than directly. They are provided here as iam usage logic is still the same, only divided differently between layers. Interaction between osd and iam will be covered in osd design specifications.

3.1 osd object index (ooi)

ooi maps fids local for the given osd into storage cookies.

```
struct iam_descr ooi_descr;
struct ooi_info {
        struct iam_container oi_container;
};
struct osd_dev {
        ...
        struct ooi_info ooi;
        ...
};
struct osd_thread_info {
        ...
        /*
         * Description-specific data preallocated for object index iam.
         */
        struct iam_path_descr ipd;
        ...
};
int ooi_init(struct osd_dev *osd)
{
        int result;
        struct iam_container *bag;
        struct file *ooi_file;
        ooi_file = filp_open("/ooi", O_RDWR, S_IRWXU);
        /* sanity and security checks... */
        bag = &osd->ooi.oi_container;
        result = iam_container_init(bag, &ooi_descr,
                                    ooi_file->f_dentry->d_inode);
        if (result == 0) {
                result = iam_container_setup(bag);
                if (bag->ic_descr->id_key_size != sizeof(struct lu_fid) ||
                    bag->ic_descr->id_rec_size != sizeof(struct storage_cookie)) {
                        CERROR("Wrong key or record size in ooi iam file!\n");
                        result = -EINVAL;
                }
        }
        return result;
}
void ooi_fini(struct osd_dev *osd)
{
        iam_container_fini(&osd->ooi.oi_container);
}
int ooi_lookup(struct osd_thread_info *info, struct ost_dev *osd,
               const struct lustre_fid *fid, struct storage_cookie *cookie)
{
        return iam_lookup(&osd->ooi.oi_container,
                          (const struct iam_key *)fid, (struct iam_rec *)cookie,
                          &info->ipd);
```

```
}
int ooi_insert(struct osd_thread_info *info, struct ost_dev *osd, handle_t *txn,
               const struct lustre_fid *fid, const struct storage_cookie *cookie)
{
        return iam_insert(txn, &osd->ooi.oi_container,
                          (const struct iam_key *)fid,
                          (const struct iam_rec *)cookie,
                          &info->ipd);
}
```

Cursor api usage.

```
/*
 * Possible example of iam iterator interface usage.
 *
 * Scan all fids from given sequence and call actor function on each found
 * fid.
 */
int fid_scan_seq(struct osd_thread_info *info, struct osd_dev *osd, __u64 seq,
                 int (*actor)(struct osd_dev *,
                              struct lustre_fid *, struct storage_cookie *))
{
        int result;
        struct iam_iterator it;
        struct lu_fid key0 = {
                .f_seq     = cpu_to_be64(seq),
                .f_id      = 0,
                .f_version = 0
        };
        result = iam_it_init(&it, &osd->ooi.oi_container, IAM_IT_MOVE,
                             &info->ipd);
        if (result != 0)
                return result;
        /*
         * position cursor above first fid in the given sequence. If any.
         */
        result = iam_it_get_at(&it, &key0);
        if (result < 0)
                return result;
        /*
         * break out of loop when end of container is reached, or on error.
         */
        while (result == 0) {
                struct lu_fid  fid;
                struct lu_fid *key;
                assert(it.ii_state == IAM_IT_ATTACHED);
                key = (struct lu_fid *)iam_it_key_get(&it, (void *)&fid);
                assert(be32_to_cpu(key->f_seq) >= seq);
                if (be32_to_cpu(key->f_seq) != seq)
                        /*
```

```
                         * end of sequence reached
                         */
                        break;
                result = actor(osd, key, (void *)iam_it_rec_get(&it));
                if (result != 0)
                        break;
                result = iam_it_next(&it);
        }
        iam_it_put(&it);
        iam_it_fini(&it);
        return result;
}
```

3.2 fid location database (fld)

Mostly the same as ooi. Fid sequences are used as keys. mds number is a record:

```
struct iam_descr fld_descr;
struct fld_info {
        struct iam_container fi_container;
};
struct mdd_obd {
        ...
        struct fld_info fld;
        ...
};
struct mdd_thread_info {
        ...
        struct iam_path_descr ipd;
        ...
};
int fld_init(struct mdd_obd *mdd)
{
        int result;
        struct file *fld_file;
        struct iam_container *bag;
        fld_file = filp_open("/fld", O_RDWR, S_IRWXU);
        /* sanity and security checks... */
        bag = &mdd->fld.fi_container;
        result = iam_container_init(bag, &fld_descr,
                                    fld_file->f_dentry->d_inode);
        if (result == 0) {
                result = iam_container_setup(bag);
                if (bag->ic_descr->id_key_size != sizeof(fid_seq_t) ||
                    bag->ic_descr->id_rec_size != sizeof(mds_no_t)) {
                        CERROR("Wrong key or record size in fld iam file!\n");
                        result = -EINVAL;
                }
        }
        return result;
```

```
}
void fld_fini(struct mdd_obd *mdd)
{
        iam_container_fini(&mdd->fld.fi_container);
}
mdsno_t fld_lookup(struct mdd_obd *mdd, __u64 seq)
{
        msdno_t mdsno;
        int result;
        seq = cpu_to_be64(seq);
        result = iam_lookup(&mdd->fld.fi_container,
                            (const struct iam_key *)&seq, (struct iam_rec *)&mdsno);
        if (result == 0)
                return -ENOENT;
        else if (result > 0)
                return be64_to_cpu(mdsno);
        else
                return result;
}
int fld_insert(struct mdd_obd *mdd, handle_t *txn, __u64 seq, mdsno_t mdsno)
{
        seq = cpu_to_be64(seq);
        mdsno = cpu_to_be64(mdsno);
        return iam_insert(txn, &mdd->fld.fi_container,
                          (const struct iam_key *)&seq, (struct iam_rec *)&mdsno);
}
```

3.3 Unit tests

General notes about unit tests:

- iam code has special debugging compilation mode in which many internal invariants for in-memory and persistent data structures are checked. Tests should be run with this mode on.
- where appropriate, tests should be run in compatibility mode to check that resulting tree created by iam is recognized by ext3 and vise versa.
- tests that have "prepare" and "check" phases should be run in two modes:
  - continuous: prepare and check are done in one run;
  - separate: prepare and check are separated by umount.

3.3.1 populate container with N records with known keys and delete.

insert N records with unique keys generated according to given formula. Use iam_lookup() to check that all records were in fact inserted. Use iterator to check that no other records exist. Then delete some of keys. Check (through lookup and iterator) that tree contains expected records only.

