Parallel Directory High Level Design

This document was submitted to OpenSFS for review on: 26-10-2011. This document was signed off on 2011-11-09.

Introduction

Single directory performance is critical for HPC workloads. Many applications create a separate output file for each task of a job resulting in hundreds of thousands of files written to a single output directory. Currently, both filename lookup and file system modifying operations such as create and unlink are protected with a single lock for the whole directory. This causes MDS threads to block for the one thread holding the directory lock, which may be waiting for disk IO to complete (e.g. reading a directory block from disk, or waiting for enough space in the journal).

This document is high-level design for a parallel locking mechanism for single ldiskfs directories. The current lock for a single directory can only be held by one thread at any time. The new parallel locking mechanism will allow multiple threads to lookup, create and unlink operations simultaneously.

Requirements

This section is just taken from our “Parallel Directory solution architecture”

Parallelize file creation, lookup and unlink large shared directory

Ldiskfs uses a hashed-btree (htree) to organize and locate directory entries, which is protected by a single mutex lock. The single lock protection strategy is simple to understand and implement, but is also a performance bottleneck because directory operations must obtain and hold the lock for their duration. The Parallel Directory Operations (PDO) project implements a new locking mechanism that ensures it is safe for multiple threads to concurrently search and/or modify directory entries in the htree. PDO means MDS and OSS service threads can process multiple create, lookup, and unlink requests in parallel for the shared directory. Users will see performance improvement for these commonly performed operations.

It should be noted that the benefit of PDO may not be visible to applications if the files being accessed are striped across many OSTs. In this case, the performance bottleneck may be with the MDS accessing the many OSTs, and not necessarily the MDS directory.

No performance degradation for operations on small directory

Htree directories with parallel directory operations will provide optimal performance for large directories. However, within a directory the minimum unit of parallelism is a single directory block (on the order of 50-100 files, depending on filename length). Parallel directory operations will not show performance scaling for modifications within a single directory block, but should not degrade in performance.

No performance degradation for single thread operations

In order to be practically useful, any new locking mechanism should should maintain or reduce resource consumption compared to the previous mechanism. To measure this, performance of PDO with a single application thread should be similar to that of the previous mechanism.

Easy to maintain

The existing htree implementation is well tested and in common usage. To avoid deviating from this state, it is not desirable to significantly restructure the htree implementation of ldiskfs. Ideally, the new lock implementation would be completely external to ldiskfs. In reality, this is not possible, but a maintainable PDO implementation will minimize in-line ldiskfs changes and maximize ease of maintenance. There is relatively little interest to accept this change into the upstream ext4 codebase because it is only useful for a workload that is very uncommon for normal servers (hundreds or thousands of threads operating in the same directory).

Definitions

Ldiskfs filesystem

ldiskfs filesystem is backend filesystem of Lustre. ldiskfs is an enhanced version of ext4.

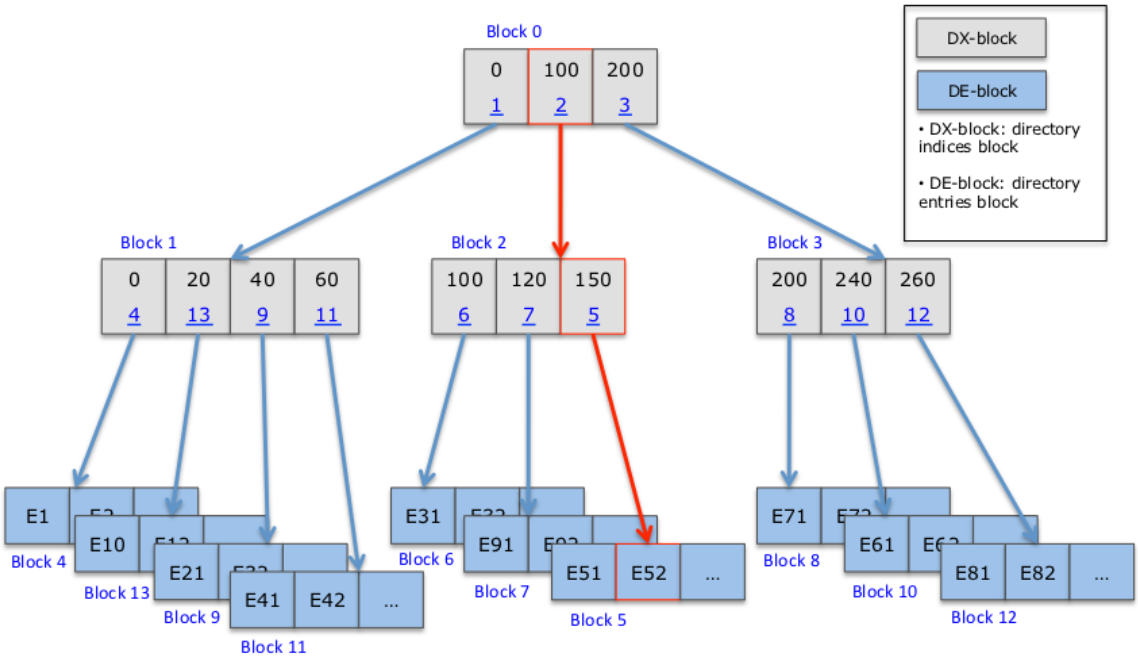
Htree (hashed-btree) and indexed directory

Hashed-btree is the data structure that is used by ext3/4 and ldiskfs as directory layout. The figure above illustrates a ldiskfs indexed directory stored as a htree.

