Design and Implementation of XNU port of Lustre Client File System

Danilov Nikita

2005.02.01

Abstract

Describes structure of Lustre client file system module for XNU (Darwin kernel). In particular, changes that were necessary in core XNU kernel to enable unique Lustre requirements (e.g., intents) are discussed in much detail. Changes to the platform-independent core of Lustre in order to make it more portable are discussed in the companion paper Lustre Universal Portability Specification.

Contents

1 Introduction

This document describes design and implementation of XNU port of Lustre client file system (*xll*). XNU is kernel component of *Darwin* (`http://developer.apple.com/darwin/`), and Darwin is low-level portion of Mac OS X (`http://www.apple.com/macosx/`). xll is XNU equivalent of *llite* module for Linux. Specifically, this document concentrates on describing changes that were necessary in XNU core to allow efficient xll implementation.

Structure of the document is following: first some background information of XNU kernel is provided, then HLD for components of XLL is described.

2 Distribution

People interested in

- porting Lustre to new platforms,
- debugging Lustre on XNU

should read this document. Understanding of Linux kernel internals (mostly file system and VM) and implementation of Lustre on Linux is assumed.

This document contains confidential information, and is a property of ClusterFS. If you received it without appropriate authorization, please shoot yourself to death before continuing.

3 Background on XNU

XNU is kernel of Mac OS X operating system that Apple currently ships with its desktop and server hardware. The name XNU presumably stands for "Xnu is Not Unix". Apple maintains proprietary version of this kernel. Periodically it releases sources to public after removing some small pieces from it. Released source is distributed under APSL (`http://www.opensource.apple.com/apsl/`) license and it used by OpenDarwin (`http://www.opendarwin.org/`) project.

XNU code is very diverse, including some really old and obscure bits. It can be roughly divided into three large components:

- osfmk-part: this is MACH source code. It provides basic functionality like threads, synchronization objects, scheduler, memory management, and VM used by other components. MACH code used in XNU is a descendant of original CMU MACH 3 that was later used by OSF/1 ("osf" part), and after that by (now defunct) mkLinux (`http://www.mklinux.org`) project ("mk" part).
- bsd-part: this runs on top of MACH and provides standard UNIX/POSIX APIs and semantics. BSD code in XNU is a mix of old BSD-Lite2 and NeXT-step code-bases. It has structure of traditional UNIX kernel with some low-level parts removed, and plugged directly into MACH. Note: contrary to the popular belief, XNU is *not* a micro-kernel: BSD code executes in the kernel mode and interacts with MACH through normal function calls rather than IPC.
- IOKit: this is framework for device drivers in C++. We are not interested in this.

Note: this document mostly deals with Mac OS X 10.3 (Panther), it is supposed to be updated when version 10.4 (Tiger) will be available.

3.1 XNU VFS

XNU inherits VFS from FreeBSD with very few changes. FreeBSD VFS is based (mainly) on SunOS VFS. Motivation and background of FreeBSD VFS design can be found at `http://docs.freebsd.org/44doc/papers/fsinterface.html`. Support for "remote" file systems was among main goals, along with support for the "stackable" file systems, see `http://www.chesapeake.net/~jroberson/Heidemann95e.pdf` for more details.

Main data structures in XNU VFS are struct vnode and struct mount. vnode represents file system object (a la struct inode in Linux). mount is a file system. XNU doesn't support Linux distinction between super-block (an instance of file system) and mount-point (a point where super-block is grafted into global name-space, represented by struct vfsmount in Linux).

vnode contains a pointer to an array of "vnode operations" or "VOPs". struct mount also points to an array of VFS operations.

Important differences with Linux:

– vnode does *not* contain attributes like [cma]time, nlink, size, [ug]id, etc. Attributes are kept in separate data structure struct vattr. Each time VFS wants to access vnode attribute, it calls VOP_GETATTR(). To update attributes, VOP_SETATTR() is used. It seems that this has important ramifications for Lustre: there is no need to always keep vnode attributes up-to-date: attribute intent lock can be obtained during VOP_GETATTR().

– locking. VFS locks vnode during while performing operations related to it. Implementation of vnode lock is up to file system back end. VFS uses VOP_LOCK() and VOP_UNLOCK() methods as an interface. VOP_LOCK() is called with an argument containing a set of locking flags: exclusive or shared lock, whether recursion is allowed, can lock request block, etc. File systems usually implement VOP_LOCK by bsd/kernel/kern_lock.c:lockmgr()—a lock manager. It seems tempting to try map VOP_LOCK() functionality onto LDLM one, but I think that this should not be done, at least in the initial implementation. Instead, our VOP_LOCK and VOP_UNLOCK will use standard lockmgr() interface, LDLM locks will be acquired separately as needed. Note that this means, that *local* locking (within single client) is coarse grained: whole object is locked (in either shared or exclusive mode) for the duration of system call. I think that this is acceptable, because the same thing happens on Linux (with ->i_sem semaphore).

– name resolution. XNU has standard UNIX namei() interface to resolve path-names. namei() descents through file system tree, calling VOP_LOOKUP() to resolve single path component. Symlinks are handled through VOP_READLINK() method. VOP_LOOKUP() gets a lot of flags indicating required locking mode on the target vnode and parent directory. It is also possible to determine when last path component is being resolved. Basically VOP_LOOKUP() method implementation is much more complex than ->lookup() in Linux, including handling of special cases, permission checks, and name-cache.

– name cache. XNU VFS is vnode based. All VOP_* operations take vnode as argument. There is simple name-cache, but it is not used by the VFS layer. File system back-end is supposed to use it by its own. This provides some advantage for Lustre: all file system operations go through VOP_LOOKUP() stage (even if name/vnode is in name cache). This simplifies lookup lock revocation: in revocation call-back vnode is removed from name cache, and we are free to implement whatever locking we need at this point to avoid races with VOP_LOOKUP() that pokes into name cache concurrently.

– file descriptors. File descriptors are completely handled by the VFS in XNU. There is a per-process table of open file descriptors, and at the beginning of file descriptor based system calls (fchmod(2), ftruncate(2), etc.), file descriptor is obtained from this table. File descriptor contains a pointer to vnode. After extracting that pointer, all operations are based on vnode: VFS calls VOP_* methods and doesn't pass file descriptor to it as an argument. As a result it is not possible for file system to get access to the file descriptor used by current operation. This is not acceptable to Lustre, because we want read-ahead parameters to be associated with each file descriptor.

3.1.1 namei()

Path-name resolution API in XNU is defined through following API:

```
struct componentname {
    u_long  cn_nameiop; /* namei operation */
    u_long  cn_flags;   /* flags to namei */
    ...
    char   *cn_nameptr; /* pointer to looked up name */
    long    cn_namelen; /* length of looked up component */
    ...
};
```

```
struct nameidata {
    ...
    struct  vnode *ni_vp;           /* vnode of result */
    ...
    struct componentname ni_cnd;
};
int namei(struct nameidata *nd);
```

Control flow in a typical system call accepting path-name as its argument (open(2), truncate(2), chmod(2), chdir(2), link(2), rename(2), *etc.*) looks like this:

```
int op(struct proc *p, struct op_args *uap, register_t *retval)
{
    struct nameidata nd;
    struct vnode    *vp;
    NDINIT(&nd, OPCODE, ...);
    vp = namei(&nd); /* resolve path-name, return vp locked */
    ...
    VOP_OP(vp, ...); /* call some vnode operation(s) */
    ...
    VOP_UNLOCK(vp);
}
```

All system calls have the same standard set of arguments:

`struct proc *p` process on behalf of which system call is executed (maybe different from current process in some cases);

`struct op_args *uap` packaged arguments of system call. This structure is created by assembly entry point code. System call code extract individual arguments from @uap. This is quite different from Linux, where system call arguments are visible as separate C function arguments. This difference is important to XLL and is exploited to keep amount of changes in the core XNU kernel at minimum (see 4.1.2 on page 10);

`register_t *retval` not actually used by C code.

Control flow in `namei("/foo/bar/baz")` looks roughly following:

```
/* obtain root vnode of file system */
v = VFS_ROOT(); /* This is also called when crossing a mount point */
v = VOP_LOOKUP(v, "foo"); /* @v is vnode of /foo */
v = VOP_LOOKUP(v, "bar"); /* @v is vnode of /foo/bar */
/* if /foo/bar is a symlink, read its content into buffer ... */
VOP_READLINK(v, buf);
/* then paste buffer into path-name, and continue resolution ... */
v = VOP_LOOKUP(v, "first-component-of-symlink");
...
v = VOP_LOOKUP(v, "last-component-of-symlink");
/* after symlink is resolved, */
/* continue with the rest of original path-name */
v = VOP_LOOKUP(v, "baz");
```

Important differences with Linux:

- Resolution of single path-name component is completely handled by VOP_LOOKUP(), in Linux, VFS calls at least ->permission() and ->lookup() methods.
- name-cache is not used at VFS layer. File system backend is free to cache path-name resolution results in any way.
- VFS doesn't handle any „special" names like *dot* or *dotdot*, whereas in Linux *dot* and *dotdot* lookups are completely handled by the VFS. The only case where XNU VFS resolves path-name component without calling VOP_LOOKUP() is when *dotdot* crosses mount point. ***Note***: this is possible source of bugs, because Lustre servers weren't designed to handle dot and dotdot lookups (completely absorbed by Linux VFS), or, at least, were never tested to handle them correctly.

System calls that take file descriptor as an argument (read(2), write(2), ftruncate(2), fchdir(2), *etc.*) usually look like this:

```
int fop(struct proc *p, struct fop_args *uap, register_t *retval)
{
    struct file  *fp;
    struct vnode *vp;
    getvnode(p, uap->fd, &fp);
    vp = fp->f_data;
    VOP_LOCK(vp, LK_EXCLUSIVE | LK_RETRY, p);
    ...
    VOP_OP(vp, ...); /* call some vnode operation(s) */
    ...
    VOP_UNLOCK(vp);
}
```

3.1.2 vnode life cycle

Vnode can be in one of two states: *used* (->v_usecount > 0) and *free* (->v_usecount == 0). Free vnodes are lingering on the free list (vnode_free_list). vget() recovers vnode from free list, changing its state to used, vrele() performs reverse operation. New vnode is created by call to getnewvnode(). This function tries to balance amount of memory occupied by vnode cache: it either allocates the vnode by MALLOC(), or reuses vnode from the free list. In the latter case, VOP_RECLAIM() method is called, that file system may use to release resources associated with vnode. It is *BAD* idea to perform write-back of dirty pages from VOP_RECLAIM() though, because getnewvnode() calls usually come in batches.

3.2 XNU page cache

As was already mentioned, memory management and virtual memory support in XNU are provided by MACH. Internally, MACH considers all available physical memory as subdivided into same sized *frames* (albeit all frames are of the same size, possibility exists to choose this size during boot). Frame is identified by *frame number*.

Virtual memory is divided into pages, each represented by *vm_page_t*. Page is backed up by a frame, and multiple pages can refer to the same frame. Mapping from virtual to physical address is performed in hardware, by using platform dependent data structures, directly accessed by hardware (page tables, inverted hash tables, *etc.*). MACH kernel keeps track of virtual to physical mapping through *vm_map_t* objects. For example, when new frame of physical memory is allocated for kernel usage, vm_page_t is created for this frame, and

page is inserted into kernel vm_map_t. This is quite different from the Linux, where 1-to-1 correspondence between physical and virtual kernel addresses is assumed in most cases.

Page belongs to vm_object_t, which is entity that maintains a collection of virtual pages, and caches some of them in the physical memory (typical example of vm_object_t is vnode). VM subsystem may ask vm_object_t to push some of its cached pages out of the physical memory (when memory goes low), or to bring some pages back into memory (when page fault happens on a page from given vm_object_t that is mapped into user address space).