3.3.2 non-unique populate and delete

insert N*M records with N unique keys in total (so that every key is repeated M times). Check correctness. Check that lookup always returns first record among records with unique keys. Then delete some keys through iam_delete(). Check that first records were deleted.

3.3.3 degenerate populate and delete

insert M records all with the same key. Check with iterator that records are returned in creation order. Delete some of records, check correctness.

3.3.4 fence conditions

Check how appropriate simple container methods and cursor methods work in border conditions:

- empty container;
- cursor positioned beyond last record.

3.3.5 tall trees

insert enough records to build a tree taller than K index levels.

3.3.6 concurrency control

run appropriate unit tests with multiple worker threads.

3.3.7 different block size support

Run all other unit tests with all supported block sizes.

4 Logic Specification

4.1 General container data-types

```
/*
 * Scalar type into which certain iam_key's can be uniquely mapped. Used to
 * support interfaces like readdir(), where iteration over index has to be
 * re-startable.
 */
typedef __u64 iam_ptr_t;
/*
 * Index node traversed during tree lookup.
 */
struct iam_frame {
        struct buffer_head *bh;    /* buffer holding node data */
        struct iam_entry *entries; /* array of entries */
        struct iam_entry *at;      /* target entry, found by binary search */
};
struct iam_path;
```

```
/* leaf node reached by tree lookup */
struct iam_leaf {
        struct iam_path    *il_path;
        struct buffer_head *il_bh;
        struct iam_lentry  *il_entries;
        struct iam_lentry  *il_at;
        void               *il_descr_data;
};
/*
 * Return values of ->lookup() operation from struct iam_leaf_operations.
 */
enum iam_lookup_t {
        /*
         * lookup positioned leaf on some record
         */
        IAM_LOOKUP_OK,
        /*
         * leaf was empty
         */
        IAM_LOOKUP_EMPTY,
        /*
         * lookup positioned leaf before first record
         */
        IAM_LOOKUP_BEFORE
};
/*
 * Format-specific container operations. These are called by generic iam code.
 */
struct iam_operations {
        /*
         * Returns pointer (in the same sense as pointer in index entry) to
         * the root node.
         */
        __u32 (*id_root_ptr)(struct iam_container *c);
        /*
         * Check validity and consistency of index node.
         */
        int (*id_node_check)(struct iam_path *path, struct iam_frame *frame);
        /*
         * Copy some data from node header into frame. This is called when
         * new node is loaded into frame.
         */
        int (*id_node_load)(struct iam_path *path, struct iam_frame *frame);
        /*
         * Initialize new node (stored in @bh) that is going to be added into
         * tree.
         */
        int (*id_node_init)(struct iam_container *c,
                            struct buffer_head *bh, int root);
        int (*id_node_read)(struct iam_container *c, iam_ptr_t ptr,
```

```
                                handle_t *h, struct buffer_head **bh);
        /*
         * Key comparison function. Returns -1, 0, +1.
         */
        int (*id_keycmp)(const struct iam_container *c,
                         const struct iam_key *k1, const struct iam_key *k2);
        /*
         * Create new container.
         *
         * Newly created container has a root node and a single leaf. Leaf
         * contains single record with the least possible key.
         */
        int (*id_create)(struct iam_container *c);
        /*
         * Modify root node when tree height increases.
         */
        void (*id_root_inc)(struct iam_container *c, struct iam_frame *frame);
        /*
         * Format name.
         */
        char id_name[DX_FMT_NAME_LEN];
};
/*
 * Another format-specific operation vector, consisting of methods to access
 * leaf nodes. This is separated from struct iam_operations, because it is
 * assumed that there will be many formats with different format of leaf
 * nodes, yes the same struct iam_operations.
 */
struct iam_leaf_operations {
        /*
         * leaf operations.
         */
        /*
         * initialize just loaded leaf node.
         */
        int (*init)(struct iam_leaf *p);
        /*
         * Format new node.
         */
        void (*init_new)(struct iam_container *c, struct buffer_head *bh);
        /*
         * Release resources.
         */
        void (*fini)(struct iam_leaf *l);
        /*
         * returns true iff leaf is positioned at the last entry.
         */
        int (*at_end)(const struct iam_leaf *l);
        /* position leaf at the first entry */
        void (*start)(struct iam_leaf *l);
```

```
        /* more leaf to the next entry. */
        void (*next)(struct iam_leaf *l);
        /* return key of current leaf record. This method may return
         * either pointer to the key stored in node, or copy key into
         * @k buffer supplied by caller and return pointer to this
         * buffer. The latter approach is used when keys in nodes are
         * not stored in plain form (e.g., htree doesn't store keys at
         * all).
         *
         * Caller should assume that returned pointer is only valid
         * while leaf node is pinned and locked.*/
        struct iam_key *(*key)(const struct iam_leaf *l, struct iam_key *k);
        /* return pointer to entry body. Pointer is valid while
           corresponding leaf node is locked and pinned. */
        struct iam_rec *(*rec)(const struct iam_leaf *l);
        void (*key_set)(struct iam_leaf *l, const struct iam_key *k);
        void (*rec_set)(struct iam_leaf *l, const struct iam_rec *r);
        /*
         * Search leaf @l for a record with key @k or for a place
         * where such record is to be inserted.
         *
         * Scratch keys from @path can be used.
         */
        int (*lookup)(struct iam_leaf *l, const struct iam_key *k);
        int (*can_add)(const struct iam_leaf *l,
                       const struct iam_key *k, const struct iam_rec *r);
        /*
         * add rec for a leaf
         */
        void (*rec_add)(struct iam_leaf *l,
                        const struct iam_key *k, const struct iam_rec *r);
        /*
         * remove rec for a leaf
         */
        void (*rec_del)(struct iam_leaf *l);
        /*
         * split leaf node, moving some entries into @bh (the latter currently
         * is assumed to be empty).
         */
        void (*split)(struct iam_leaf *l, struct buffer_head **bh,
                      iam_ptr_t newblknr);
};
/*
 * Parameters, describing a flavor of iam container.
 */
struct iam_descr {
        /*
         * Size of a key in this container, in bytes.
         */
        size_t       id_key_size;
```

```
        /*
         * Size of a pointer to the next level (stored in index nodes), in
         * bytes.
         */
        size_t       id_ptr_size;
        /*
         * Size of a record (stored in leaf nodes), in bytes.
         */
        size_t       id_rec_size;
        /*
         * Size of unused (by iam) space at the beginning of every non-root
         * node, in bytes. Used for compatibility with ext3.
         */
        size_t       id_node_gap;
        /*
         * Size of unused (by iam) space at the beginning of root node, in
         * bytes. Used for compatibility with ext3.
         */
        size_t       id_root_gap;
        struct iam_operations           *id_ops;
        struct iam_leaf_operations      *id_leaf_ops;
};
/*
 * An instance of iam container.
 */
struct iam_container {
        /*
         * Underlying flat file. IO against this object is issued to
         * read/write nodes.
         */
        struct inode     *ic_object;
        /*
         * container flavor.
         */
        struct iam_descr *ic_descr;
};
/*
 * Structure to keep track of a path drilled through htree.
 */
struct iam_path {
        /*
         * Parent container.
         */
        struct iam_container  *ip_container;
        /*
         * Number of index levels minus one.
         */
        int                    ip_indirect;
        /*
         * Nodes that top-to-bottom traversal passed through.
```

```
         */
        struct iam_frame       ip_frames[DX_MAX_TREE_HEIGHT];
        /*
         * Last filled frame in ->ip_frames. Refers to the 'twig' node (one
         * immediately above leaf).
         */
        struct iam_frame      *ip_frame;
        /*
         * Leaf node: a child of ->ip_frame.
         */
        struct iam_leaf        ip_leaf;
        /*
         * Key searched for.
         */
        const struct iam_key  *ip_key_target;
        /*
         * Description-specific data.
         */
        struct iam_path_descr *ip_data;
};
```