There are two types and blocks in a indexed directory:

- DX-block (directory indices block): stores hash-value/block-ID pairs
  - Hash-value: hash value of the entry name.
  - Block-ID: either file logical block number of leaf block, or the next level indices block.
- DE-block (directory entries block): stores directory entries (filenames).

Htree path



A tree node path for a name entry. The red line in Graph-1 is htree path for name E52.

Htree operations

Numerous operations may be conducted on a on htree:

- Lookup
  Search a name entry in htree, or probe htree path for the name entry
- Insert name entry
  Insert a name entry to the DE-block on htree path
- Delete name entry
  Remove a name entry from DE-block on htree path
- Split DE-block
  If an entry is inserted into a full DE-block, a new block is allocated and half of name entries in the original DE-block are moved to the new block.
- Split DX-block
  When a new DE-block is allocated it must be inserted into the parent DX-block. Inserting a DE-block into a full DX-block results in the DX-block being split.
- Grow htree
  A htree grows if a DE-block split takes place and all the DX-blocks on the htree-path are full. In this case another layer is added to the htree.

Hash collision

Entries in htree directory are indexed by a hashed value of the name. An unlikely, but finite possibility exists that any two names do not generate a unique hash-value. This is called hash collision. If these names spread over more than one block, then searching any of these names needs to iterate over all these blocks, and this should be handled correctly for PDO.

Htree-lock

htree-lock is an advanced read/write lock. There are two locking operations for htree-lock.

- child-lock
  child-lock may be used to protect any node in htree with PR mode (Private Read) or PW (Private Write) mode. For example, a child-lock may protect a DE-block of a directory while a name entry is inserted.
- tree-lock
  tree-lock is used to protect the htree against any operations not protected by child-lock. For example, if a thread adds a new level to htree, it will obtain a tree-lock.

OSD object

OSD object is a in-memory object that represents a inode of backend filesystem.

Changes from Solution Architecture

None.

Functional specification

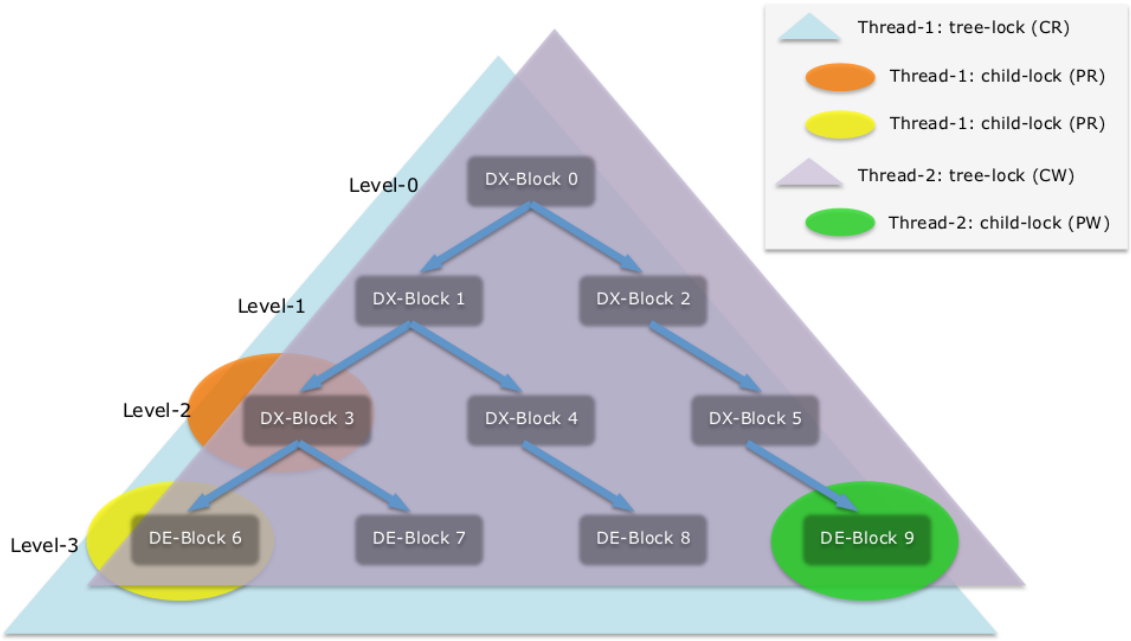
This section of the document covers design of htree-lock and how to exploit htree-lock to support parallel directory operations.

htree-lock definition

The htree-lock lock mechanism can provide multiple level of protection for tree-like data structure.

There are two external data structures for htree-lock:

- htree_lock_head_t
  htree_lock_head_t is created for the target resource (directory), it is a 'OSD object' in this design.
- htree_lock_t
  htree_lock_t is created for each thread that requests access the htree, it is a member of osd_thread_info in this design.



Two types of locking operation are available: tree-lock and child-lock. Graph-2 shows how they are used to protect htree directory.