When free memory goes low, VM subsystem tries to push rarely used pages out of memory. To determine what pages are good candidates for eviction it uses scanning algorithm quite similar to one used by Linux VM (mm/vmscan.c). All pages are divided into two "queues": active and inactive. VM scanner looks at tail of the active queue, and if page there wasn't accessed while on active queue, page is transferred to head of the inactive queue. Not accessed pages from tail of the inactive queue are evicted.

BSD code doesn't access MACH VM objects (vm_page_t and vm_object_t) directly. Instead it uses one of two intermediate layers:

- Universal Page List (UPL), and
- buffer cache.

UPL is XNU specific mechanism of interaction between page cache and upper layers of the system. UPL is a collection of pages contiguous within some vm_object_t. UPL is usually used as a container for pages that are affected by some operation. For example, when VM asks vnode to page its pages out of memory, it calls VOP_PAGEOUT() method passing UPL of pages that should be paged out. One can see UPL as a way to abstract upper layers of system from the knowledge of low-level details of memory management and VM.

When VM or upper layer requests creation of UPL, pages that fall into UPL are scanned, their attributes are stored in special per-UPL array, and pages themselves are locked. As operation on UPL is executed, it can *abort* or *commit* portions of UPL. When page from UPL is aborted or committed, attributes of underlying vm_page_t are updated to reflect possible changes that were made in per-UPL page attributes array.

Note: UPL code (osfmk/vm/vm_pageout.c) is very complicated and almost undocumented. Try to stay as far from it as you can.

Another way to access page cache is through buffer cache. Buffer cache in XNU is quite traditional UNIX buffer cache, except (as in many BSD flavors) buffers are indexed by (vnode, logical-offset) rather than by (device, block-no) as in Linux.

3.3 XNU Synchronization

XNU kernel provides fairly standard set of concurrency control mechanisms:

- Several types of spin-locks:
  - ulock_t: lowest level spin-lock provided by platform-dependent code;
  - usimple_lock_t: spin-lock with debugging, statistics, and preermption support;
  - simple_lock_t: like usimple_lock_t, but vanishes on uniprocessor.
- General purpose sleeping locks:
  - mutex_t: mutex;
  - lock_t: read-write lock;
  - struct semaphore: counting semaphores.

- Generic lock used by BSD part of XNU (mostly by VFS):
  - struct lock__bsd__ (no kidding, that's how it is named): this is generic blocking lock that supports:
    - read-write locking;
    - upgrades from read to write, and downgrades back;
    - try-locking;
    - *instantenious locking* (wait for lock, but do not acquire it, return failure);
    - recursive locking;
    - automatic retying;
    - interlocking with simple_lock_t.
- Funnels. Funnel is sort of named Big Kernel Lock in Linux terms. In other words this is lock that is treated specially by scheduler: it's automatically released when thread is just about to be de-scheduled and is re-acquired when thread is scheduled back. In effect, funnel protects CPU execution of thread. Important differences with Linux are:
  - funnel is a blocking lock (Linux BKL is a spin-lock), it's based on mutex_t.
  - there are multiple funnels. Actually, two of them: kernel funnel, and network funnel.
  - system call declaration, optionally includes indication of a funnel under which system call is executed. For example, all file system related calls are executed under kernel funnel. MACH VM routines also require kernel funnel to work.
  - low level (for BSD) tsleep() function, that puts thread to sleep until specified event is signalled, requires funnel too. This is so, because traditional UNIX idiom

    ```
    while (object->flags & FOOBUSY) {
            object->flags |= FOOWANT;
            tsleep(object, PRIORITY, "waiting for access to object", 0);
    }
    ```

    obviously only works correctly on uniprocessor or when some external serialization between waiters setting FOOWAIT flag, and wakers doing

    ```
    object->flags &= ~FOOBUSY;
    if (object->flags & FOOWANT) {
            object->flags &= ~FOOWANT;
            wakeup(object);
    }
    ```
- Miscellaneous:
  - thread_call_t: timer;
  - struct wait_queue: wait queues;
  - direct thread sleep/wakeup API: assert_wait(), thread_block(), thread_wakeup().

See USENIX Paper `http://www.usenix.org/publications/library/proceedings/bsdcon02/full_papers/gerbarg/gerbarg_html/BSDCon.html` by an Apple engineer for some interesting details.

3.4 Miscellania

XNU error code convention is different from Linux one: functions usually return positive error code (*i.e.*, plain errno value) whereas in Linux negated errno is returned as a rule. XLL uses Linux convention everywhere except for the cases where it return values to XNU code (top level VOP method, kext loading/unloading callbacks, *etc.*).

4 High Level Design

High level design of XLL is partitioned into multiple components:

- intent handling
- sessions
- file descriptor handling
- page cache
- vnode cache
- lock revocation
- VOP_ and VFS_ methods
- debugging infrastructure: pseudo files, and statistical counters
- kext initialization/finalization
- directory code: readdir
- interaction with OSC

4.1 XLL Intent Handling

To minimize number of RPC that are sent to server during system call execution, Lustre client file system for Linux uses *intents* and *raw operations*. Intent is a sort of context that is passed down to the path-name resolution routine. This context contains information about operation that name resolution is part of (which it is a system's *intention* to perform after finishing path-name resolution). Lustre ->lookup() method analyzes this context and if possible pre-executes operation indicated by the context, usually packing it in the single RPC together with request to resolve last path-name component. Raw operations are logically the same thing, they exist as a separate mechanism, because due to certain technical problems, intent doesn't always contain all information necessary to perform an operation.

4.1.1 Requirements

- XNU port has to support optimizations to RPC traffic equivalent to Linux ones;
- minimal changes to XNU core;
- minimal changes to MDC/OSC.

4.1.2 Functional Specification

Whenever path-name resolving routine namei() is called, we pass into it some „context“ information. This information should be sufficient for VOP_LOOKUP() method of Lustre to perform intended operation. RPC is sent to the server when last component of path-name is resolved. Call to VOP_OP() (3.1.1 on page 5) method that, in other file systems performs actual operation, simply returns error code given by server.

Advantages:

- subsumes both intents and raw operations of Linux port.

Possible problems:

- each namei() call-site has to be augmented with context-passing code,
- after calling namei() VFS sometimes analyzes returned state (e.g., whether vnode actually exists or not) and behaves differently depending on result. Our VOP_LOOKUP() should be careful enough to return correct values and to keep VFS happy.

Alternative solution (by Phil Schwan): context is passed into namei(), but that context is smaller than proposed above. That smaller context is only sufficient for VOP_LOOKUP() to return „valid“ state that will pass VFS checks mentioned above. Last VOP_LOOKUP() call is no-op: it does nothing and always return success. Actual operation is performed in VOP_OP().

Advantages:

- fewer changes to the core XNU code (see requirements (4.1.1 on the preceding page)),
- more straightforward mapping between actual and „intended“ execution flow: VOP_LOOKUP() does lookup, VOP_OP() does OP.

Disadvantages:

- will require changes throughout existing XLL implementation, and will diverge XNU and Linux ports further,
- for the cases when VFS requires namei() to return existing vnode, some fake vnode has to be returned and its life-time has to be tracked,
- there is a problem of "morphing" fake vnode into "real" one once VOP_FOO() operation was actually performed. Note that we cannot just substitute "real" vnode, because namei() caller possibly cached ndp->ni_vp in its local variable,
- in the open(2) case, for example, neither VOP_CREATE() nor VOP_OPEN() get complete set of parameters sufficient to send "lookup-create-and-open" RPC to the server: VOP_CREATE() doesn't take fmode, and VOP_OPEN() doesn't take name.

4.1.3 Use Cases

Using intent in a system call

```
int some_syscall(struct proc *p,
                 struct some_syscall_args *uap, register_t *retval);
```

1. System call calls namei() to resolve pathname (usually uap->path). Operation context (including operation tag uniquely identifying namei() call-site, and @uap structure) is passed to namei() through additional field in struct nameidata.
2. namei() calls VOP_LOOKUP() method of XLL (xll_lookup()).
3. xll_lookup() calls xll_intent_lock() to execute intent.
4. xll_intent_lock() extracts operation tag, and, if last component of path-name is resolved, and operation requires intent handling, calls appropriate sub-function passing to it @uap pointer.
5. sub-function sends RPC to the server to resolve last component of path-name and perform required operation.
6. after receiving reply from server, error code is saved (in a xll session, see 5.2 on page 24), and control returns to VFS.
7. VFS calls VOP_SOMEOP() method.
8. VOP_SOMEOP() method immediately returns error code given by server.
9. control return to VFS and then to user.

How do acceptance tests use HLD components?

How do other subsystems use HLD components?

See 3.1.1 on page 5 for example of typical path-name resolving system call. It should be modified to pass additional context into namei().

How do integration tests use HLD components?

Tests have to cover every namei() call-site, which, basically, means, covering every path-name resolving system call.

How do system tests use HLD components?

A coverage check of architecture / HLD.

4.1.4 Logical Specification

- Unique identifier (*operation tag*) is introduced for each „interesting“ namei() call-site (i.e., for each call-site to namei() for which we may wish to handle intents).
- New field is added to struct nameidate to hold operation context. Context at least includes operation tag. For system call, context also includes pointer to @uap argument.
- For each interesting namei() call-site that context field is initialized. For not-interesting call-sites, context get automatically initialized with operation tag 0 („no operation“).
- When VOP_LOOKUP() method of XLL (called by namei()) is called to resolve last component of a path-name, it looks at the context embedded in struct nameidata, and sends RPC to the server to execute intent (together with resolving path-name component).

Locking: all new state is thread-local (and only valid during single kernel entrance).

4.1.5 State Specification

N/A

4.2 Sessions

As mentioned above (4.1 on page 9), XLL performs some meta-data operations while resolving last component of path-name. VOP_OP() method called by VFS after path-name is completely resolved just returns error code, returned by MDS. Additionally VOP_OP() may have to return some additional information (like vnode of newly created object) to VFS. This begs a question of return code and other bits of information are communicated between VOP_LOOKUP() and VOP_OP(). Moreover, as indicated in, for example 5.1.2 on page 22, sometimes XLL needs to store some information in one scope C1 (in C language sense), and retrieve it in another scope C2, where least common parent scope of C1 and C2 does not belong to XLL (and cannot be modified).

4.2.1 Requirements

- A mechanism for storing data during system call execution, and retrieving these data later in the same system call (and, obviously, in the same thread).
- This mechanism should be able to pass information from XLL function F1() to XLL function F2() even when common root of F1() and F2() in the call tree is not XLL code.
- A mechanism to keep track of DLM locks acquired by the current system call, and of portals requests issued by the current system call. These are needed to simplify resource handling: locks and requests are accumulated during system call execution and are released all at once when XLL portion of the system call is just about to finish. In the case of requests this has additional advantage: request body may contain some XLL data-structures (like struct lustre_md) and request can be freed only when its body is known to be no longer used. By releasing all requests at the very end of the system call we can omit that dependency tracking.

One may implement required information-passing by adding some additional state to some common parent scope of C1 and C2. For example, by adding additional automatic variables to every system call entry and passing pointer to these variables all the way down to XLL. This method has severe disadvantages though:

- it requires a lot of changes to XNU core;
- obvious trick of localizing changes by adding new fields to struct nameidata (much like it was done for intents handling) doesn't work, because sometimes XLL code is called by VFS when corresponding nameidata is already out of scope (or doesn't exist at all).

If effect we want something like per-thread per-system-call *dynamic variables* (of LISP), *i.e.*, variables that have temporal rather than lexical scope. (If this doesn't make sense to you, just ignore it.)