4.2 Index node structure

```
/*
 * iam: big theory statement.
 *
 * iam (Index Access Module) is a module providing abstraction of persistent
 * transactional container on top of generalized ext3 htree.
 *
 * iam supports:
 *
 *     - key, pointer, and record size specifiable per container.
 *
 *     - trees taller than 2 index levels.
 *
 *     - read/write to existing ext3 htree directories as iam containers.
 *
 * iam container is a tree, consisting of leaf nodes containing keys and
 * records stored in this container, and index nodes, containing keys and
 * pointers to leaf or index nodes.
 *
 * iam does not work with keys directly, instead it calls user-supplied key
 * comparison function (->dpo_keycmp()).
 *
 * Pointers are (currently) interpreted as logical offsets (measured in
 * blocksful) within underlying flat file on top of which iam tree lives.
 *
 * On-disk format:
 *
 * iam mostly tries to reuse existing htree formats.
```

```
 *
 * Format of index node:
 *
 * +-----+-------+-------+-------+------+-------+------------+
 * |     | count |       |       |      |       |            |
 * | gap |   /   | entry | entry | .... | entry | free space |
 * |     | limit |       |       |      |       |            |
 * +-----+-------+-------+-------+------+-------+------------+
 *
 *       gap           this part of node is never accessed by iam code. It
 *                     exists for binary compatibility with ext3 htree (that,
 *                     in turn, stores fake struct ext2_dirent for ext2
 *                     compatibility), and to keep some unspecified per-node
 *                     data. Gap can be different for root and non-root index
 *                     nodes. Gap size can be specified for each container
 *                     (gap of 0 is allowed).
 *
 *       count/limit   current number of entries in this node, and the maximal
 *                     number of entries that can fit into node. count/limit
 *                     has the same size as entry, and is itself counted in
 *                     count.
 *
 *       entry         index entry: consists of a key immediately followed by
 *                     a pointer to a child node. Size of a key and size of a
 *                     pointer depends on container. Entry has neither
 *                     alignment nor padding.
 *
 *       free space    portion of node new entries are added to
 *
 * Entries in index node are sorted by their key value.
 *
 */
```

There are no data-types for index node entry, because its size is variable. Instead we declare incomplete type (forward reference):

```
/*
 * Entry within index tree node. Consists of a key immediately followed
 * (without padding) by a pointer to the child node.
 *
 * Both key and pointer are of variable size, hence incomplete type.
 */
struct iam_entry;
```

Actual definition of `struct iam_entry` is not provided, so no variables of type `struct iam_entry` can be instantiated. Value of `struct iam_entry` is that pointers to it are used in iam interfaces, providing type-checking. This device is used extensively throughout iam code, compare with `struct iam_key` and `struct iam_record` above.

4.3 Simple iam interface

Simple iam interface is implemented on top of iterator interface:

```
/*
 * Helper wrapper around iam_it_get(): returns 0 (success) only when record
 * with exactly the same key as asked is found.
 */
static int iam_it_get_exact(struct iam_iterator *it, const struct iam_key *k) {
        result = iam_it_get(it, k);
        if (result == 0 &&
            (it_state(it) != IAM_IT_ATTACHED ||
             it_keycmp(it, k, iam_it_key_get(it, it_scratch_key(it, 1))) != 0))
                /*
                 * Return -ENOENT if cursor is located above record with a key
                 * different from one specified, or in the empty leaf.
                 *
                 * XXX returning -ENOENT only works if iam_it_get never
                 * returns -ENOENT as a legitimate error.
                 */
                result = -ENOENT;
        return result;
}
/*
 * Search container @c for record with key @k. If record is found, its data
 * are moved into @r.
 *
 * Return values: +ve: found, 0: not-found, -ve: error
 */
int iam_lookup(struct iam_container *c, const struct iam_key *k,
               struct iam_rec *r, struct iam_path_descr *pd) {
        struct iam_iterator it;
        iam_it_init(&it, c, 0, pd);
        result = iam_it_get_exact(&it, k);
        if (result == 0)
                /*
                 * record with required key found, copy it into user buffer
                 */
                iam_reccpy(&it.ii_path, r, iam_it_rec_get(&it));
        iam_it_put(&it);
        iam_it_fini(&it);
        return result;
}
/*
 * Insert new record @r with key @k into container @c (within context of
 * transaction @h.
 *
 * Return values: 0: success, -ve: error, including -EEXIST when record with
 * given key is already present.
 *
 * postcondition: ergo(result == 0 || result == -EEXIST,
```

```
 *                                  iam_lookup(c, k, r2) > 0 &&
 *                                  !memcmp(r, r2, c->ic_descr->id_rec_size));
 */
int iam_insert(handle_t *h, struct iam_container *c, const struct iam_key *k,
               struct iam_rec *r, struct iam_path_descr *pd) {
        struct iam_iterator it;
        iam_it_init(&it, c, IAM_IT_WRITE, pd);
        result = iam_it_get_exact(&it, k);
        if (result == -ENOENT)
                result = iam_it_rec_insert(h, &it, k, r);
        else if (result == 0)
                result = -EEXIST;
        iam_it_put(&it);
        iam_it_fini(&it);
        return result;
}
/*
 * Update record with the key @k in container @c (within context of
 * transaction @h), new record is given by @r.
 *
 * Return values: 0: success, -ve: error, including -ENOENT if no record with
 * the given key found.
 */
int iam_update(handle_t *h, struct iam_container *c, const struct iam_key *k,
               struct iam_rec *r, struct iam_path_descr *pd) {
        struct iam_iterator it;
        iam_it_init(&it, c, IAM_IT_WRITE, pd);
        result = iam_it_get_exact(&it, k);
        if (result == 0)
                iam_it_rec_set(h, &it, r);
        iam_it_put(&it);
        iam_it_fini(&it);
        return result;
}
/*
 * Delete existing record with key @k.
 *
 * Return values: 0: success, -ENOENT: not-found, -ve: other error.
 *
 * postcondition: ergo(result == 0 || result == -ENOENT,
 *                                  !iam_lookup(c, k, *));
 */
int iam_delete(handle_t *h, struct iam_container *c, const struct iam_key *k,
               struct iam_path_descr *pd) {
        struct iam_iterator it;
        iam_it_init(&it, c, IAM_IT_WRITE, pd);
        result = iam_it_get_exact(&it, k);
        if (result == 0)
                iam_it_rec_delete(h, &it);
        iam_it_put(&it);
```

```
        iam_it_fini(&it);
        return result;
}
```