To understand htree-lock in practice, let us assume that two threads want to access the same directory simultaneously (figure above):

- Thread-1
  - Takes CR mode tree-lock. This still allows other threads to get CW or CR lock on the htree directory, but prohibits PW, PR or EX locks on the htree directory.
  - Takes PR mode child-lock on block-3 and block-6 so these two blocks cannot be modified.
- Thread-2
  - Takes CW mode tree-lock. This is compatible with the CR tree-lock taken by thread-1.
  - Takes PW mode child-lock on block-9 and modifies the block (i.e: insert a new entry). This lock gives this thread exclusive access to block-9.

For a N-level htree, a child-lock can protect level[PUB:N] and level[PUB:N-1] blocks of htree directory. A tree-lock can protect all other level blocks. Using the example presented in the figure above, block-0, block-1 and block-2 are protected by tree-lock.

- An initial implementation of PDO only used a child-lock to protect level[PUB:N] blocks, or DE-block, and using tree-lock to protect all other DX-blocks. Prototyping this design revealed a high occurrence of DE-block splitting:
  - As described in solution architecture, each DE-block (level[PUB:N] block) can contain about 80 entries. If level[PUB:N-1] DX-blocks are protected by tree-lock, then the entire tree must be locked after inserting about 80 entries or 1.25% of all insertions (need to split the DE-block and insert new DE-block to its parent DX-block). This may be a performance issue, particularly as hundreds or thousands service threads may try to inserting/remove entries simultaneously.

- In theory a child-lock can protect level-M blocks (0<= M && M <= N). However, in practice it is only used to protect level[PUB:N] and level[PUB:N-1] blocks. The reasoning behind this is twofold:
    - Restructuing the ldiskfs htree implementation is undesirable. The existing htree implementation has specific logic path for level[PUB:N] DE-block and level[PUB:N-1] DX-block. This makes it simple to add a htree-lock. Requiring the use of an htree-lock at any level will likely require a significant rewrite current code and make it hard to maintain.
    - As described in the solution architecture: using child-lock is used to protect level[PUB:N] DE-block and level[PUB:N-1] DX-block, and a tree-lock to protect all other blocks, the chance of locking the whole tree is about 1/(512 * 80) or 0.00244%. This is judged to be sufficiently small to be negligible.

Hree-lock interfaces and compatibility matrix

The tree-lock API is:

- `htree_lock(htree_lock_t *lock, htree_lock_head_t *head, int mode)`
  Lock the htree with specified locking mode
    - Parameter `@lock`: per-thread lock handle.
    - Parameter `@head`: the htree-lock created for the target resource (directory).
    - Parameter `@mode`: locking mode of tree-lock which can be any of EX, PW, PR, CW, CR. It is anticipated that most operations will only take CW and CR so the htree can be accessed concurrently.
- `htree_unlock(htree_lock_t *lock)`
  Release tree-lock, and all child locks holding by current thread.

Child-lock interfaces

If a share access mode lock (CW or CR) is obtained for a htree an additional “child lock” may also be obtained and additional protection is gained on portion of the htree:

- `htree_node_lock(htree_lock_t *lock, int mode, __u32 key, int depth).`
  this function must be called after `htree_lock()`
    - Parameter `@lock`: per-thread lock handle.
    - Parameter `@mode`: child lock mode, there are only two locking modes for child lock (PW and PR).
    - Parameter `@key`: ID of the child resource to protect, for example: block-ID.
    - Parameter `@depth`:
      Child lock may have N-depths, each depth has a individual lock and can protect a specific resource.
      For example, in the figure above, thread-1 is using the 0-depth of child-lock to protect the last level DX-block (block-3), and the 1-depth of child-lock to protect DE-block (block-6):
      `htree_node_lock(lock, HTREE_LOCK_PR, 3, 0);`
      `htree_node_lock(lock, HTREE_LOCK_PR, 6, 1);`
      The third parameter is “key”, in this case, a block-ID. The last required parameter is depth.
      Thread-2 is using the 1-depth to protect block-9. Thread-2 did not require the 0-depth lock.
      `htree_node_lock(lock, HTREE_LOCK_PW, 9, 1);`
      NB: The application author is responsible for ordering of locks to avoid deadlock.
- `htree_node_unlock(htree_lock_t *lock, int depth, void *event)`
  Unlock child-lock at `@depth`,
  The purpose of @event will be described later.

Child-lock advanced features:

- Non-blocking lock
  Verbose API for child lock: `htree_node_lock_try(… int wait, …);`
  This call will immediately return 0 to caller if `@wait` is false and required lock can not be granted. Otherwise this call will return 1 with hold of the required lock. This feature can be used as a spinlock for some fast

operations. This feature may also be helpful for violating locking order to avoid extra locking dance and possible thread context switch.

- Locking event
  A thread can listen on “locking event” of the resource even after releasing the child-lock. This event will notify listeners of change after the lock is released.
  The below table demonstrates use of these calls. When thread-1 releases the child-lock it provides a buffer for event `@event_buf`. As a result, thread-1 listens on “locking event” of the child-lock it just released. thread-1 will stop listening after calling `htree_node_stop_listen()`.
  In this example, thread-1 will observe updates in @event_buf after listening has ceased (`htree_node_stop_listen()`). thread-2 has obtained a lock for the same resource `@key` and sent `@event` to all listeners on the resource.