4.2.2 Functional Specification

Let's call all data that has to be globally accessible to XLL (bypassing argument passing mechanisms) during execution of a single system call, a *session*. Data-type is introduced to represent session. Instance of this data-type is attached to each thread that executes XLL code during system call (or executes XLL code outside of any system call, *e.g.*, lock revocation call-backs.). As session is per-thread the notion of *current session* is well-defined.

4.2.3 Use Cases

- file creation.

  To handle open("/foo/bar", O_CREAT) VFS first calls namei() routine to resolve path-name. When resolving last component of the path-name ("bar"), XLL executes IT_OPEN intent on the MDS. Server returns (among other things) error code and an id of newly created file. XLL stores them in the current session.

  Later, after return from namei() VFS calls VOP_CREATE(). xll_create() extracts error code from the current session. If there is no error, it extracts id of newly created object from the session, and returns to VFS vnode for it.

- other system calls that require sharing of information between VOP_LOOKUP() and VOP_OP()

  Other cases are handled similarly.

4.2.4 Logical Specification

Nothing to add to the functional specification at the HLD level.

4.3 File Descriptors

4.3.1 Requirements

As was noted above (3.1 on page 4) XNU VFS support for file descriptors is quite different from Linux one:

- file descriptors do not have a field to attach file system specific data to (compare with ->private_data field in Linux struct file);
- file descriptors are visible to VFS only: given user-visible file descriptor number, VFS extracts file descriptor from the per-process file descriptor table, and takes vnode from ->f_data field. When VFS calls VOP methods, it doesn't pass file descriptor as an argument.

On the other hand, Lustre needs access to file descriptor for the following reasons:

- efficient read-ahead. This is not a must though. It is conceivable that read-ahead parameters can be stored in vnode/xnode instead, because file descriptor sharing is probably not widely employed by cluster applications;
- tracking open/close status. As Lustre supports exact UNIX semantics over a cluster, server tracks what files are opened on clients. Client in its turn, tracks when open request is committed on server. Information about this in inherently per-file-descriptor;
- to support LL_FILE_GROUP_LOCKED file locking.

4.3.2 Functional Specification

From requirements, it is clear that the only thing Lustre needs w.r.t file descriptors, is access to the *current* file descriptor, viz. the only file descriptor passed as an argument to the current system call. This notion is well defined, because by exhaustive search of all XNU system calls (bsd/kern/syscalls.c:syscallnames[]) it can be seen that neither of the only two system calls that take more than one file descriptor as an argument (dup2(2) and pipe(2)) invokes VOP methods: the former is completely handled at the VFS layer, and the latter is implemented on top of sockets.

Another complication is that sometimes kernel opens and then uses files for its internal purposes (ktrace(2) system call, swap-files, working set profiles maintained by XNU VM, *etc.*). When opening such files kernel doesn't supplies a file descriptor. Moreover, such internal files can be accessed (by calling VOP_WRITE() for example), while normal write(2) system call is executing, writing into unrelated (or the same as internal!) file.

As a result, following functionality is required:

- ability to efficiently associate some file system specific data with file descriptor;
- mechanism to access current file descriptor;
- said mechanism should be robust in the face of files used internally by kernel as mentioned above.

Alternatively one may change XNU VFS to properly support file descriptors, namely to update prototypes of all relevant VOP_* methods to take file descriptor as an argument. This conflicts with the implied goal of minimizing changes to the XNU core.

4.3.3 Use cases

- write(2):

  read starts with known vnode/xnode. It accesses current file descriptor to check for LL_FILE_IGNORE_LOCK bit in per-file-descriptor flags while taking extent DLM lock.

- read(2):

  in addition to checking for LL_FILE_IGNORE_LOCK, read uses file descriptor to perform read-ahead.

- open(2):

  open associates instance of struct ll_file_data with the current file descriptor.

4.3.4 Logical Specification

Nothing to add to the functional specification at the HLD level.

4.4 Page Cache

4.4.1 Requirements

- minimal changes to XNU core;
- minimal changes to OSC and portals;
- non-requirement: shouldn't be especially scalable, because single Lustre client doesn't cache a lot of data anyway.

4.4.2 Functional Specification

To re-use generic Lustre code in OSC and portals, each platform has to provide implementation of *cfs_page_t* API. This API closely matches Linux page cache interface. To implement it on XNU, intermediate layer (*xll page cache*) is introduced that sits on top of MACH VM and provides data-structures and interfaces similar to ones found in Linux. cfs_page_t API is trivially implemented in terms of xll page cache.

- XLL page cache provides usual abstraction of *objects* composed of fixed-size *pages*. Pages within given object are identified by scalar index. Pair (object, index) uniquely identifies page in the xll page cache.

– Page may be *owned* by some thread.

– Page holds a set of bit-flags. Some of them are maintained by xll page cache implementation (page cache clients have read-only access to such flags), others are set/cleared/read by clients.

– There is an API to wait for given bit-flags combination to be set/cleared.

– Page has a field in which clients store per-page data.

– Page can be temporarily mapped into KVM by client. While page is mapped, client can inspect and/or modify its content.

– There is a *page-iteration* API to execute given call-back function against each page in the given range of indices within given object (additionally API can be instructed to apply call-back only to pages that match given bit-flags combination).

– XLL page cache maintains tunable thresholds on total number of pages and total number of dirty pages in the system. When threshold is crossed, client supplied call-back is invoked on LRU page, until number of pages falls back under appropriate threshold.

4.4.3 Use Cases

read write path. XLL uses page cache to keep files bodies. Each *xnode* of regular file embeds xll page cache object. Pages of this object store file data. Pages within regular file are indexed by logical offset within file. *I.e.,* page with offset n keeps bytes in $[n * CFS\_PAGE\_SIZE, (n+1) * CFS\_PAGE\_SIZE)$ range. Relevant portion of file read looks like following:

```
ssize_t read(struct xnode *x, char *buffer, int size, int offset)
{
        while (size > 0) {
                index = offset / CFS_PAGE_SHIFT;
                find page p for (x->page_cache_object, index) in page cache;
                if (UPTODATE bit-flag is not set in p) {
                        set IO bit-flag in p;
                        call OSC to read page content;
                        wait until IO bit-flag is cleared;
                }
                map p into KVM;
                copy (portion of) p content into buffer;
                unmap p from KVM;
                buffer += bytes-copied;
                size -= bytes-copied;
                offset += bytes-copied;
        }
}
```

portals. To perform IO to and from xll page, portals (NALs) use *cfs_page_t* API to map page into KVM.

DLM lock invalidation. To invalidate extent lock page-iteration API is used to send all dirty pages that are covered by lock in question back to server. Similarly, fsync(2) implementation uses page-iteration API to initiate write-out of all dirty pages of given file.

readdir. Content of directories is also cached in xll page cache. readdir(2) implementation looks quite similar to read above.

XLL-to-OSC interaction. XLL uses per-page field available to clients to attach struct xll_async_page to each page. See io1-filerw.lyx, especially section „Mac OS X Port“. IO completion handler, called by OSC clears IO bit-flag in the page, thus waking all threads that are waiting for IO completion (this is used by read code above).

VM. When MACH VM scanner (3.2 on page 7) decides to evict a range of dirty vnode pages from primary storage, it calls VOP_PAGEOUT() method of vnode. XLL implementation of this method (xll_pageout()) uses page-iteration API to send pages back to server.

4.4.4 Logical Specification

– XLL page cache object contains:

  * a set of *page queues*, together with counter of pages for each queue,
  * a pointer to *page operations vector*,
  * a linkage into global list of all page cache objects.

  Page operations vector is supplied by page cache client, when page cache object is initialized. XLL page cache invokes methods from this vector on certain changes of page state and to perform operations specific to particular *page type*.

  Page belongs to some page queue that depends on bit-flags of this page (*i.e.*, there is page queue for all dirty pages, for all „locked“ pages, *etc.*)

– Page belongs to some page type, determined by page operations vector in page's object (which trivially implies that all pages of a given page cache object belong to the same page type). While XLL page cache provides infrastructure to create/destroy pages, to keep track of them, and to access them efficiently, it doesn't provide mechanism to actually associate some portion of primary storage with the page. It is responsibility of methods in page operations vector to implement such an association (to provide backing storage for pages). Clients are free to add new page types. Below are few representative examples (this is more like use-cases):

  Directory pages. Directories do not normally occupy a lot of primary storage. For that reason, it is possible to use general purpose XNU kernel memory allocator (MALLOC/FREE) to provide backing storage for directory pages. This is very simple page type.

  File pages. File bodies consume more storage than directory pages, plus (and this is more important) we want to get VM memory shortage notification call-backs for them. To this end, page type for file pages is implemented on top of MACH VM *vm_page_t*.

  No-cache pages. As it is known from experience that page cache related defects are numerous and hard to find and fix, beta version of XLL will include an option to run without page cache. This option will be implemented as special page type, that uses some pre-allocated buffer as backing store.

– Page contains:

  * pointer to xll page cache object this page is part of,
  * index within object,
  * set of bit-flags,

    * reference counter: page cache clients only use page after acquiring a reference to it,
    * linkage into page hash table,
    * linkage into object page queue,
    * linkage into LRU list,
    * condition variable that is signalled when page changes its state (when bit-flags on this page are modified),
    * field reserved for use by page clients,
    * field reserved for use by page type implementations.

- Pages are indexed by (object, index) key in a global hash table,
- Notion of page ownership is implemented through special bit-flag,
- Page iteration API is implemented either

    * by checking hash-table for each (object, index) key in given range of indices, or
    * by scanning appropriate object page queue and selecting pages falling into given range of indices

  whichever is estimated to be cheaper. This is important optimization, because sometimes page iteration API is asked to iterate over small range of indices within large object (*e.g.*, while handling VOP_PAGEOUT()), and sometimes to scan very huge ranges (*e.g.*, while invalidating [0, EOF) DLM extent lock).

Locking: To Be Done Later.

4.4.5 State Specification

See DLD.