4.4 Iterator interface

```
static enum iam_it_state it_state(const struct iam_iterator *it) {
        return it->ii_state;
}
/*
 * Helper function returning scratch key.
 */
static struct iam_key *it_scratch_key(const struct iam_iterator *it, int n) {
        return iam_path_key(&it->ii_path, n);
}
static struct iam_container *iam_it_container(const struct iam_iterator *it) {
        return it->ii_path.ip_container;
}
static inline int it_keycmp(const struct iam_iterator *it,
                            const struct iam_key *k1, const struct iam_key *k2) {
        return iam_keycmp(iam_it_container(it), k1, k2);
}
static inline int it_at_rec(const struct iam_iterator *it) {
        return !iam_leaf_at_end(&it->ii_path.ip_leaf);
}
static inline int it_before(const struct iam_iterator *it) {
        return it_state(it) == IAM_IT_SKEWED && it_at_rec(it);
}
void iam_container_write_lock(struct iam_container *ic) {
        down(&ic->ic_object->i_sem);
}
void iam_container_write_unlock(struct iam_container *ic) {
        up(&ic->ic_object->i_sem);
}
void iam_container_read_lock(struct iam_container *ic) {
        down(&ic->ic_object->i_sem);
}
void iam_container_read_unlock(struct iam_container *ic) {
        up(&ic->ic_object->i_sem);
}
static void iam_it_lock(struct iam_iterator *it) {
        if (it->ii_flags&IAM_IT_WRITE)
                iam_container_write_lock(iam_it_container(it));
        else
                iam_container_read_lock(iam_it_container(it));
}
static void iam_it_unlock(struct iam_iterator *it) {
        if (it->ii_flags&IAM_IT_WRITE)
                iam_container_write_unlock(iam_it_container(it));
```

```
        else
                iam_container_read_unlock(iam_it_container(it));
}
/*
 * Initialize iterator to IAM_IT_DETACHED state.
 *
 * postcondition: it_state(it) == IAM_IT_DETACHED
 */
int  iam_it_init(struct iam_iterator *it, struct iam_container *c, __u32 flags,
                 struct iam_path_descr *pd) {
        memset(it, 0, sizeof *it);
        it->ii_flags  = flags;
        it->ii_state  = IAM_IT_DETACHED;
        iam_path_init(&it->ii_path, c, pd);
        return 0;
}
/*
 * Finalize iterator and release all resources.
 *
 * precondition: it_state(it) == IAM_IT_DETACHED
 */
void iam_it_fini(struct iam_iterator *it) {
        iam_path_fini(&it->ii_path);
}
/*
 * Performs tree top-to-bottom traversal starting from root, and loads leaf
 * node.
 */
static int iam_path_lookup(struct iam_path *path) {
        struct iam_container *c;
        struct iam_descr *descr;
        struct iam_leaf  *leaf;

        c = path->ip_container;
        leaf = &path->ip_leaf;
        descr = iam_path_descr(path);
        result = dx_lookup(path);
        assert(iam_path_check(path));
        if (result == 0) {
                result = iam_leaf_load(path);
                assert(ergo(result == 0, iam_leaf_check(leaf)));
                if (result == 0)
                        result = iam_leaf_ops(leaf)->
                                lookup(leaf, path->ip_key_target);
        }
        return result;
}
/*
 * Attach iterator. After successful completion, @it points to record with
 * least key not larger than @k.
```

```
 *
 * Return value: 0: positioned on existing record,
 *             -ve: error.
 *
 * precondition:  it_state(it) == IAM_IT_DETACHED
 * postcondition: ergo(result == 0 && it_state(it) == IAM_IT_ATTACHED,
 *                     it_keycmp(it, iam_it_key_get(it, *), k) <= 0)
 */
int iam_it_get(struct iam_iterator *it, const struct iam_key *k) {
        it->ii_path.ip_key_target = k;
        iam_it_lock(it);
        result = iam_path_lookup(&it->ii_path);
        if (result >= 0) {
                switch (result) {
                case IAM_LOOKUP_OK:
                        it->ii_state = IAM_IT_ATTACHED;
                        break;
                case IAM_LOOKUP_BEFORE:
                case IAM_LOOKUP_EMPTY:
                        it->ii_state = IAM_IT_SKEWED;
                        break;
                }
                result = 0;
        } else
                iam_it_unlock(it);
}
/*
 * Attach iterator, and assure it points to the record (not skewed).
 *
 * Return value: 0: positioned on existing record,
 *             -ve: error.
 *
 * precondition:  it_state(it) == IAM_IT_DETACHED &&
 *                !(it->ii_flags&IAM_IT_WRITE)
 * postcondition: ergo(result == 0, it_state(it) == IAM_IT_ATTACHED)
 */
int iam_it_get_at(struct iam_iterator *it, const struct iam_key *k) {
        result = iam_it_get(it, k);
        if (result == 0) {
                if (it_state(it) != IAM_IT_ATTACHED) {
                        assert(it_state(it) == IAM_IT_SKEWED);
                        result = iam_it_next(it);
                }
        }
        return result;
}
/*
 * Detach iterator. Does nothing it detached state.
 *
 * postcondition: it_state(it) == IAM_IT_DETACHED
```

```
 */
void iam_it_put(struct iam_iterator *it) {
        if (it->ii_state != IAM_IT_DETACHED) {
                it->ii_state = IAM_IT_DETACHED;
                iam_leaf_fini(&it->ii_path.ip_leaf);
                iam_it_unlock(it);
        }
}
/*
 * Move iterator one record right.
 *
 * Return value: 0: success,
 *               +1: end of container reached
 *             -ve: error
 *
 * precondition:  (it_state(it) == IAM_IT_ATTACHED ||
 *                 it_state(it) == IAM_IT_SKEWED) && it->ii_flags&IAM_IT_MOVE
 * postcondition: ergo(result == 0, it_state(it) == IAM_IT_ATTACHED) &&
 *                ergo(result >  0, it_state(it) == IAM_IT_DETACHED)
 */
int iam_it_next(struct iam_iterator *it) {
        struct iam_path      *path;
        struct iam_leaf      *leaf;
        path = &it->ii_path;
        leaf = &path->ip_leaf;
        result = 0;
        if (it_before(it)) {
                it->ii_state = IAM_IT_ATTACHED;
        } else {
                if (!iam_leaf_at_end(leaf))
                        /* advance within leaf node */
                        iam_leaf_next(leaf);
                /*
                 * multiple iterations may be necessary due to empty leaves.
                 */
                while (result == 0 && iam_leaf_at_end(leaf)) {
                        /* advance index portion of the path */
                        result = iam_index_next(iam_it_container(it), path);
                        if (result == 1) {
                                iam_leaf_fini(leaf);
                                result = iam_leaf_load(path);
                                if (result == 0)
                                        iam_leaf_start(leaf);
                        } else if (result == 0)
                                /* end of container reached */
                                result = +1;