This feature will provide for complex locking strategies.

	Thread 1	Thread 2
1	`htree_node_lock(…key, dep)`	
2	……	
3	`htree_node_unlock(…, dep, event_buf)`	
4		`htree_node_lock_try(…key, dep, event)`
5	`htree_node_stop_listen(…)`	……

Configurable buffer LRU cache

Buffer least recently used (LRU) is a per-cpu cache of Linux for fast searching of buffers. The default buffer LRU size is eight. Eight is too small for Lustre where supporting N-levels htree for very large directories is desirable. In addition, future Distributed Namespace work will require a larger LRU. If the default buffer size remains at eight, as buffer slots are consumed hot buffers may be excluded from the LRU cache because the LRU cache is unrealistically small. This is scenario will significantly harm performance as buffers required - but not available in the LRU cache - are slow and expensive to find.

A simple kernel patch will be necessary to allow specification of the LRU size. By default, the LRU size to be set to 16.

Use Cases

Non-indexed directory

Obtain EX tree-lock for all changing operations and obtain PR tree-lock for all non-changing operations.

Readdir for indexed directory

Obtain PR tree-lock. Currently Lustre Distributed Lock Manager (ldlm) will obtain PR lock on the directory. Improving

this behavior requires changing ldlm and is out of scope of this project.

Lookup under indexed directory

Obtain CR tree-lock and obtain PR child-lock for the target DE-block. If a hash-collision occurs, also obtain PR DX-lock for the last level indices-block to ensure searching will be atomic. Additional details on this topic appears in the logic specifications.

Remove entry under indexed directory

Obtain CW htree-lock and obtain PW child-lock for the target DE-block. The PW lock is obtained on the last level DX-block to ensure searching is atomic if there is hash collision.

Add entry under indexed directory

Obtain CW htree-lock and obtain PW lock for the target DE-block. The PW child-lock is obtained for the last level DX-block if the target entries-block is full and needs to be split.

If the last DX-block is full and a split is required, then re-lock the htree with EX mode and retry. As discussed, this is expected to be sufficiently rare that this overhead can be ignored.

Logic specification

Prevent starvation on parent lock

Thousands of server threads may enter the same directory by holding shared lock mode (CW, CR). If compatible locking request are always granted, and there are a few threads acquire incompatible locks (i.e. EX), starvation may occur. A simple solution if presented that will block locking request if:

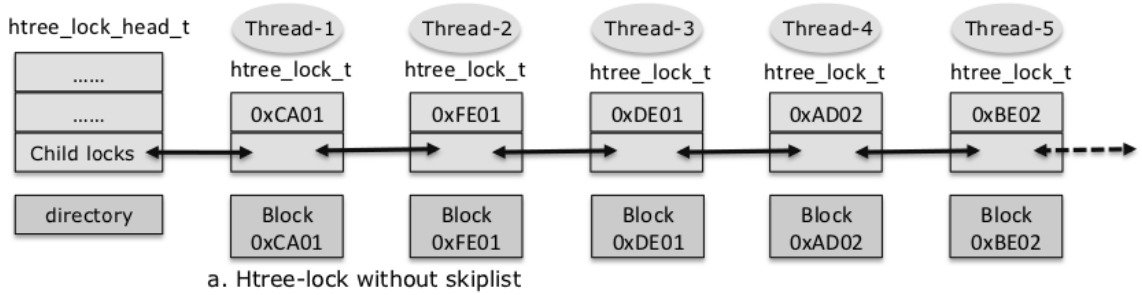
1. there is conflict with any granted lock
2. there is conflict with any blocked lock, so any locking request after PW or EX locking request will block even if PW/EX locking request is not granted yet.

As discussed, most locking requests require compatible locks. As a result, this scheme is not expected to degrade performance.