5 Detailed Level Design

5.1 XLL Intent Handling

5.1.1 Functional Specification

New field is added to struct componentname:

```
struct componentname {
    ...
    /* context high-level operation that invoked namei() */
    struct vfs_op cn_op;
};
/*
 * Description of operation that is invoking name resolution routines.
 */
struct vfs_op {
   /* operation tag */
   enum vfs_op_tag tag;
   /* operation arguments. System calls put uap pointer here. */
   void           *args;
};
```

Possible values of vfs_op->tag field are taken from

```
enum vfs_op_tag {
      VFS_OP_OTHER,
      VFS_OP_OPEN,
      VFS_OP_LINK_SOURCE,
      VFS_OP_LINK_TARGET,
      VFS_OP_UNLINK,
      VFS_OP_CHDIR,
      VFS_OP_MKNOD,
      VFS_OP_CHMOD,
      VFS_OP_CHOWN,
      VFS_OP_OSTAT,
      VFS_OP_OLSTAT,
      VFS_OP_REVOKE,
      VFS_OP_SYMLINK,
      VFS_OP_READLINK,
      VFS_OP_EXECVE,
      VFS_OP_CHROOT,
      VFS_OP_RENAME_SOURCE,
      VFS_OP_RENAME_TARGET,
      VFS_OP_OTRUNCATE,
      VFS_OP_MKFIFO,
      VFS_OP_MKDIR,
      VFS_OP_RMDIR,
      VFS_OP_UTIMES,
      VFS_OP_CHFLAGS,
      VFS_OP_STATFS,
      VFS_OP_UNMOUNT,
      VFS_OP_GETFH,
      VFS_OP_QUOTACTL,
      VFS_OP_MOUNT,
      VFS_OP_MOUNT_DEV,
      VFS_OP_STAT,
      VFS_OP_LSTAT,
      VFS_OP_PATHCONF,
      VFS_OP_GETFSSTAT,
      VFS_OP_TRUNCATE,
      VFS_OP_UNDELETE,
      VFS_OP_ACCESS,
      VFS_OP_FSCTL,
      /* some more obscure XNU things... */
      VFS_OP_COPYFILE_SOURCE,
      VFS_OP_COPYFILE_TARGET,
      VFS_OP_MKCOMPLEX,
      VFS_OP_GETATTRLIST,
      VFS_OP_SETATTRLIST,
      VFS_OP_EXCHANGEDATA_TARGET,
      VFS_OP_EXCHANGEDATA_SOURCE,
      VFS_OP_CHECKUSERACCESS,
```

```
        VFS_OP_SEARCHFS,
        /* not really VFS operations... */
        VFS_OP_SWAPON,
        VFS_OP_SWAPOFF,
        VFS_OP_LOAD_SO,
        VFS_OP_OPEN_PAGE_CACHE_NAMES,
        VFS_OP_OPEN_PAGE_CACHE_DATA,
        VFS_OP_PROFILE_NAMES,
        VFS_OP_PROFILE_DATA,
        VFS_OP_VN,
        VFS_OP_KTRACE,
        VFS_OP_LAST
};
```

Initializer for new fields:

```
#define NDOP(ndp, op, uap) ...
```

New structure:

```
/*
 * Structure used to keep track of intent execution.
 */
struct mdc_lock_ret {
        /* intent that was executed on server */
        struct lookup_intent   it;
        /* request that was used to carry intent to the server */
        struct ptlrpc_request *request;
        /* DLM lock that was acquired as part of intent execution */
        struct xll_lock_info   lock;
        /* meta-data for child object */
        struct lustre_md       md;
        /* session within which intent was executed */
        struct xll_session    *session;
        /* child object (if found) */
        struct vnode          *child;
};
```

New function:

```
int xll_intent_lock(struct xnode *parent,
                    struct xnode *child,
                    struct componentname *name,
                    struct mdc_lock_ret *lock_ret);
```

description resolve path-name component @name in @parent directory, and perform operation indicated in @name->cn_op field.

return negative error code on failure, 0 on success.

parameters

parent parent directory in which lookup is performed

child if non-NULL, child of @parent, having name @name (can be non-NULL only if @child was found in name-cache)

name path-name component to be resolved. @name->cn_op field contains description of intended operation.

lock_ret structure in which results of xll_intent_lock() are accumulated.

5.1.2 Use Cases

System call stat(2) (error handling omitted):

```
int stat(struct proc *p, struct stat_args *uap, register_t *retval) {
        struct stat sb;
        int error;
        struct nameidata nd;
        NDINIT(&nd, LOOKUP,
               FOLLOW | LOCKLEAF | SHAREDLEAF | AUDITVNPATH1,
               UIO_USERSPACE, uap->path, p);
        NDOP(&nd, VFS_OP_STAT, uap); /* initialize operation context */
        namei(&nd);
        vn_stat(nd.ni_vp, &sb, p); /* calls VOP_GETATTR() */
        vput(nd.ni_vp);
        copyout(&sb, uap->ub, STATBUFSIZE);
        return (error);
}
```

namei() calls VOP_LOOKUP() (xll_lookup() function) repeatedly until last path-name component is reached. At this point xll_intent_lock(parent, child, name, &lock_ret) called by xll_lookup() executes INTENT_GETATTR intent on server. If child object was successfully found, its vnode (identified by lock_ret.md) is returned as nd.ni_vp.

When VOP_GETATTR() (xll_getattr()) is called by vn_stat(), target vnode already has DLM lock, and its attributes are valid.

System call unlink(2) (error handling omitted):

```
int unlink(struct proc *p, struct unlink_args *uap, register_t *retval) {
        struct vnode *vp;
        int error;
        struct nameidata nd;
        NDINIT(&nd, DELETE, LOCKPARENT | AUDITVNPATH1,
               UIO_USERSPACE, uap->path, p);
        NDOP(&nd, VFS_OP_UNLINK, uap); /* (1) initialize context */
        namei(&nd);            /* (2) resolve path-name */
        vp = nd.ni_vp;  /* (3) result of path-name resolution.
                           For XLL this is fake vnode (see below). */
        vn_lock(vp, LK_EXCLUSIVE | LK_RETRY, p); /* (4) lock victim vnode */
        /* (4) call VOP_REMOVE(dir, child, name) to remove @child */
        VOP_REMOVE(nd.ni_dvp, nd.ni_vp, &nd.ni_cnd);
        return (error);
}
```

As in stat(2) case above, VOP_LOOKUP() is called until last path-name component is reached. At the moment, VOP_LOOKUP() (xll_lookup()) calls xll_intent_look(). xll_intent_look() determines that there is special handler for current operation: vfs_op_intent_map[VFS_OP_UNLINK]->raw_op() (unlink_op()) and calls it:

```
int unlink_op(struct xnode *parent,
              struct xnode *child,
              struct componentname *name,
              struct xll_session *session, struct mdc_lock_ret *locret) {
        struct ptlrpc_request *request;
        struct mdc_op_data data;
        struct xll_fs *sb;
        int result;
        int isdir;
        /*
         * Are we executing rmdir, or unlink?
         */
        isdir = (name->cn_op.tag == VFS_OP_RMDIR);
        if (is_dot_or_dotdot(name->cn_nameptr, name->cn_namelen) == NAME_IS_DOT) {
                /*
                 * rmdir(".") fails with -EINVAL,
                 *
                 * rmdir("..") will be handled (with -ENOTEMPTY result
                 * hopefully), by md_unlink.
                 */
                return -EINVAL;
        }
        build_mdc_op(&data, parent, NULL, name);
        md_unlink(sb->md_exp, &data, &request);
        if (child != 0)
                cache_purge(XTOV(child));
        if (!isdir) {
                /* for non-directories, do OST part of unlink */
                result = ll_objects_destroy_generic(request, sb->osc_exp);
                /*
                 * Pages in the page-cache will be removed in blocking_ast():
                 * MDS/OST sends appropriate revocation request.
                 */
        }
        /*
         * now fun begins...
         *
         * object was successfully unlinked from the parent. We
         * are currently deep within namei() called by
         * vfs_syscalls.c:unlink(); it expects us to return
         * success, _and_ set nd.ni_vp to something non-zero, to
         * call VOP_REMOVE() against.
         *
         * Let's keep it happy: introduce special dummy vnode
         * specifically for this purpose.
         */
```

```
        locret->child = XTOV(xnode_get_fake());
        session->all_is_done = 1;
        session->uerror = result;
        return result;
}
```

unlink_op() (that handles both unlink and rmdir cases), sends unlink RPC to MDS (and optionally destroys file body on OST), and then returns *fake vnode* to the VFS. To understand why fake vnode is needed look at the point (3) in unlink() code above: VFS uses nd.ni_vp (result of lookup), but XLL already unlinked (and possibly deleted) this object by the time when namei() returns. unlink_op() returns fake vnode, and we are back in unlink() at point (3). Fake vnode is locked at point (4) (which is a no-op), and then VOP_REMOVE() is called. At this point „normal“ file systems actually unlink file: remove entry from the directory, decrement nlink count, *etc.* But for XLL everything is already done by unlink_op(), so xll_unlink() just returns error code, that unlink_op() stored in session->uerror (4.2 on page 12).

System call rename(2). rename(2) is most complex meta-data operation, it resolves two path-names:

```
int rename(struct proc *p, struct rename_args *uap, register_t *retval) {
        struct vnode *tvp, *fvp, *tdvp;
        struct nameidata fromnd, tond;
        int error;
        NDINIT(&fromnd, DELETE, WANTPARENT | SAVESTART | AUDITVNPATH1,
               UIO_USERSPACE, uap->from, p);
        NDOP(&fromnd, VFS_OP_RENAME_SOURCE, uap);
        /* (1) resolve "source" path-name, one that is going to be unlinked */
        namei(&fromnd);
        fvp = fromnd.ni_vp; /* (2) "from" vnode. For XLL this is fake vnode
                               returned by rename_src_op() */
        NDINIT(&tond, RENAME,
               LOCKPARENT | LOCKLEAF | NOCACHE | SAVESTART | AUDITVNPATH2,
               UIO_USERSPACE, uap->to, p);
        NDOP(&tond, VFS_OP_RENAME_TARGET, uap);
        namei(&tond); /* (3) at this point XLL does actual rename */
        tvp = tond.ni_vp; /* (4) "to" vnode */
        /*
         * ... a lot of complicated checks and special cases ...
         */
        /* (5) this is where "normal" file systems do actual rename */
        VOP_RENAME(fromnd.ni_dvp, fvp, &fromnd.ni_cnd,
                   tond.ni_dvp, tvp, &tond.ni_cnd);
        /*
         * ... and even more checks ...
         */
        return error;
}
```

rename(2) has two namei() call-sites tagged by VFS_OP_RENAME_SOURCE and VFS_OP_RENAME_TARGET. For these call-sites xll_intent_lock() uses rename_src_op() and rename_dst_op() handlers respectively:

```
int rename_src_op(struct xnode *parent, struct xnode *child,
                  struct componentname *name,
```

```
                  struct xll_session *session,
                  struct mdc_lock_ret *locret) {
        if (child != NULL)
                /*
                 * expecting rename to be successful, purge
                 * namecache bindings for @child.
                 */
                cache_purge(XTOV(child));
        /* store parent of source object in the session */
        session->u.rename.src_dir  = parent;
        /* store name of source object in the session */
        session->u.rename.src_name = name;
        /* return fake vnode to VFS */
        locret->child = XTOV(xnode_get_fake());
        return 0;
}
```

rename_src_op() is quite simple: it just stores information about last path-name component in the current session and without doing any actual lookup (and without contacting server at all) immediately returns fake vnode to the VFS much like unlink_op() does.

rename_dst_op() is a little bit more involved:

```
int rename_dst_op(struct xnode *parent, struct xnode *child,
                  struct componentname *name,
                  struct xll_session *session,
                  struct mdc_lock_ret *locret) {
        if (session->u.rename.src_dir == NULL ||
            session->u.rename.src_dir->sb != parent->sb)
                /*
                 * if src_dir wasn't filled in the session, last
                 * component of source pathname is not xll, which means
                 * cross-file-system rename is attempted.
                 */
                return -EXDEV;
        }
        oldname = session->u.rename.src_name;
        src = session->u.rename.src_dir;
        build_mdc_op(&data, src, parent, name);
        md_rename(sparent->sb->md_exp, &data, oldname, name, &request);
        ll_objects_destroy_generic(request, parent->sbs->osc_exp);
        session->uerror = result;
        return 0;
}
```

If session->u.rename.src_name is not set when rename_dst_op() is called, rename_src_op() wasn't called in this system call, which means, that rename is crossing file system boundaries (XNU VFS doesn't check for this). Otherwise, source directory and name of source object are extracted from session and RPC is sent to MDS to perform the rename. As in unlink() case, error code from the server is stored in the session, and control returns to the VFS (to point (3) in rename() code above). When VFS calls VOP_RENAME() (xll_rename()), the latter simply returns error code from session.