                        if (result != 0)
                                iam_it_put(it);
                }
        }
```

```
        return result;
}
/*
 * Return pointer to the record under iterator.
 *
 * precondition:  it_state(it) == IAM_IT_ATTACHED && it_at_rec(it)
 * postcondition: it_state(it) == IAM_IT_ATTACHED
 */
struct iam_rec *iam_it_rec_get(const struct iam_iterator *it) {
        return iam_leaf_rec(&it->ii_path.ip_leaf);
}
static void iam_it_reccpy(struct iam_iterator *it, const struct iam_rec *r) {
        struct iam_leaf *folio;
        folio = &it->ii_path.ip_leaf;
        iam_leaf_ops(folio)->rec_set(folio, r);
}
static void iam_it_keycpy(struct iam_iterator *it, const struct iam_key *k) {
        struct iam_leaf *folio;
        folio = &it->ii_path.ip_leaf;
        iam_leaf_ops(folio)->key_set(folio, k);
}
/*
 * Replace contents of record under iterator.
 *
 * precondition:  it_state(it) == IAM_IT_ATTACHED &&
 *                it->ii_flags&IAM_IT_WRITE
 * postcondition: it_state(it) == IAM_IT_ATTACHED &&
 *                ergo(result == 0, !memcmp(iam_it_rec_get(it), r, ...))
 */
int iam_it_rec_set(handle_t *h, struct iam_iterator *it, struct iam_rec *r) {
        result = iam_txn_add(h, path, bh);
        if (result == 0) {
                iam_it_reccpy(it, r);
                result = iam_txn_dirty(handle, path, bh);
        }
        return result;
}
/*
 * Return pointer to the key under iterator.
 *
 * precondition:  it_state(it) == IAM_IT_ATTACHED ||
 *                it_state(it) == IAM_IT_SKEWED
 * postcondition: it_state(it) == IAM_IT_ATTACHED
 */
static struct iam_key *iam_it_key_get(const struct iam_iterator *it,
                                      struct iam_key *k) {
        return iam_leaf_key(&it->ii_path.ip_leaf, k);
}
static int iam_new_leaf(handle_t *handle, struct iam_leaf *leaf) {
        u32 blknr; /* XXX 32bit block size */
```

```
        struct buffer_head   *new_leaf;
        struct iam_container *c;
        struct inode         *obj;
        c = iam_leaf_container(leaf);
        obj = c->ic_object;
        new_leaf = ext3_append(handle, c->ic_object, &blknr, &err);
        if (new_leaf != NULL) {
                iam_leaf_ops(leaf)->init_new(c, new_leaf);
                iam_leaf_split(leaf, &new_leaf, blknr);
                err = iam_txn_dirty(handle, iam_leaf_path(leaf), new_leaf);
                brelse(new_leaf);
                if (err == 0)
                        err = ext3_mark_inode_dirty(handle, c->ic_object);
        }
        return err;
}
static int iam_add_rec(handle_t *handle, struct iam_path *path,
                       const struct iam_key *k, const struct iam_rec *r) {
        struct iam_leaf *leaf;
        leaf = &path->ip_leaf;
        err = iam_txn_add(handle, path, leaf->il_bh);
        if (err == 0) {
                if (!iam_leaf_can_add(leaf, k, r)) {
                        err = split_index_node(handle, path);
                        if (err == 0) {
                                err = iam_new_leaf(handle, leaf);
                                /*
                                 * refresh @leaf, as split may retarget path
                                 * to the new leaf node.
                                 */
                                leaf = &path->ip_leaf;
                                if (err == 0)
                                        err = iam_txn_dirty(handle, path,
                                                            path->ip_frame->bh);
                        }
                }
                if (err == 0) {
                        iam_leaf_rec_add(leaf, k, r);
                        err = iam_txn_dirty(handle, path, leaf->il_bh);
                }
        }
        return err;
}
/*
 * Insert new record with key @k and contents from @r, shifting records to the
 * right.
 *
 * precondition: it->ii_flags&IAM_IT_WRITE &&
 *               (it_state(it) == IAM_IT_ATTACHED ||
 *                it_state(it) == IAM_IT_SKEWED) &&
```

```
 *               ergo(it_state(it) == IAM_IT_ATTACHED,
 *                    it_keycmp(it, iam_it_key_get(it, it_scratch_key(it, 0)),
 *                              k) < 0) &&
 *               ergo(it_before(it),
 *                    it_keycmp(it, __iam_it_key_get(it, it_scratch_key(it, 0)),
 *                              k) > 0));
 * postcondition: ergo(result == 0,
 *                     it_state(it) == IAM_IT_ATTACHED &&
 *                     it_keycmp(it, iam_it_key_get(it, *), k) == 0 &&
 *                     !memcmp(iam_it_rec_get(it), r, ...))
 */
int iam_it_rec_insert(handle_t *h, struct iam_iterator *it,
                      const struct iam_key *k, const struct iam_rec *r) {
        struct iam_path *path;
        path = &it->ii_path;
        result = iam_add_rec(h, path, k, r);
        if (result == 0)
                it->ii_state = IAM_IT_ATTACHED;
        return result;
}
/*
 * Delete record under iterator.
 *
 * precondition:  it_state(it) == IAM_IT_ATTACHED &&
 *                it->ii_flags&IAM_IT_WRITE &&
 *                it_at_rec(it)
 * postcondition: it_state(it) == IAM_IT_ATTACHED ||
 *                it_state(it) == IAM_IT_DETACHED
 */
int iam_it_rec_delete(handle_t *h, struct iam_iterator *it) {
        struct iam_leaf *leaf;
        struct iam_path *path;
        path = &it->ii_path;
        leaf = &path->ip_leaf;
        result = iam_txn_add(h, path, leaf->il_bh);
        /*
         * no compaction for now.
         */
        if (result == 0) {
                iam_rec_del(leaf);
                result = iam_txn_dirty(h, path, leaf->il_bh);
                if (result == 0 && iam_leaf_at_end(leaf) &&
                    it->ii_flags&IAM_IT_MOVE) {
                        result = iam_it_next(it);
                        if (result > 0)
                                result = 0;
                }
        }
        return result;
}
```

```
/*
 * Convert iterator to cookie.
 *
 * precondition:  it_state(it) == IAM_IT_ATTACHED &&
 *                iam_path_descr(it->ii_path)->id_key_size <= sizeof(iam_pos_t)
 * postcondition: it_state(it) == IAM_IT_ATTACHED
 */
iam_pos_t iam_it_store(const struct iam_iterator *it) {
        iam_pos_t result;
        result = 0;
        iam_it_key_get(it, (struct iam_key *)&result);
        return result;
}
/*
 * Restore iterator from cookie.
 *
 * precondition:  it_state(it) == IAM_IT_DETACHED && it->ii_flags&IAM_IT_MOVE &&
 *                iam_path_descr(it->ii_path)->id_key_size <= sizeof(iam_pos_t)
 * postcondition: ergo(result == 0, it_state(it) == IAM_IT_ATTACHED &&
 *                                  iam_it_store(it) == pos)
 */
int iam_it_load(struct iam_iterator *it, iam_pos_t pos) {
        return iam_it_get(it, (struct iam_key *)&pos);
}
```