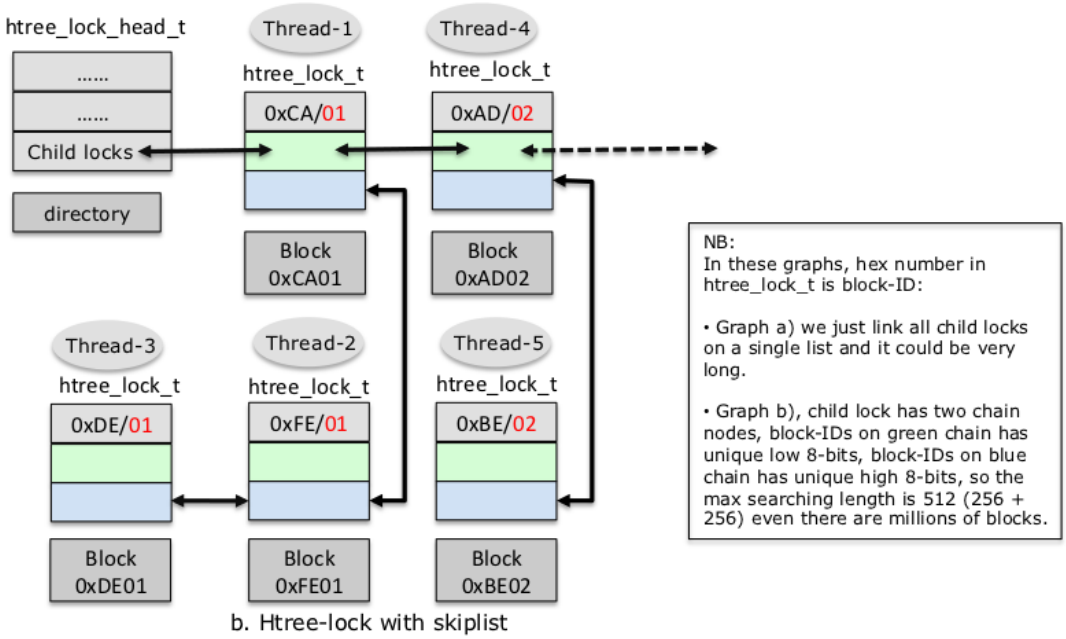
Child-lock scalability

Assume thousands of server threads have obtained tree-lock with shared mode. Now assume these threads operate on different child resource (e.g. search on different entries-block). Each thread will acquire a child-lock with different key (block-ID). These child-locks (`htree_lock_t`) will enqueue to a increasing list if they are simply linked on htree_lock_head_t. As such a list grows, search performance for a given block-ID will degrade.

As a result, new structure name `skiplist` will replace the list. The figure below illustrates `skiplist`.



a. Htree-lock without skiplist



b. Htree-lock with skiplist

Ldiskfs lock definition and Locking order

Child-locks protect level[PUB:N-1] DX-block and level[PUB:N] DE-blocks. Other changes are protected by htree-lock. 2-depths child lock are defined as:

- DX-lock: a child-lock to protect level[PUB:N-1] DX-block.
- DE-lock: a child-lock to protect level[PUB:N] DE-block.

The use-cases include:

- All no-change operations obtain both DX-lock and DE-lock with PR mode.
- All changing operations obtain both DX-lock and DE-lock with PW mode.

However, this design has limitations: Assume the tree has only one level DX-block (one DX-block). In this case, all threads always require PR/PW lock on the DX-block. The effect is that htree-lock will behave as a single rw-semaphore and parallel locking benefit is lost.

Assume the tree has two level DX-blocks, and there are a small number of second level DX-blocks. In this case, the effect is a small number of rw-semaphores to protect the htree directory. If many hundred of threads request parallel access to the directory, and only a small number of rw-semaphores are available, only a few threads can enter at any time.

The solution to this problem is add another depth for child-lock:

- DX-spinlock

rw-spinlock for level[PUB:N-1] DX-block (again). It protects the same DX-block, but a different lock depth is used to protect different operations:

- DX-lock is a blocking lock and protects the entire split process of DE-blocks. As a result only one thread can split DE-blocks under the same DX-block.
- DX-lock is not necessary for lookup, delete or insert, required only if:
  - Hash collision for lookup/delete
  - Split DE-block
- DX-spinlock is non-blocking lock that is used to perform a quick scan of DX-block, or changing of DX-block (i.e. insert a new DE-block entry to the DX-block.) This lock is always required but it is non-blocking and only protecting light operation. There is no thread context switch even under lock contention.

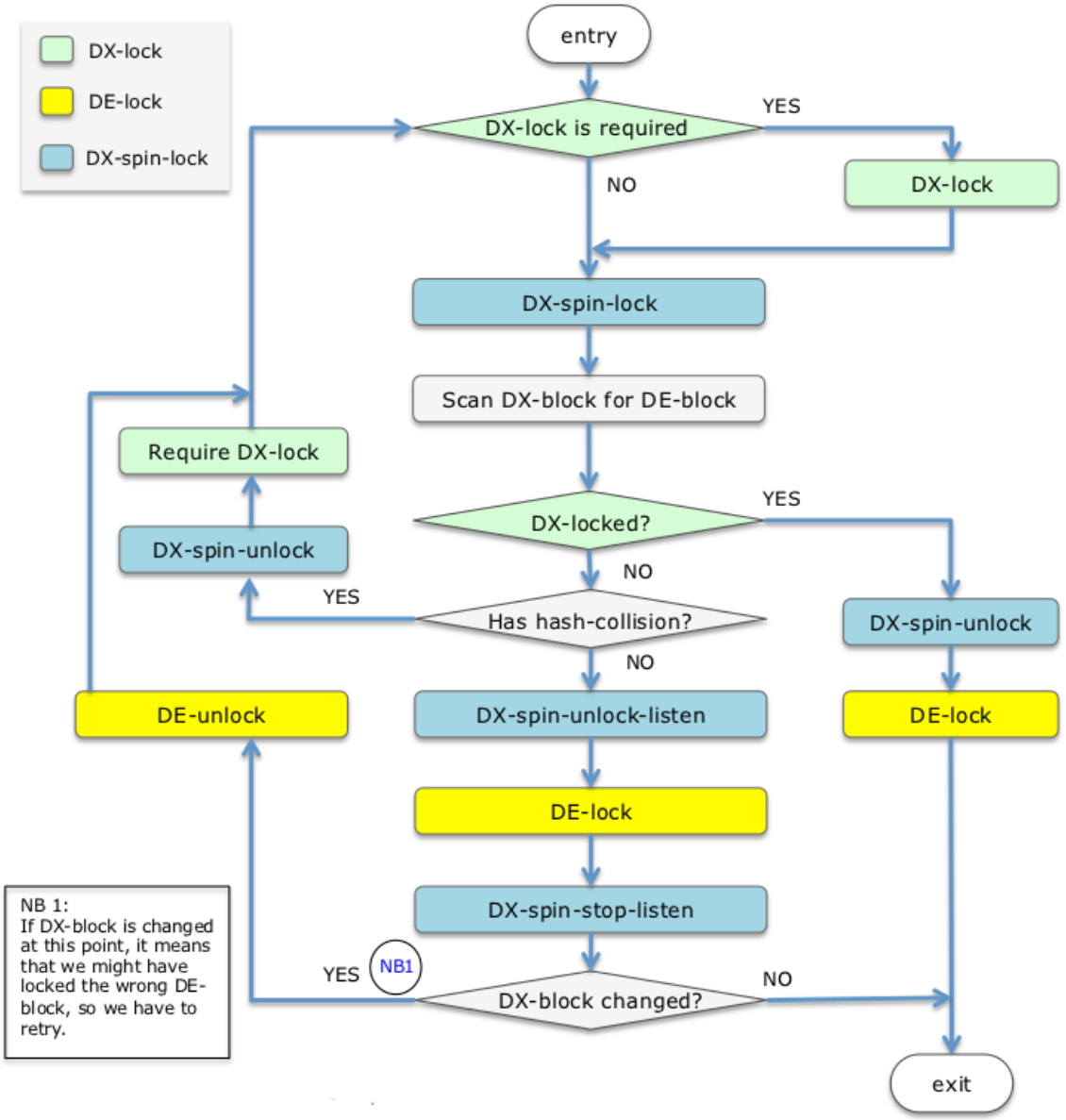
DX-lock and DE-lock are both blocking locks, DX-spinlock is non-blocking lock. After obtaining DX-spinlock it should not be necessary to acquire a DX/DE-lock. The locking order must be: DX-lock, DE-lock, DX-spinlock. All these locks only have two modes: PR and PW.