5.1.3 Logical Specification

```
int xll_intent_lock(struct xnode *parent,
                    struct xnode *child,
                    struct componentname *name,
                    struct mdc_lock_ret *lock_ret);
```

xll_intent_lock() uses auxiliary table vfs_op_intent_map[]:

```
/*
 * Entry describing how intent locking should be done for the particular
 * namei() entry call.
 */
struct map_entry {
      /* value to put into it.it_op for this operation */
      int intent_op;
      /* if this field is non-NULL, it points to the handler function
       * that will complete this operation. This handler is called
       * from VOP_LOOKUP(), it is intended to cover both "raw"
       * operations of Linux port and system calls that issue multiple
       * namei() calls (like rename(2)).
       */
      int (*raw_op)(struct xnode *parent, struct xnode *child,
                    struct componentname *name, struct xll_session *session,
                    struct mdc_lock_ret *locret);
};
static struct map_entry vfs_op_intent_map[VFS_OP_LAST] = { ... };
```

If vfs_op_intent_map[name->cn_op.tag]->raw_op is not NULL (meaning that there is special handler function for that particular namei() call-site), this handler is called and it's return value is return value of xll_intent_lock().

Otherwise (no special handler) xll_intent_lock() simply calls md_lock_get() function (wrapper over md_intent_lock()) that ultimately calls mdc_enqueue() to execute intent on the server.

5.1.4 State Specification

N/A

5.2 Sessions

5.2.1 Functional Specification

Session is represented by struct xll_session:

```
/*
 * XLL SESSIONS. BIG THEORY STATEMENT.
 *
 * struct xll_session represents xll portion of single system call. It
 * serves two purposes:
 *
 * - keeps track of resources associated with this system call:
 *
```

```
 *   . portal requests issued on behalf of this system call;
 *
 *   . DLM locks acquired while executing this system call
 *
 *   . etc.
 *
 * - maps Lustre system call execution model to the VFS one. From the
 * VFS point of view, system call is composed of (roughly) following
 * sequence of the calls to the underlying file system:
 *
 * . VOP_LOOKUP(); VOP_LOOKUP(); ... VOP_LOOKUP() [pathname resolution]
 *
 * . VOP_ACCESS() [optional permission check]
 *
 * . VOP_OP() [actual method performing necessary operation]
 *
 * But xll performs all actual work during last VOP_LOOKUP() call (by
 * sending intent lock request to the server). All following VOP_*
 * calls during this system call are dummies, that should simply
 * return success, except the last one that should return actual
 * result of intent operation back to the user. (Compare this with an
 * approach of Linux Lustre version that uses 'raw' operations as an
 * escapes.) Session allows VOP_* operations to detect when they are
 * dummy-calls.
 *
 * Session life-cycle
 *
 * Session is attached to particular thread (struct uthread). New field
 * ->uu_user_data is used for this purpose.
 *
 * xll_session is allocated dynamically, and reference-counted. As new
 * objects are involved into session, its reference counter is
 * increased. xnodes are detached from session during
 * VOP_UNLOCK(). This works well, because VOP_UNLOCK() (or vput()) is
 * usually at the end of the system call.
 *
 * Session locking
 *
 * Session is only accessed by the thread/process it is attached to, so
 * no locking is necessary.
 *
 */
enum xll_session_state {
        /* new session */
        XLL_SESSION_NEW,
        /* intent operation was performed */
        XLL_SESSION_INTENT,
        /* result was returned to the user */
        XLL_SESSION_DONE
```

```
};
enum xll_session_tag {
        XLL_SESSION_CREATE = 1,
        XLL_SESSION_RENAME = 2
};
/*
 * structure collecting information about "intent lock session".
 */
struct xll_session {
        __u32 magic;
        enum xll_session_state state;
        /* intent operation */
        int op;
        /*
         * DLM locks for this object are placed on the ->locks list below.
         *
         * Also a list of requests executed against this object on behalf of
         * the current system call is in ->requests. These requests are
         * "finished" with when VFS lock is released. This guarantees that
         * pointers to in-request data (MDs, for example) are valid under VFS
         * lock.
         */
        /* meta-data and OS DLM locks in this session */
        struct list_head locks;
        /* portal requests issued on behalf of this session */
        struct list_head requests;
        /*
         * if this is not 0, then, actual operation was already performed by
         * intent lock handler (xll_intent_lock()), and ->uerror contains a
         * error code that should be returned to the user.
         */
        int all_is_done;
        /*
         * user error code. Linux convention: negative.
         */
        int uerror;
        /*
         * discriminator for the union below
         */
        enum xll_session_tag tag;
        /*
         * various pieces of information that are used during system call.
         */
        union {
                /*
                 * information shared by implementation of
                 *
                 * open(..., O_CREAT)
                 */
```

```
                struct {
                        /*
                         * Object created by server while handling open(...,
                         * O_CREAT). Filled in by
                         *
                         * VOP_LOOKUP()->xll_intent_lock()->open_op()->do_open()
                         *
                         * Consumed by VOP_CREATE().
                         */
                        struct xnode *child;
                } create;
                /*
                 * information shared by implementation of rename()
                 */
                struct {
                        /*
                         * Parent directory of object being renamed, and name
                         * of object being renamed. Filled in by
                         *
                         * VOP_LOOKUP()->xll_intent_lock()->rename_src_op()
                         *
                         * Consumed by
                         *
                         * VOP_LOOKUP()->xll_intent_lock()->rename_dst_op().
                         */
                        struct xnode *src_dir;
                        struct componentname *src_name;
                } rename;
        } u;
        /*
         * Value of current_uthread()->uu_user_data at the session
         * entry. Restored when session is deactivated.
         */
        void *parent_user_data;
        /*
         * Reference counter for this session. Session is deallocated when
         * ->ref goes to 0
         */
        int ref_count;
        /*
         * Counter of "extra" references to this session. Extra references are
         * counted in ->ref too. Separate counter exists for better error
         * detection.
         */
        int extra_count;
        /* Super block this session is for */
        struct xll_fs *sb;
};
```

New field is added to struct uthread:

```
struct uthread {
        ...
        /*
         * arbitrary data for subsystem use. It should be maintained as a
         * stack: when subsystem is activated, it should save old value of
         * ->uu_user_data somewhere, before storing its own information here,
         * and restore it back to the original value on
         * exit. --nikita. 2004.06.24
         */
        void *uu_user_data;
};
```

Pointer to current session is stored in ->uu_user_data

Session API:

- Basic session functionality

  `struct xll_session *xll_session_cur(void);`

  description return current session if any, NULL if session doesn't exist yet

  return value current session, or NULL

  `struct xll_session *xll_session_add(struct xll_fs *sb);`

  description if current session doesn't exits, create it.

  return value

  NULL failure to allocate new session

  current-session, either newly created or old one

  parameters

  sb super block which session will be created for

  post-condition `return != NULL => return->sb == sb`

  `void xll_session_get(struct xll_session *session);`

  description acquire additional reference to session

  parameters

  session session to acquire reference to

  post-condition `session'->ref == session->ref + 1`

  `void xll_session_put(struct xll_session *session);`

  description release reference to session. When last reference to session is released, session is destroyed. This includes releasing all DLM locks and portals requests associated with session.

  parameters

  session session to release reference to

  pre-condition `session->ref > 0`

- Extra references

  `void xll_session_get_extra(struct xll_session *session);`

  description acquire extra reference to session

  parameters

  session session to acquire extra reference to

  post-condition `session'->extra_ref == session->extra_ref + 1 && session'->ref == session->ref + 1`

  `void xll_session_put_extra(struct xll_session *session);`

  description release extra reference to session

  parameters

session session to release extra reference to

pre-condition `session->extra_ref > 0 && session->ref > 0`

post-condition `session'->extra_ref == session->extra_ref - 1 && session'->ref == session->ref - 1`

`void xll_session_cleanup(struct xll_session *session);`

description release all extra references

parameters

session session to release all extra references to

post-condition `session'->extra_ref == 0 && session'->ref == session->ref - session->extra_ref`

– Request tracking

`void xll_session_add_request(struct xll_session *session, struct ptlrpc_request *request);`

description attach request to the session. Request will be finished (by ptlrpc_req_finish()) when session is destroyed.

parameters

session session to add request to

request request to add

– Lock tracking

`void xll_session_add_lock(struct xll_session *session, struct xll_lock_info *li);`

description attach DLM lock to the session. Lock will be released (by xll_lock_info_done()) when session is destroyed.

parameters

session session to add lock to

li lock to add to session

`void xll_session_drop_locks(struct xll_session *session);`

description release all locks attached to session

parameters

session session to release all locks of

`void xll_session_drop_mdc_locks(struct xll_session *session);`

description release all MDC locks attached to session (extent locks are not affected)

parameters

session session to release MDC locks of

`int xll_session_has_mdc_locks(void);`

description check whether current session has MDC locks attached

return value

0 current session has no MDC locks attached

!0 current session has at least one MDC lock attached

5.2.2 Use-cases

Code accesses current session as

```
xll_session_cur()
```

For example, xll_lock.c:do_open() uses session as following:

```
...
session->op = IT_OPEN;
...
if (flags & O_CREAT) {
        ...
        session->op |= IT_CREAT;
        ...
}
...
if (result == 0) {
        /*
         * pin session. It will be unpinned in the following
         * VOP_OPEN() (that is guaranteed to be called, as open/create
         * is successful), but see xll_readlink().
         */
        xll_session_get_extra(session);
        ...
}
if (result != +1) {
        /*
         * session is completed, unless we fell back to the "default"
         * path in xll_intent_lock().
         */
        session->all_is_done = 1;
        session->uerror      = result;
}
...
```

5.2.3 Logical specification

Session is attached to new field ->uu_user_data in the struct uthread. Generically speaking, this field can be used by any kernel sub-system to attach some sub-system specific data to the thread. To avoid collisions between different sub-systems, this field is maintained according to the stack discipline: when sub-system is entered, it saves old value of current_uthread()->uu_user_data somewhere and stores pointer to sub-system data there. Before leaving sub-system, old pointer is restored. Debugging code is added to the system calls entry point to check that ->uu_user_data is NULL on system call exit.

Session life-cycle. New session is created in Lustre VOP_LOCK() method (xll_lock()), and destroyed in VOP_UNLOCK() (xll_unlock()). xll_lock() first checks whether there is already existing session for the current thread. If session already exists, it simply increments its reference counter. Similarly xll_unlock() decreased reference counter until is reached 0, at which point session is destroyed. This works OK most of the time, because:

- when VFS calls VOP_ method is locks corresponding vnode. As session is created in VOP_LOCK(), this guarantees that session already exists when VOP_ method is called.
- all vnodes are unlocked before leaving system call, that is session will be automatically destroyed.

But there are few places, where this doesn't work. Sometimes (for example during open) VFS does things like this:

```
VOP_LOCK(vnode1); /* session created here */ vnode2 = VOP_OP1(vnode1); VOP_UNLOCK(vnode1); /* sess
                     released */
/* (1): at this point all vnodes are unlocked */
VOP_LOCK(vnode2); /* new session is created here */
error = VOP_OP2(vnode2);
VOP_UNLOCK(vnode2);
```

But we want the same session to exist during whole system call. To work around such problems, xll_session has additional reference counter ->extra_count, that allows to "pin" session in memory even if all vnodes are locked. In the case above, ->extra_count will be increased by VOP_OP1() and decreased by VOP_OP2().

In addition to error code, session also keeps list of DLM locks and ptlrpc requests. Locks and requests are added to the session during system call and are all released when session is destroyed. This simplifies tracking lock and request life-times.