4.5 Other generic code

This is code used by iterator implementation:

```
/*
 * Lookup through index part of the tree: descends from the root node, by
 * doing binary search at each level.
 */
int dx_lookup(struct iam_path *path) {
        u32 ptr;
        int err = 0;
        int i;
        struct iam_descr *param;
        struct iam_frame *frame;
        struct iam_container *c;
        param = iam_path_descr(path);
        c = path->ip_container;

        for (frame = path->ip_frames, i = 0,
                     ptr = param->id_ops->id_root_ptr(c);
             i <= path->ip_indirect;
             ptr = dx_get_block(path, frame->at), ++frame, ++i) {
                struct iam_entry *entries;
                struct iam_entry *p;
                struct iam_entry *q;
```

```
                struct iam_entry *m;
                unsigned count;
                err = param->id_ops->id_node_read(c, (iam_ptr_t)ptr, NULL,
                                                  &frame->bh);
                if (err != 0)
                        break;
                err = param->id_ops->id_node_check(path, frame);
                if (err != 0)
                        break;
                err = param->id_ops->id_node_load(path, frame);
                if (err != 0)
                        break;
                assert(dx_node_check(path, frame));
                entries = frame->entries;
                count = dx_get_count(entries);
                assert(count && count <= dx_get_limit(entries));
                p = iam_entry_shift(path, entries, 1);
                q = iam_entry_shift(path, entries, count - 1);
                /*
                 * Sanity check: target key is larger or equal to the leftmost
                 * key in the node.
                 */
                if (iam_keycmp(c,
                               iam_key_at(path, p), path->ip_key_target) > 0) {
                        struct inode *obj;
                        obj = c->ic_object;
                        ext3_error(obj->i_sb, __FUNCTION__,
                                   "corrupted search tree #%lu", obj->i_ino);
                        err = -EIO;
                        break;

                }
                while (p <= q) {
                        m = iam_entry_shift(path,
                                            p, iam_entry_diff(path, q, p) / 2);
                        dxtrace(printk("."));
                        if (iam_keycmp(c, iam_key_at(path, m),
                                       path->ip_key_target) > 0)
                                q = iam_entry_shift(path, m, -1);
                        else
                                p = iam_entry_shift(path, m, +1);
                }
                frame->at = iam_entry_shift(path, p, -1);
        }
        if (err != 0)
                iam_path_fini(path);
        path->ip_frame = --frame;
        return err;
}
/*
```

```
 * Split one or more index nodes to provide free space necessary to insert
 * pointer to the new leaf.
 */
int split_index_node(handle_t *handle, struct iam_path *path)
{
        struct iam_frame *frame, *safe;
        u32 newblock[DX_MAX_TREE_HEIGHT] = {0};
        struct inode *dir = iam_path_obj(path);
        struct iam_descr *descr;
        int nr_splet;
        int i, err;
        descr = iam_path_descr(path);
        /*
         * Algorithm below depends on this.
         */
        assert(descr->id_node_gap < descr->id_root_gap);
        frame = path->ip_frame;
        entries = frame->entries;
        /*
         * Tall-tree handling: we might have to split multiple index blocks
         * all the way up to tree root. Tricky point here is error handling:
         * to avoid complicated undo/rollback we
         *
         *   - first allocate all necessary blocks
         *
         *   - insert pointers into them atomically.
         *
         * XXX nikita: this algorithm is *not* scalable, as it assumes that at
         * least nodes in the path are locked.
         */
        /* What levels need split? */
        for (nr_splet = 0; frame >= path->ip_frames &&
             dx_get_count(frame->entries) == dx_get_limit(frame->entries);
             --frame, ++nr_splet) {
                if (nr_splet == DX_MAX_TREE_HEIGHT)
                        return -ENOSPC;
        }
        safe = frame;
        /* Go back down, allocating blocks, and adding blocks into
         * transaction... */
        for (frame = safe + 1, i = 0; i < nr_splet; ++i, ++frame)
                bh_new[i] = ext3_append (handle, dir, &newblock[i], &err);
        /* Go through nodes once more, inserting pointers */
        for (frame = safe + 1, i = 0; i < nr_splet; ++i, ++frame) {
                if (frame == path->ip_frames) {
                        /* splitting root node. Tricky point:
                         *
                         * In the "normal" B-tree we'd split root *and* add
                         * new root to the tree with pointers to the old root
                         * and its sibling (thus introducing two new nodes).
```

```
                         *
                         * In htree it's enough to add one node, because
                         * capacity of the root node is smaller than that of
                         * non-root one.
                         */
                        descr->id_ops->id_root_inc(path->ip_container, frame);
                        /* Shift frames in the path */
                        memmove(frames + 2, frames + 1,
                                (sizeof path->ip_frames) - 2 * sizeof frames[0]);
                        /* Add new access path frame */
                        frames[1].at = iam_entry_shift(path, entries2, idx);
                        frames[1].entries = entries = entries2;
                        frames[1].bh = bh2;
                } else {
                        /* splitting non-root index node. */
                        unsigned count1 = count/2, count2 = count - count1;
                        struct iam_key *pivot = iam_path_key(path, 3);
                        struct iam_frame *parent = frame - 1;
                        iam_get_key(path,
                                    iam_entry_shift(path, entries, count1),
                                    pivot);
                        memcpy ((char *) entries2,
                                (char *) iam_entry_shift(path, entries, count1),
                                count2 * iam_entry_size(path));
                        dx_set_count (entries, count1);
                        dx_set_count (entries2, count2);
                        dx_set_limit (entries2, dx_node_limit(path));
                        /* Which index block gets the new entry? */
                        if (idx >= count1) {
                                frame->at = iam_entry_shift(path, entries2,
                                                            idx - count1);
                                frame->entries = entries = entries2;
                                swap(frame->bh, bh2);
                                bh_new[i] = bh2;
                        }
                        iam_insert_key(path, parent, pivot, newblock[i]);
                }
        }
        if (nr_splet > 0)
                /*
                 * Log ->i_size modification.
                 */
                err = ext3_mark_inode_dirty(handle, dir);
        return err;
}
```