Probe htree path

Probe htree patch is a key functions that must be changed within ldiskfs. The function name is `dx_probe()`. `dx_probe()` probes htree-path for a name entry.

In current ldiskfs, this function is only called with hold RW semaphore. RW semaphore only two modes: protected read and protected write.

PDO requires a new implementation to support shared locking mode (CR or CW). While one thread is searching the htree, other threads may simultaneously change the htree. The new `dx_probe()` must ensure data correctness.



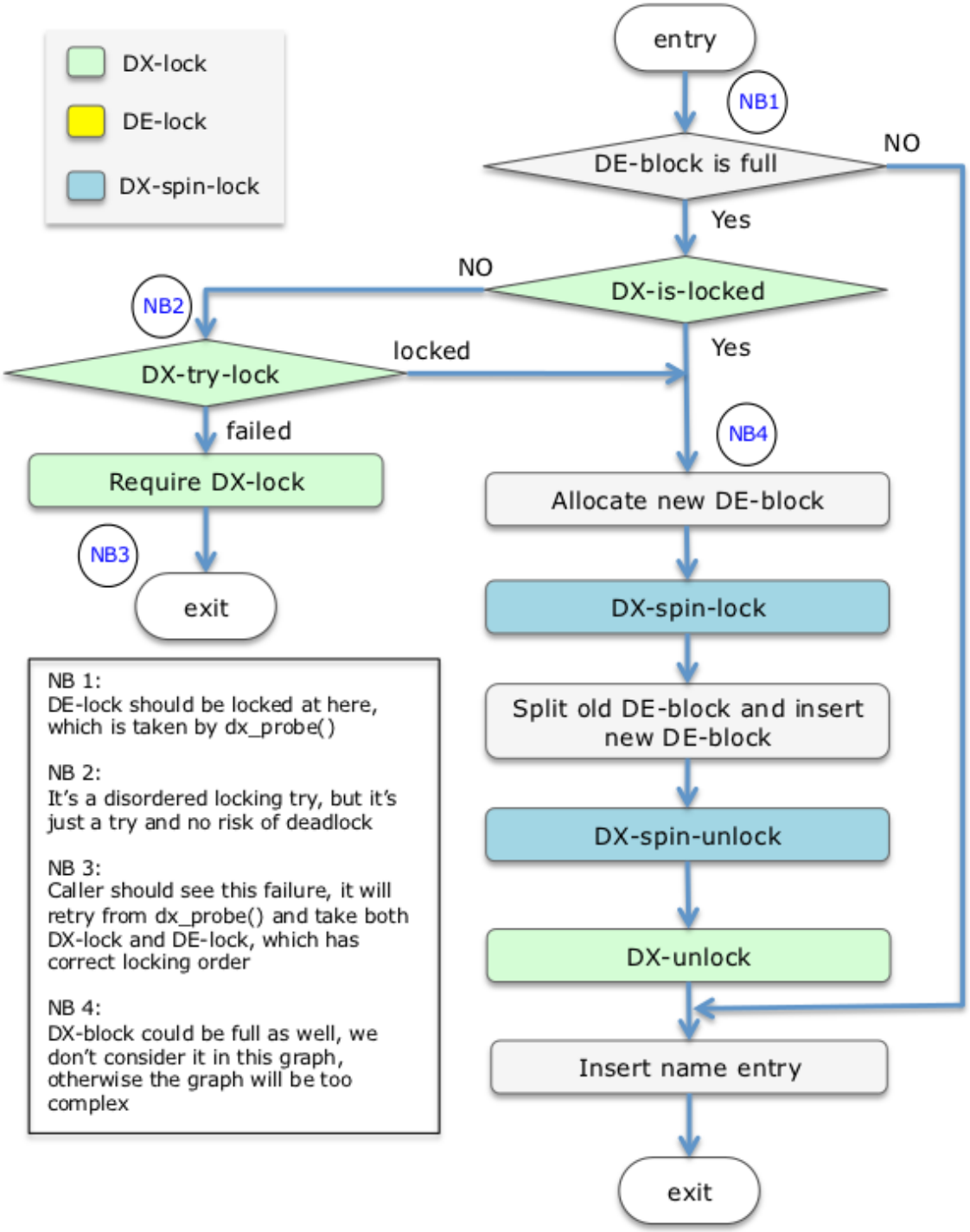
The figure above illustrates the logic using child-locks of htree-lock to protect the process of `dx_probe()`. The method is:

- call `dx_probe()` with hold shared locking mode (CR or CW).
- Level[PUB:0, 1, …, N-2] DX-blocks are not protected by tree-lock and do not require any additional protection. This figure only shows logic of protecting level[PUB:N-1] DX-block and level[PUB:N] DE-block.
- locking order to avoid deadlock
  - As previously described, the correct locking order is: DX-lock, DE-lock, then DX-spinlock.
  - DX-spinlock must be held to scan the level[PUB:N-1] DX-block and discover the target DE-block. The DE-lock is obtained. This process violates the locking order, so the DX-spinlock must be released before locking DE-lock. However, no guarantee is available that the target DE-block will not be changed after we release DX-spinlock and take DE-lock. The solution is to call DX-spin-unlock-listen:
    - DX-spin-unlock-listen will be notified if the DE-block is changed (i.e. it is split by another thread.)
    - If notification arrives, attempt to re-validate the htree-path again ensuring the correct DE-block is locked.
- This function will return reference of DE-block with hold DE-lock, and caller can continue to search/delete the target entry in the target DE-block, or insert a new entry to the target DE-block.
- If there is hash collision for “lookup”, then the DX-lock must be obtained as well. This will allow searching the

next adjacent DE-block. In this case, the parent (level[PUB:N-1] DX-block) remains unchanged.

Add name entry and split DE-block

This is logic specification for add entry or split DE-block, which is `dx_add_entry()` of ldiskfs. This function is always called after `dx_probe()`. The target DE-block will be already locked by DE-lock. To simplify the figure, growing htrees and DX-block split have been omitted.



If the DX-block is full or the tree is full and needs to grow, we just need to lock the whole htree with EX mode and repeat the process of `dx_probe()` and `dx_add_entry()`.