5.2.4 State Specification

Session invariant (checked by xll_session.c:session_invariant()):

```
session->magic == SESSION_MAGIC && session->extra_count <= session->ref_count
```

5.3 File Descriptors

5.3.1 Functional Specification

New field is added to struct uthread:

```
struct uthread {
        ...
        /*
         * 'current' file: set when system calls operating on file descriptors
         * (fstat(2), read(2), etc.) resolve file descriptor into struct
         * file. Used by file systems that maintain per-file data in
         * file->f_private.
         *
         * XXX nikita: this is a hack to avoid massive changes to the
         * kernel. Proper solution is to pass struct file to the corresponding
         * VOP_* operations as an argument.
         */
        struct file *uu_cur_file;
};
```

New field is added to struct file:

```
struct file {
        ...
        void   *f_private;      /* field for use by vnode/file-system */
};
```

New helper function fdcheck() is added to the XNU core, and fdgetf() function is modified: fdgetf() is called to translate user-visible file descriptor number into file descriptor, and it is modified to store resulting file

descriptor in the ->uu_cur_file field of corresponding thread. In few places where fdgetf() is not called, ->uu_cur_file is set up manually.

New function is added:

`struct file *xnode_get_file(struct xnode *xnode);`

description. return file descriptor that is used to work with @xnode during current system call

return. current file descriptor associated with @xnode, or NULL if there is no one

parameters.

xnode xnode to return current file descriptor for. xnode parameter is necessary to distinguish accesses to files that kernel opened internally.

post-condition `result != NULL => result == current_uthread()->uu_cur_file && result->f_data == XTOV(xnode)`

5.3.2 Use cases

– read(2):

```
int xll_read(struct xnode *xnode, struct uio *uio) {
        struct file  *file;

       if (uio->uio_segflg == UIO_USERSPACE ||
           uio->uio_segflg == UIO_USERISPACE) {
               file = xnode_get_file(xnode);
       } else
               /*
                * this read was initiated by kernel on its own behalf (from
                * kern/kern_exec.c:execve() for example).
                *
                * In this case there is no file descriptor.
                */
               file = NULL;
      /*
       * xll_read_file() analyzes
       *
       * file->f_private->fd_flags && LL_FILE_IGNORE_LOCK
       *
       * while taking extent lock, and uses file->f_private->fd_ras to
       * read-ahead.
       */
      xll_read_file(file, xnode, uio);
}
```

– open(2):

```
int local_open(struct xnode *xnode, int flags, struct ptlrpc_request *req) {
        struct ll_file_data *fd;
        struct mds_body     *body;
        struct file         *file;
```

```
        struct xll_readahead_state *ras;
        file = current_uthread()->uu_cur_file;
        if (file == NULL)
                /*
                 * VOP_OPEN() called without existing file descriptor. Most
                 * likely, this 'open' is for internal kernel usage (ktrace,
                 * for example). In this case VOP_CLOSE() will also be called
                 * without file descriptor, so everything should be okay.
                 */
                return 0;
        XLASSERT(file->f_data == NULL);
        XLASSERT(file->f_private == NULL);
        fd = _MALLOC(sizeof *fd, M_TEMP, M_WAITOK | M_ZERO);
        memcpy(&fd->fd_mds_och.och_fh,
               &body->handle, sizeof body->handle);
        fd->fd_mds_och.och_magic = OBD_CLIENT_HANDLE_MAGIC;
        file->f_private = fd;
        ras = &fd->fd_ras;
        xll_readahead_state_init(ras);
        xnode->io_epoch = body->io_epoch;
        md_set_open_replay_data(&fd->fd_mds_och, req);
        ...
}
```

5.3.3 Logical specification

- Current file descriptor is stored in current_uthread()->uu_cur_file by fdgetf();
- ->f_private field of struct file is used by XLL to store pointer to struct ll_file_data.

5.4 Page Cache

5.4.1 Functional Specification

Few scalar types used by xll page cache:

```
typedef __u32 nr_pages_t;
typedef __u64 nr_bytes_t;
typedef __u32 page_off_t;
typedef __u64 byte_off_t;
enum {
     /*
      * There is one queue and one counter for each possible combination of
      * lowest 3 bits in page->flags (XLL_PAGE_{BUSY,DIRTY,IO}). This is
      *
      *     - for simplicity
      *
      *     - to avoid a loss of generality early.
      *
      * This can be changed (queues and counters can be "aggregated") by
```

```
        * changing flags_to_counts() and flags_to_queue() functions.
        */
        /* number of page counters in xll_cache and xll_domain */
        XLL_COUNT_NR = 8,
        /* number of page queues in xll_cache */
        XLL_QUEUE_NR = 8
};
struct xll_cache;
struct xll_page;
```

struct xll_compartment represents xll page cache as a whole.

```
/*
 * Represents collection of xll_cache's and xll_page's that are scope of
 * balancing algorithm: xll_balance_{dirty,pages}() maintains thresholds
 * within single compartment. There (potentially) may be multiple compartments
 * in the system (e.g., one per Lustre mount), but currently only one is used:
 * xll_default_compartment.
 */
struct xll_compartment {
        /* protects access to other fields */
        spinlock_t        guard;
        /* number of xll_cache's in this compartment */
        unsigned          nr_caches;
        /* total number of pages*/
        nr_pages_t        nr_pages;
        /* dirty pages threshold. Used by xll_balance_dirty(). */
        nr_pages_t        max_dirty;
        /* total pages threshold. Used by xll_balance_pages(). */
        nr_pages_t        max_pages;
        /* per-queue page counters */
        nr_pages_t        count[XLL_COUNT_NR];
        /* global LRU page list */
        struct list_head  lru;
        /* list of all caches in this compartment */
        struct list_head  caches;
};
extern struct xll_compartment xll_default_compartment;
```

Page operations vector:

```
/*
 * Page operations vector.
 */
struct xll_cache_ops {
   /* called when new page of this page-type is created */
   int   (*page_init)      (struct xll_page *);
   /*
    * called when page of this page-type is just about to be destroyed
    * (page_recycle())
```

```
     */
    void  (*page_done)      (struct xll_page *);
    /* called to map page into KVM */
    void *(*page_map)       (struct xll_page *);
    /* called to unmap page from KVM */
    void  (*page_unmap)     (struct xll_page *);
    /*
     * returns virtual address of page content. Can only be called on
     * mapped pages.
     */
    void *(*page_address)   (struct xll_page *);
    /*
     * called (by xll_page_del()) when page cache client decides it does not
     * want this page in cache (e.g, truncate or extent lock
     * invalidation).
     */
    void  (*page_invalidate)(struct xll_page *);
    /*
     * should initiate process of pushing content of page to the secondary
     * storage if necessary (e.g., sending dirty page to the OST).
     */
    int   (*page_out)       (struct xll_page *);
    /*
     * moves given number of bytes, starting from given offset within page
     * between page and uio.
     *
     *         page->cache->ops->page_move(page, offset, bytes, uio);
     *
     * has to be identical to
     *
     *         page->cache->ops->page_map(page);
     *         uiomove(page->cache->ops->page_address(page) + offset,
     *                 bytes, uio);
     *         page->cache->ops->page_unmap(page);
     *
     * modulo error checking. This method exists for the sake of possible
     * optimizations like cluster_copy_ubc_data() that can avoid
     * installation of temporary KVM mappings.
     */
    int   (*page_move)      (struct xll_page *, int, int, struct uio *);
    /* called when page reference counter goes from 0 to 1 */
    void  (*page_load)      (struct xll_page *);
    /* called when page reference counter goes from 1 to 0 */
    void  (*page_store)     (struct xll_page *);
};
```

XLL page cache object is represented by struct xll_cache:

```
/* bit-flags stored in xll_cache->flag field */
enum xll_cache_flag {
```

```
        /*
         * if this flag is set pages in xll_cache in question are kept in LRU
         * order on xll_cache->lru rather than on global
         * xll_cache->domain->lru.
         */
        XLL_CACHE_LOCAL_LRU = (1 << 0)
};
/*
 * XLL page cache object: a collection of related pages. Usually xll_cache
 * represents portion of client object (regular file, or directory) cached in
 * the primary storage.
 *
 * Pages within given page cache object belong to the same page type (have the
 * same page operations vector page->cache->ops), have the same size (1 <<
 * page->cache->page_shift), and are uniquely identified by page->index.
 */
struct xll_cache {
     /* flags taken from enum xll_cache_flag */
     __u32                   flags;
     /* page operations vector */
     struct xll_cache_ops   *ops;
     /* compartment this object is part of */
     struct xll_compartment *domain;
     /* log_2 of page size in this object */
     unsigned                page_shift;
     /* protects the rest of fields */
     spinlock_t              guard;
     /* total number of pages in this object */
     nr_pages_t              nr_pages;
     /* ->count[i] is number of elements in ->queue[i] */
     nr_pages_t              count[XLL_COUNT_NR];
     /*
      * page queues. Each page is in some queue in its object. Page queue
      * depends on bit-flags of this page (page->flags), and is
      * determined by flags_to_queue().
      */
     struct list_head        queue[XLL_QUEUE_NR];
     /*
      * local LRU list. Used if ->flags & XLL_CACHE_LOCAL_LRU
      */
     struct list_head        lru;
     /* linkage into list of all page cache objects within compartment */
     struct list_head        list;
};
```

And finally XLL page is represented by struct xll_page:

```
/*
 * Page flags.
 */
```

```
enum xll_page_flag {
     /* page is owned by some thread */
     XLL_PAGE_BUSY          = (1 << 0),
     /* page is dirty, write-back is required */
     XLL_PAGE_DIRTY         = (1 << 1),
     /*
      * IO is in progress. Only page owner can set this bit. It can be
      * cleared by io completion handler.
      *
      * Note: this bit is set when page is actually heading for IO without
      * any further delay. Do NOT set this bit when queuing asynchronous
      * OBD IO: page will linger in cache until there is enough dirty pages
      * to form a RPC, and other threads will bump into its XLL_PAGE_IO
      * bit.
      */
     XLL_PAGE_IO            = (1 << 2),
     /* page content was read from the storage */
     XLL_PAGE_UPTODATE      = (1 << 3),
     /*
      * page->cache->op->init_page() method was successfully ran against
      * this page
      */
     XLL_PAGE_INIT          = (1 << 4),
     /*
      * page will be destroyed when last reference to it released. This is
      * set during cache invalidation (on truncate, or object
      * destruction). Only page owner can set this bit, and it's never
      * cleared once set.
      */
     XLL_PAGE_HEARD_BANSHEE = (1 << 5),
     /*
      * page-io operation failed. This is set in xll_io.c:completion() and
      * analyzed after waiting for output completion.
      *
      * Not used for directory pages---they are never paged-out.
      */
     XLL_PAGE_ERROR         = (1 << 6)
};
struct xll_page {
      /* header used as an anchor by cfs_page_t API */
      struct xnu_page   header;
      /*
       * ->cache and ->index are protected by BUSY bit and hash lock (see
       * xll_cache_del()).
       */
      struct xll_cache *cache;
      /* unique index within ->cache */
      page_off_t        index;
      /*
```

```
         * ->ref protects page existence, but not its identity.
         */
        unsigned          ref;
        /* from xll_page_flag */
        __u32             flags;
        /* hash table linkage. Protected by global hash-table spin-lock */
        struct list_head  hash;
        /*
         * appropriate list (dirty, busy, etc.) hanging off
         * page->cache. Protected by page->cache->guard spin-lock.
         */
        struct list_head  list;
        /*
         * Global (or local) LRU list. Protected by page->cache->domain->guard
         * spin-lock.
         */
        struct list_head  lru;
        /*
         * condition variable signaling changes to the page state (IO
         * completion, etc.). If this is too expensive, hashing can be used as
         * in Linux.
         */
        struct kcond      cond;
        /*
         * ->data and ->data2 are reserved for use by page operations vector.
         */
        void              *data;
        void              *data2;
        /*
         * for private use of page cache client. Usually pointer to struct
         * xll_async_page.
         */
        void              *private;
};
```

XLL page cache API:

- General bits: references, ownership, page life-cycle. Invariants:
  - page is owned => current thread has a reference to it;
  - only owner can initiate IO;
- XLL page cache objects handling

  `void xll_cache_init(struct xll_cache *cache);`

  description. initialize all fields in @cache. Called by page cache clients for every new xll page cache object they introduce.

  return value. none

  parameters.

  cache xll page cache object to be initialized

`void xll_cache_done(struct xll_cache *cache);`

description. finalize fields of @cache. Called by page cache clients just before freeing page cache object they created.

return value. none

parameters.

cache xll page cache object to finalize.