4.6 Format implementation

As an example, implementation of few methods of lfix container format is provided.

Data-structures defining node layout:

```
enum {
        IAM_LEAF_HEADER_MAGIC = 0x1976,
        IAM_LFIX_ROOT_MAGIC   = 0xbedabb1edULL
};
struct iam_leaf_head {
        __le16 ill_magic;
        __le16 ill_count;
};
struct iam_lfix_root {
        __le64  ilr_magic;
        __le16  ilr_keysize;
        __le16  ilr_recsize;
        __le16  ilr_ptrsize;
        __le16  ilr_indirect_levels;
};
static int lentry_count_get(const struct iam_leaf *leaf)
{
        return le16_to_cpu(iam_get_head(leaf)->ill_count);
}
static void lentry_count_set(struct iam_leaf *leaf, unsigned count)
{
        assert(0 <= count && count <= leaf_count_limit(leaf));
        iam_get_head(leaf)->ill_count = cpu_to_le16(count);
}
struct iam_key *iam_lfix_key(const struct iam_leaf *l, struct iam_key *key)
{
        void *ie = l->il_at;
        assert(iam_leaf_at_rec(l));
        return (struct iam_key*)ie;
}
static void iam_lfix_start(struct iam_leaf *l)
{
        l->il_at = iam_get_lentries(l);
}
static int iam_lfix_init(struct iam_leaf *l)
{
        int result;
        struct iam_leaf_head *ill;
        int count;
        assert(l->il_bh != NULL);
        ill = iam_get_head(l);
        count = le16_to_cpu(ill->ill_count);
        if (ill->ill_magic == le16_to_cpu(IAM_LEAF_HEADER_MAGIC) &&
            0 <= count && count <= leaf_count_limit(l)) {
                l->il_at = l->il_entries = iam_get_lentries(l);
                result = 0;
        } else {
                struct inode *obj;
```

```
                obj = iam_leaf_container(l)->ic_object;
                ext3_error(obj->i_sb, __FUNCTION__,
                           "Wrong magic in node %llu (#%lu): %#x != %#x or "
                           "wrong count: %i (%i)",
                           (unsigned long long)l->il_bh->b_blocknr, obj->i_ino,
                           ill->ill_magic, le16_to_cpu(IAM_LEAF_HEADER_MAGIC),
                           count, leaf_count_limit(l));
                result = -EIO;
        }
        return result;
}
static void iam_lfix_fini(struct iam_leaf *l)
{
        l->il_entries = l->il_at = NULL;
        return;
}
struct iam_rec *iam_lfix_rec(const struct iam_leaf *l)
{
        void *e = l->il_at;
        assert(iam_leaf_at_rec(l));
        return e + iam_leaf_descr(l)->id_key_size;
}
static void iam_lfix_next(struct iam_leaf *l)
{
        assert(iam_leaf_at_rec(l));
        l->il_at = iam_lfix_shift(l, l->il_at, 1);
}
static int iam_lfix_lookup(struct iam_leaf *l, const struct iam_key *k)
{
        struct iam_lentry *p, *q, *m, *t;
        struct iam_container *c;
        int count;
        int result;
        count = lentry_count_get(l);
        if (count == 0)
                return IAM_LOOKUP_EMPTY;
        result = IAM_LOOKUP_OK;
        c = iam_leaf_container(l);
        p = l->il_entries;
        q = iam_lfix_shift(l, p, count - 1);
        if (iam_keycmp(c, k, iam_leaf_key_at(p)) < 0) {
                /*
                 * @k is less than the least key in the leaf
                 */
                l->il_at = p;
                result = IAM_LOOKUP_BEFORE;
        } else if (iam_keycmp(c, iam_leaf_key_at(q), k) <= 0) {
                l->il_at = q;
        } else {
                /*
```

```
                 * EWD1293
                 */
                while (iam_lfix_shift(l, p, 1) != q) {
                        m = iam_lfix_shift(l, p, iam_lfix_diff(l, q, p) / 2);
                        assert(p < m && m < q);
                        (iam_keycmp(c, iam_leaf_key_at(m), k) <= 0 ? p : q) = m;
                }
                assert(iam_keycmp(c, iam_leaf_key_at(p), k) <= 0 &&
                       iam_keycmp(c, k, iam_leaf_key_at(q)) < 0);
                /*
                 * skip over records with duplicate keys.
                 */
                while (p > l->il_entries) {
                        t = iam_lfix_shift(l, p, -1);
                        if (iam_keycmp(c, iam_leaf_key_at(t), k) == 0)
                                p = t;
                        else
                                break;
                }
                l->il_at = p;
        }
        assert(iam_leaf_at_rec(l));
        return result;
}
static void iam_lfix_key_set(struct iam_leaf *l, const struct iam_key *k)
{
        assert(iam_leaf_at_rec(l));
        iam_keycpy(iam_leaf_container(l), iam_leaf_key_at(l->il_at), k);
}
static void iam_lfix_rec_set(struct iam_leaf *l, const struct iam_rec *r)
{
        assert(iam_leaf_at_rec(l));
        iam_reccpy(iam_leaf_path(l), iam_lfix_rec(l), r);
}
static void iam_lfix_rec_add(struct iam_leaf *leaf,
                             const struct iam_key *k, const struct iam_rec *r)
{
        struct iam_lentry *end;
        struct iam_lentry *cur;
        struct iam_lentry *start;
        ptrdiff_t diff;
        int count;
        assert(iam_leaf_can_add(leaf, k, r));
        count = lentry_count_get(leaf);
        /*
         * This branch handles two exceptional cases:
         *
         *   - leaf positioned beyond last record, and
         *
         *   - empty leaf.
```

```
                 */
                if (!iam_leaf_at_end(leaf)) {
                        end   = iam_lfix_get_end(leaf);
                        cur   = leaf->il_at;
                        if (iam_keycmp(iam_leaf_container(leaf),
                                       k, iam_leaf_key_at(cur)) >= 0)
                                iam_lfix_next(leaf);
                        else
                                /*
                                 * Another exceptional case: insertion with the key
                                 * less than smallest key in the leaf.
                                 */
                                assert(cur == leaf->il_entries);
                        start = leaf->il_at;
                        diff  = (void *)end - (void *)start;
                        assert(diff >= 0);
                        memmove(iam_lfix_shift(leaf, start, 1), start, diff);
                }
                lentry_count_set(leaf, count + 1);
                iam_lfix_key_set(leaf, k);
                iam_lfix_rec_set(leaf, r);
                assert(iam_leaf_at_rec(leaf));
        }
        static void iam_lfix_rec_del(struct iam_leaf *leaf)
        {
                struct iam_lentry *next, *end;
                int count;
                ptrdiff_t diff;
                assert(iam_leaf_at_rec(leaf));
                count = lentry_count_get(leaf);
                end = iam_lfix_get_end(leaf);
                next = iam_lfix_shift(leaf, leaf->il_at, 1);
                diff = (void *)end - (void *)next;
                memmove(leaf->il_at, next, diff);
                lentry_count_set(leaf, count - 1);
        }
        static int iam_lfix_can_add(const struct iam_leaf *l,
                                    const struct iam_key *k, const struct iam_rec *r)
        {
                return lentry_count_get(l) < leaf_count_limit(l);
        }
        static int iam_lfix_at_end(const struct iam_leaf *folio)
        {
                return folio->il_at == iam_lfix_get_end(folio);
        }
```