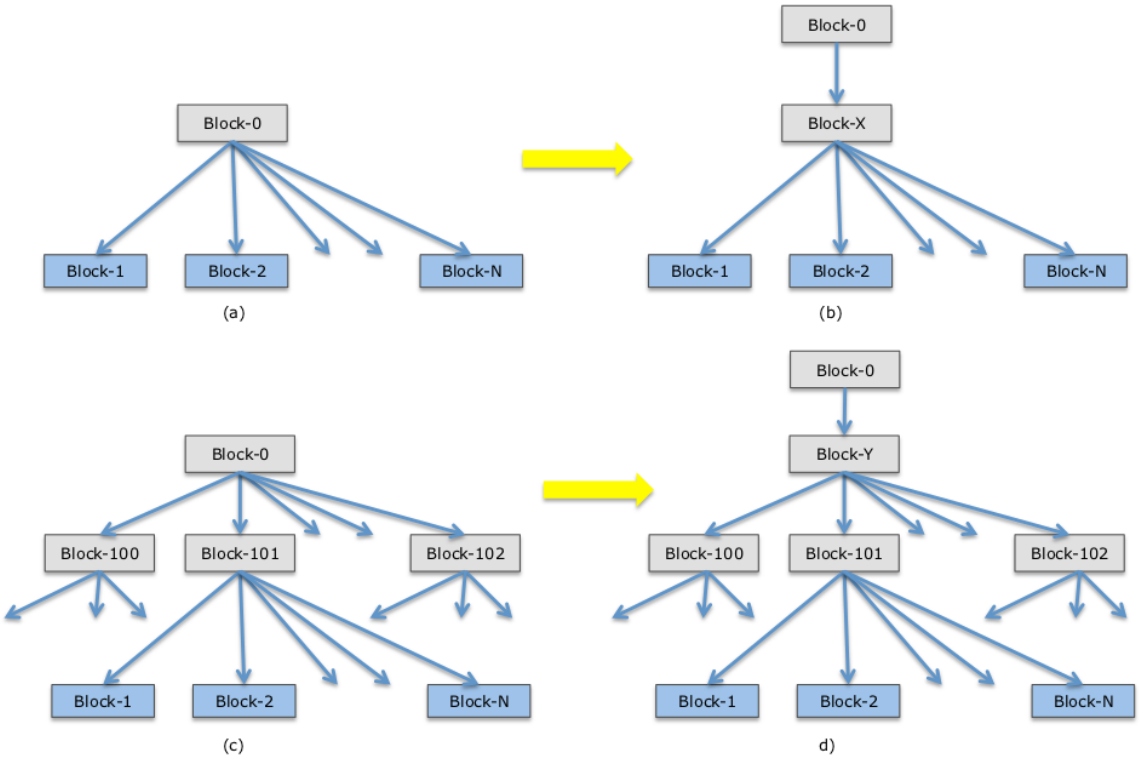
Remove name entry

Removal of name entry (delete_entry) is simple. It is called after after `dx_probe()` and inherits the locked target DE-block returned by `dx_probe()`. No changes are necessary to `delete_entry()`.

Grow htree and N-level htree

Htree growing logic is as follows:

- Non-indexed directory grows to 1-level indexed directory.
  This is identical to the current ldiskfs behaviour. A non-indexed directory is protected by EX locking mode. This means the directory freely modified.
- N-level indexed directory grows to (N+1) level indexed directory.
  This is similar to current ldiskfs behavior. However current ldiskfs can only have 2-level DX-blocks:
  1. If a DE-block is split, an entry is inserted into the DX-block. If the parent DX-block is full and all DX-blocks on the htree-path are full, then a new DX-block is allocated and the depth of the htree grows.
  2. Copy all entries from root DX-block to new DX-block.
  3. The root DX-block is still the root. However it only has one entry which points to the new DX-block. The htree now has N+1 levels.



(a) and (b) show htree growing from 1-level to 2-level. (c) and (d) illustrate htree growing from N-level to (N+1) level. A new DX-block is added as the only child of DX-block 0 and obtains children of DX-block 0.

A minor difference exists between growing from 1-level to 2-level, and N-level to (N+1)-level:

- - The root DX-block always has additional bytes for describing htree. After new DX-block obtains all the entries from the root block without the additional extra bytes, the new DX-block has free bytes to receive a new entry so we can go ahead splitting DE-block.

For example, spliting DE-block[PUB:2] in Graph-6 (a): after the htree grows, the htree will become (b). A new DX-block[PUB:X] will always have free bytes for new DE-block split from DE-block[PUB:2]. In this case a split on DE-block[PUB:2] can be performed and the new DE-block added to DX-block[PUB:X]. Finally, insert name entry to target DE-block.

- - While growing from N-level to (N+1)-level, the new DX-block is still inserted as child of the root DX-block. For example, in Graph-6 (d): after htree growing, block-Y is the new block. block-Y now has free bytes that can receive an entry. However, if DE-block[PUB:2] is split, and its parent DX-block[PUB:101] is split, block-Y remains full and can not insert any new block entry.

This situation can be resolve by simply aborting the insert operation after growing the tree and restarting from dx_probe. In the second shot, DX-block[PUB:101] is full but a split can be performed because the new DX-block[PUB:Y] has free entries. In this case, the process is splitting DX-block, splitting DE-block and inserting an entry.
In practice, to simplify the logic, `dx_probe()` is always restarted after growing of htree.

Cache block 0 for in htree-lock (optional)

Htree-lock can provide private fields to store extra information. For example, block-0 (root block of the directory) can be cached. Subsequent threads entering the same directory will not have to search the block again. This will save time and in some cases a slot in buffer LRU cache. This feature is optional as it is unknown what the performance benefit will be in practice and it may increase overhead of single thread use-case.

Configurable parameters

Define configurable parameters available in the feature.

Osd-ldiskfs tunable

New module parameter for ldiskfs:

- ldiskfs_pdo=1: turn on PDO for ldiskfs
- ldiskfs_pdo=0: turn off PDO for ldiskfs

A proc entry “ldiskfs_pdo” will be provided to osd-ldiskfs so user can turn on/off PDO without reloading moudules.

Kernel config for buffer LRU size

Options for buffer LRU size in kernel config file: 8, 16, 32, default will be 16 instead of 8.

API and Protocol Changes

Ldiskfs API changes

A new parameter `@hlock` will be added to a couple of exported ldiskfs/ext4 APIs:

- ldiskfs_add_entry

```
int ldiskfs_add_entry(handle_t \*handle, struct dentry \*dentry,
   struct inode \*inode, htree_lock_t \*hlock);
```

- ldiskfs_find_entry

```
struct buffer_head *
ldiskfs_find_entry(struct inode \*dir, const struct qstr \*d_name, struct
    ext4_dir_entry_2 \*\* res_dir, htree_lock_t \*hlock);
```

The new parameter is a thread htree-lock handle created in osd-ldiskfs. If these APIs are called from VFS, `@hlock` should be NULL and ldiskfs will assume that PDO is off and protection comes from semaphore. All operations on @hlock should be NOOP in that case.

Open issues

N/A

Risks and Unknowns

Locking overhead for single thread performance

Htree-lock is more expensive lock than regular rw-semaphore or mutex. Single threaded operations will likely have higher overheads while using htree-lock - although multiple threads have better concurrency. It is unlikely to be a significant resource drain because locking overhead is just a tiny part relative to other processing of creating/removing a file over Lustre server stack.