`void xll_compartment_add(struct xll_compartment *appt, struct xll_cache *cache);`

description. add xll page cache object into page cache compartment. From this moment on @cache will participate in page cache balancing.

return value. none

parameters.

appt compartment to add @cache to. Usually &xll_default_compartment.

cache object to add to @appt.

`void xll_compartment_del(struct xll_cache *cache);`

description. remove xll page cache object from the page cache compartment is belonged to.

return value. none.

parameters.

cache page cache object to be removed from the compartment.

– XLL page ownership

`struct xll_page *xll_page_get(struct xll_cache *cache, page_off_t index);`

description. return page with index @index in the page cache object @cache. Page is created, if necessary. Page is always returned referenced and owned by the calling thread. Page is always returned initialized. If IO was underway on page, xll_page_get() returns only after IO completion.

return value. page on success, NULL on failure.

post-condition.

```
page != NULL => xll_page_is_mine(page) && page->cache == cache && page->index == index &&
                page->flags & XLL_PAGE_INIT &&
                !(page->flags & XLL_PAGE_IO) && !(page->flags & XLL_PAGE_HEARD_BANSHEE)
```

parameters.

cache page cache object in which page is looked for and/or created

index index of target page

`struct xll_page *xll_page_get_try(struct xll_cache *cache, page_off_t index);`

description. return page with index @index in the page cache object @cache if it wouldn't involve blocking (*i.e.*, if page is neither owned nor under IO). Page is created if necessary. Page is always returned referenced and owned by the calling thread. Page is always returned initialized.

return value. page on success, NULL if page was busy or under IO. NULL on failure.

post-condition.

```
page != NULL => xll_page_is_mine(page) && page->cache == cache && page->index == index &&
                page->flags & XLL_PAGE_INIT &&
                !(page->flags & XLL_PAGE_IO) && !(page->flags & XLL_PAGE_HEARD_BANSHEE)
```

parameters.

cache page cache object in which page is looked for and/or created

index index of target page

`struct xll_page *xll_page_look(struct xll_cache *cache,`
`page_off_t index, int try);`

description. return page with index @index in the page cache object @cache if it exists already. Page is always returned referenced and owned by the calling thread. Page is always returned initialized. If @try is set, then page will only be returned if it is neither owned nor under IO.

return value. page on success. NULL if @try is set and page was owned or under IO. NULL on failure.

post-condition.

```
page != NULL => xll_page_is_mine(page) && page->cache == cache && page->index == index &&
                page->flags & XLL_PAGE_INIT &&
                !(page->flags & XLL_PAGE_IO) && !(page->flags & XLL_PAGE_HEARD_BANSHEE)
```

parameters.

cache page cache object in which page is looked for

index index of target page

try if 0, xll_page_look() is allowed to block to wait for page ownership or IO completion.

`void xll_page_put(struct xll_page *page);`

description. release page reference and page ownership. This function should be called when client is done with the page obtained through xll_page_{get,try,look}().

return value. none

pre-condition.

```
xll_page_is_mine(page) && page->ref_count > 0
```

post-condition.

```
page'->ref_count == page->ref_count - 1 && !xll_page_is_mine(page') &&
(page->ref_count > 1 => page'->cache == page->cache && page'->index == page->index)
```

parameters.

page page to be released.

`int xll_page_own_try(struct xll_page *page);`

description. attempt to acquire page ownership non-blockingly. Caller should already has a reference to @page. *If this function fails to acquire ownership on a page it will release one reference to page*. (It may sounds strange that function releases reference it didn't acquire, but this is very convenient in practice.)

return value.

0 page ownership was acquired successfully.

-EBUSY page is owned by some thread, or IO is underway against it.

-EAGAIN page is being destroyed right now.

pre-condition.

```
page->ref_count > 0
```

post-condition.

```
result == 0 => (xll_page_is_mine(page') && !xll_page_is_mine(page))
```

parameters.

page page to attempt to acquire ownership of.

`void xll_page_del(struct xll_page *page);`

description. discard page from the page cache. Indicate that page *heard banshee*. Such page will be thrown out of cache and destroyed as soon as last reference to it is released. Ownership cannot be acquired on such page. xll_page_{get,try,look}() will never return such page.

return value. none

pre-condition.

```
xll_page_is_mine(page) && page->ref_count > 0
```

post-condition.

```
xll_page_is_mine(page') && (page'->flags & XLL_PAGE_HEARD_BANSHEE)
```

parameters.

page `victim page`

– XLL page bit-flags handling

`void xll_page_bit_while(struct xll_page *page, int mask);`

description. wait until at least one bit-flag in @mask is cleared for @page

return value. none

post-condition.

```
(page'->flags & mask) != mask /* modulo concurrency */
```

parameters.

page page to wait on
mask bit-flags to wait for

`void xll_page_bit_until(struct xll_page *page, int mask);`

description. wait until all bit-flags in @mask are set for @page
return value. none
post-condition.

```
(page'->flags & mask) == mask /* modulo concurrency */
```

parameters.

page page to wait on
mask bit-flags to wait for

`void xll_page_bit_wait_and_set(struct xll_page *page, int mask);`

description. wait until at least one bit-flag in @mask is cleared for @page, and then set bit-flags from @mask atomically.
return value. none
post-condition.

```
(page'->flags & mask) == mask /* modulo concurrency */
```

parameters.

page page to wait on, and set bit-flags on.
mask bit-flags to wait for and set.

`void xll_page_bit_set(struct xll_page *page, int mask);`

description. set bit-flags from @mask for @page
return value. none
post-condition.

```
(page'->flags & mask) == mask /* modulo concurrency */
```

parameters.

page page to set bit-flags on
mask bit-flags to set

`void xll_page_bit_clear(struct xll_page *page, int mask);`

description. clear bit-flags from @mask for @page
return value. none
post-condition.

```
(page'->flags & mask) == 0 /* modulo concurrency */
```

parameters.

page page to clear bit-flags on
mask bit-flags to clear

`int  xll_page_bit_try_set(struct xll_page *page, int mask);`

description. set bit-flags from @mask for @page and return old values of bit-flags from @mask for @page
return value. old values of bit-flags from @mask for @page
post-condition.

```
(page'->flags & mask) == mask /* modulo concurrency */ &&
return == (page->flags & mask)
```

parameters.

page page to set bit-flags on
mask bit-flags to set

`void xll_page_bit_set_raw(struct xll_page *page, int mask);`

description. set bit-flags from @mask for @page, without moving @page between page cache object page queues. *Use with care*.

return value. none

post-condition.

```
(page'->flags & mask) == mask /* modulo concurrency */
```

parameters.

page `page to set bit-flags on`

mask `bit-flags to set`

`void xll_page_bit_clear_raw(struct xll_page *page, int mask);`

description. clear bit-flags from @mask for @page, without moving @page between page cache object page queues. *Use with care*.

return value. none

post-condition.

```
(page'->flags & mask) == 0 /* modulo concurrency */
```

parameters.

page `page to clear bit-flags on`

mask `bit-flags to clear`

– low-level xll_page locking and reference counter manipulation

`void xll_page_acquire(struct xll_page *page);`

description. acquire *additional* reference to @page.

return value. none

pre-condition.

```
page->ref > 0
```

parameters.

page `page to acquire reference to`

`void xll_page_release(struct xll_page *page);`

description. release reference to @page

return value. none

pre-condition.

```
page->ref > 0 && hash_is_not_locked()
```

parameters.

page `page to acquire reference to`

`void xll_page_lock(struct xll_page *page);`

description. take @page spinlock (page->guard)

return value. none

parameters.

page `page to acquire reference to`

`void xll_page_unlock(struct xll_page *page);`

description. release @page spinlock (page->guard)

return value. none

parameters.

page `page to acquire reference to`

– page-iteration API

```
/*
 * tells page-iteration API what to do.
 */
```

```
struct xll_cache_iterator {
    /*
     * call-back to be executed against eligible pages. Page is owned by
     * calling thread when call-back is executed.
     */
    int    (*actor)(struct xll_page *, void *);
    /* cookie passed to ->actor as a second argument */
    void    *key;
    /*
     * if this is non-0, xll_cache_apply() will silently skip pages that
     * are either owned or under IO.
     */
    int      trylock;
    /* bit-flags that have to be set on eligible pages */
    unsigned set;
    /* bit-flags that have to be clear on eligible pages */
    unsigned unset;
};
int xll_cache_apply(struct xll_cache *cache, page_off_t start, nr_pages_t nr,
                    struct xll_cache_iterator *it);
```

description. call it->actor(page, it->key) for all *eligible* pages in @cache. Page is eligible if

- its index is in $[start, start + nr)$ range, and
- it has all bit-flags from it->set set, and
- it has all bit-flags from it->unset clear, and
- if it->trylock is non-0, page is neither owned nor under IO.

callback is called with page owned by calling thread. Iteration stops when callback returns non-0.

return value.

0 iteration completed successfully

-EBUSY it->trylock is set and iteration reached a page owned by some thread, or under IO

-EAGAIN iteration reached a page that being destroyed right now.

<another value> it->actor returned non-0.

parameters.

cache `page cache object to iterate in`

start starting index of range to iterate over

nr number of pages in the range to iterate over

it description of eligible pages, and call-back

– Standard page-iterators.

```
int xll_cache_invalidate(struct xll_cache *cache,
                         page_off_t start, nr_pages_t nr);
```

description. invalidate all pages in $[start, start + nr)$ range of indices (by calling xll_page_del() on each of them)

return value. result of iteration (FIXME: presumably 0).

parameters.

cache `page cache object to iterate in`

start starting index of range to iterate over

nr number of pages in the range to iterate over

```
int xll_cache_truncate(struct xll_cache *cache, page_off_t start);
```

description. truncate page cache object @cache up to (and excluding) page at index @start. Equivalent to call

```
xll_cache_invalidate(cache, start, EOF - start);
```

return value. result of iteration (FIXME: presumably 0).

parameters.

cache `page cache object to iterate in`

start starting index of range to iterate over

`int xll_page_out(struct xll_cache *cache, page_off_t start, nr_pages_t nr, int try);`

description. initiate page-out of all pages in range $[start, start + nr)$ (by calling xll_page_out() on each of them)

return value. result of iteration

parameters.

cache `page cache object to iterate in`

start starting index of range to iterate over

nr number of pages in the range to iterate over

try if non-0, skip owned pages and pages under IO.