5 State Specification

5.1 Resources Involved and Their State

5.1.1 invariants of various data-types

```
static inline int ptr_inside(void *base, size_t size, void *ptr) {
        return (base <= ptr) && (ptr < base + size);
}
int iam_frame_invariant(struct iam_frame *f) {
        return
                (f->bh != NULL &&
                f->bh->b_data != NULL &&
                ptr_inside(f->bh->b_data, f->bh->b_size, f->entries) &&
                ptr_inside(f->bh->b_data, f->bh->b_size, f->at) &&
                f->entries <= f->at);
}
int iam_leaf_invariant(struct iam_leaf *l) {
        return
                l->il_bh != NULL &&
                l->il_bh->b_data != NULL &&
                ptr_inside(l->il_bh->b_data, l->il_bh->b_size, l->il_entries) &&
                ptr_inside(l->il_bh->b_data, l->il_bh->b_size, l->il_at) &&
                l->il_entries <= l->il_at;
}
int iam_path_invariant(struct iam_path *p) {
        int i;
        if (p->ip_container == NULL ||
            p->ip_indirect < 0 || p->ip_indirect > DX_MAX_TREE_HEIGHT - 1 ||
            p->ip_frame != p->ip_frames + p->ip_indirect ||
            !iam_leaf_invariant(&p->ip_leaf))
                return 0;
        for (i = 0; i < ARRAY_SIZE(p->ip_frames); ++i) {
                if (i <= p->ip_indirect) {
                        if (!iam_frame_invariant(&p->ip_frames[i]))
                                return 0;
                }
        }
        return 1;
}
int iam_it_invariant(struct iam_iterator *it) {
        return
                (it->ii_state == IAM_IT_DETACHED ||
                 it->ii_state == IAM_IT_ATTACHED ||
                 it->ii_state == IAM_IT_SKEWED) &&
                !(it->ii_flags & ~(IAM_IT_MOVE | IAM_IT_WRITE)) &&
                ergo(it->ii_state == IAM_IT_ATTACHED ||
                     it->ii_state == IAM_IT_SKEWED,
                     iam_path_invariant(&it->ii_path) &&
```

```
                        equi(it_at_rec(it), it->ii_state == IAM_IT_SKEWED));
    }
```

Node invariant checked by dx_node_check().

Iterator state diagram:

And iterator function/state compatibility matrix:

	DETACHED	SKEWED	ATTACHED
iam_it_get()	X		
iam_it_put()	X	X	X
iam_it_rec_{get,set}()			X
iam_it_rec_insert()		X	X
iam_it_rec_delete()			X
iam_it_next()		X	X

"X" means that calling corresponding iterator function is allowed in this state.

5.2 Locking

For the initial version a read-write semaphore is used to serialize accesses to the given container. Iterators created with IAM_IT_WRITE flag acquire semaphore in write mode when switching into IAM_IT_ATTACHED state, other iterators acquire semaphore in read node. Semaphore is released when iterator transitions to IAM_IT_DETACHED state.

5.3 Recovery

Iam itself does not add new issues to the recovery: it relies on underlying file system transaction service to function correctly in the case of crash.

6 Environment

6.1 Disk Format Changes

Iam supports legacy containers binary compatible with ext3-htree directories. New containers can be created in legacy mode. If new container is created in non-legacy node, it cannot be used by old servers.

"Compat" and "rocompat" fields of the file system are updated accordingly when non-legacy container is created.

Fsck support for non-legacy containers is planned.