`int xll_io_wait(struct xll_cache *cache, page_off_t start, nr_pages_t nr);`

description. wait until all IO in the range $[start, start + nr)$ completes.

return value. 0.

parameters.

cache `page cache object to iterate in`

start starting index of range to iterate over

nr number of pages in the range to iterate over

– Page counters

`nr_pages_t xll_count_pages(nr_pages_t *counts, unsigned set, unsigned unset);`

description. generic (some would say, over-generic) interface to count pages in the queues. xll_pages are stored in queues hanging of xll_cache. There is a queue for each possible combination of first 3 bits in page->flags: XLL_PAGE_BUSY, XLL_PAGE_DIRTY, and XLL_PAGE_IO. xll_cache and xll_compartment keep counters of pages contained in each queue. xll_count_pages() calculates total number of pages contained in some subset of queues, specifically queues corresponding to some bits set in page->flags and some—unset.

return value. total number of pages in matching queues.

parameters.

counts array of page counts (usually xll_cache->count, or xll_compartment->count)

set only queues containing pages with all bit-flags from @set set are matched

unset only queues containing pages with all bit-flags from @set clear are matched

`nr_pages_t xll_dirty_pages(nr_pages_t *counts);`

description. return number of dirty pages in queues @counts. This call is equivalent to

```
xll_count_pages(counts, XLL_PAGE_DIRTY, 0);
```

return value. total number of dirty pages in queues.

parameters.

counts array of page counts (usually xll_cache->count, or xll_compartment->count)

– Page cache balancing

`void xll_balance_dirty(struct xll_compartment *appt);`

description. invoke page cache balancing for compartment @appt. If number of dirty pages in @appt is above threshold (appt->max_dirty), pages are paged-out in LRU order, until compartment returns under threshold.

return value. none

parameters.

appt compartment to balance

`void xll_balance_pages(struct xll_compartment *appt);`

description. invoke page cache balancing for compartment @appt. If total number of pages in @appt is above threshold (appt->max_pages), pages are invalidated in LRU order, until compartment returns under threshold.

return value. none
parameters.

appt compartment to balance

– Helpers

`void *xll_page_map(struct xll_page *page);`
description. map page into KVM
return value. address at which page was mapped
parameters.

page page to map

`void  xll_page_unmap(struct xll_page *page);`
description. unmap page from KVM
return value. none
parameters.

page page to unmap

`void *xll_page_address(struct xll_page *page);`
description. return address of mapped page
return value. address of mapped page
parameters.

page mapped page to return address of

`void xll_page_set(struct xll_page *page, byte_off_t s, int c, nr_bytes_t b);`
description. fill bytes in $[s, s + b)$ byte range within page with value @c. This call is equivalent to

```
page->cache->ops->page_map(page);
memset(page->cache->ops->page_address(page) + s, c, b);
page->cache->ops->page_map(page);
```

return value. none
parameters.

page page to be filled
s starting offset of region to fill
c value to fill region with
b number of bytes in region to fill

`byte_off_t xll_page_offset(struct xll_page *page);`
description. return byte offset of page starting from beginning of page cache object
return value. byte offset
parameters.

page page to return byte offset of

`int xll_page_size(struct xll_page *page);`
description. return size of page in bytes
return value. size of page in bytes
parameters.

page page to return size of

`cfs_page_t *xll_page_to_cfs(struct xll_page *page);`
description. convert XLL page cache page into portable cfs_page_t page.
return value. pointer to portable cfs_page_t object representing the same page.
parameters.

page page to convert

5.4.2 Concurrency control of page cache data-structures.

XLL page cache uses following synchronization objects:

- global hash_lock protected page cache hash table
- spin lock in struct xll_page
- spin lock in struct xll_cache
- spin lock in struct xll_compartment
- condition variable in struct xll_page
- XLL_PAGE_BUSY bit-flag in page flags used to synchronize page ownership
- ->busy bit in MACH VM page (vm_page_t) used by POP-page-type to synchronize

Lock nesting:

- XLL_PAGE_BUSY (outermost lock, all other locks are taken after this one)
- ->busy
- hash-lock
- page-spin-lock
- cache-spin-lock
- compartment-spin-lock (innermost lock)

Page condition variable is paired with page spin lock.
Things protected by locks (also see comments in data-type definitions in Functional Specification above):

- compartment spin lock:
  - all fields in struct xll_compartment
  - ->lru field in struct xll_page for pages that belong to this compartment (even, for pages that belong to page cache object that uses local LRU (page->cache->flags & XLL_CACHE_LOCAL_LRU))
  - ->list field of struct xll_cache for page cache objects that belong to this compartment
- cache spin lock
  - all fields of struct xll_cache, except ->list
  - ->list field of pages in this cache
- page spin lock
  - ->flags field of struct xll_page
  - ->ref field in case when acquiring *additional* reference to the page
  - page condition variable internals
- hash_lock

    - global hash table consistency
    - ->hash field in struct xll_page

- XLL_PAGE_BUSY (page ownership)

    - XLL_PAGE_IO bit (only page owner is allowed to start IO against this page)
    - XLL_PAGE_INIT bit (only page owner can initialize this page)
    - calls to ->page_load() method of page operations vector
    - calls to ->page_invalidate() method of page operations vector

- ->busy bit in vm_page_t

    - serializes with VM scanner. POP page operations vector (xll_io.c:pop_page_*() functions) sets ->busy bit in the underlying MACH page in ->page_load() (pop_page_load()) and clears it in ->page_store() (pop_page_store()). This means that page is accessible to MACH VM only while in NO-REFS multi-state (see State Specification).

- page condition variable

    - is signalled when page->flags changes

- XLL_PAGE_BUSY *and* hash_lock

    - page->cache and page->index fields
    - XLL_HEARD_BANSHEE bit (but this bit is never cleared once set)

- page spin lock *and* hash_lock

    - transition of page->ref from 0 to 1 (xll_page.c:page_acquire_zero()). Transition of ->ref from 1 to 0 is not protected by hash lock, because
        - if (page->flags & XLL_PAGE_HEARD_BANSHEE), then page was already removed from hash table and page queues by cache_del(). This means, that when xll_page_release() releases last page reference there is absolutely no way for any other thread to obtain new reference for this page. Existing references will do no harm, because most operations (including acquiring page ownership) will fail with -EAGAIN when invoked on a page that heard banshee.
        - if page didn't hear banshee, 1->0 transition doesn't touch hash table in any way.

Differences with Linux page cache:

- different bit-flags (XLL_PAGE_BUSY, and XLL_PAGE_IO) are used to synchronize page ownership and IO. In Linux kernel 2.4 both are covered by PG_locked bit. In 2.6 kernels page ownership and read are covered by PG_locked, and write is covered by PG_writeback bit. OSC layer assumes Linux behavior in some places. See comment in xll_io.c:issue_page_read().

5.4.3 Page life cycle

Page is created by call to one of xll_page_{get,try,look}() functions. Page is inserted into hash table, and ->page_init() method from page operations vector (specified by struct xll_cache parameter) is invoked. Page is returned to the caller, owned by calling thread (with XLL_PAGE_BUSY bit-flag set) and referenced (page->ref == 1).

Caller is free to do whatever it wants with the page:

- to acquire additional references to this page, and to place pointer to it into some data-structures;
- to start IO on a page. This involves getting additional reference to the page, and setting XLL_PAGE_IO bit-flag; Page cache clients (mostly file system) use XLL_PAGE_UPTODATE and XLL_PAGE_DIRTY bits to track page state and to determine whether IO is needed. Note: XLL_PAGE_IO bit is only set when page is actually submitted for IO. This bit should *not* be set when page is queued for IO at some later time (see comment for XLL_PAGE_IO bit in enum xll_page_flag definition in Functional Specification above (5.4.1 on page 37)). Otherwise, suppose that XLL_PAGE_IO is set when page is queued for IO. IO wouldn't start until enough pages are collected in the page cache to form large enough RPC to the server. But other threads may block on XLL_PAGE_IO, hence no new dirty pages will be added to the page cache leading to deadlock. This means, in particular, that write path (that calls obd_queue_async_io() in xll_page.c:page_submit()) doesn't set XLL_PAGE_IO itself. XLL_PAGE_IO is set by ->make_ready() call-back called by OSC. As only page owner is allowed to set XLL_PAGE_IO bit, this, in turn, means that ->make_ready() call-back has to acquire page ownership. As ->make_ready() maybe called synchronously from within call to obd_queue_async_io(), ->make_ready() cannot wait for page ownership (as this would self-deadlock calling thread), and has to call xll_page_own_try() to *attempt* to get page ownership, and fails with -EAGAIN on failure. Which is, conincidentiallycoincidently, exactly what Linux ->make_ready() call-back does, but for entirely different reasons.
- to decide that page should be destroyed. For example, when extent lock is invalidated, all pages covered by this lock have to be thrown out of the cache. Page is destroyed by call to xll_page_del(). This function sets XLL_HEARD_BANSHEE bit on a page (and it is said that page *heard banshee*).
- to release page ownership and initial reference to a page. This is done by calling xll_page_put().

When IO started by page owner finishes, completion handler (supplied to OSC by client) wakes up threads waiting on XLL_PAGE_IO bit, clears this bit, and releases reference acquired during IO startup. If IO wasn't successful, XLL_PAGE_ERROR bit is set.

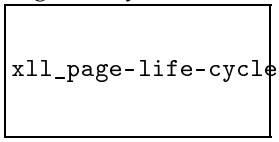
When last reference to page is released (and it is guaranteed that page has no owner at this point, because ownership is always acquired together with page reference), XLL_PAGE_HEARD_BANSHEE bit is analyzed. If this bit is set, page is immediately destroyed. Otherwise it is left in the cache, after calling ->page_store() method of page operations vector.

Later page may be found in the hash-table by call to xll_page_{get,try,look}(). Alternatively, page may be reached by calling page-iteration API or internal LRU-scanning API.

Page-iteration API, described in the Functional Specification above (5.4.1 on page 42) accesses pages by scanning page queues in page cache object.

LRU-scanning API (xll_page.c:xll_lru_apply()) iterates over „global" LRU list in xll_compartment, and „local" LRU lists in all page cache objects in this compartment. For each page it executes call-back function much like page-iteration API does.

Page-life cycle, as a result looks like



5.4.4 Use-cases

– write(2) implementation. Relevant portion of write looks like

```
int write_file_locked(struct xnode *xnode, struct uio *uio) {
       while (uio->uio_resid > 0 && result == 0) {
                byte_off_t offset; /* offset within a page */
                page_off_t index;  /* page index */
                nr_bytes_t bytes;  /* bytes to write on a page */
                struct xll_page *page;
                index = uio->uio_offset >> CFS_PAGE_SHIFT;
                offset = uio->uio_offset & ~CFS_PAGE_MASK;
                bytes = xll_min(byte_off_t,
                                CFS_PAGE_SIZE - offset, uio->uio_resid);
                /* re-use existing page, or create new one */
                page = xll_page_get(&xnode->cache, index);
                /* if necessary, read portions of page */
                page_prep(page, offset, bytes);
                /* move data from user-level buffer (specified by @uio) to the
                   page */
                page->cache->ops->page_move(page, offset, bytes, uio);
                /* queue IO against this page (obd_queue_async_io()) */
                page_submit(page, offset, bytes);
                /* drop ownership on this page, and release reference on it */
                xll_page_put(page);
                /* get page cache a chance to balance amount of dirty pages */
                xll_balance_dirty(page->cache->domain);
        }
}
```

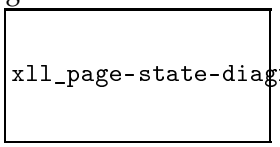
– read-ahead

```
int readahead(struct xnode *xnode, ...) {
        struct xll_page  *page;
        for (...; want-more-read-ahead;
             ++ i, (page != NULL ? xll_page_put(page) : (void)0)) {
                /*
                 * Linux version calls grab_cache_page_nowait() that try-locks
                 * a page. In our XNU page cache, page ownership and io are
                 * controlled by separate page->flags bits (XLL_PAGE_BUSY and
                 * XLL_PAGE_IO). readahead() does _not_ accumulate ownership
                 * of pages it processes: xll_page_put() is called at the end
                 * of each iteration. This means, that we can (conceivably)
                 * wait for page ownership here. But,
                 *
                 *     (1) we don't want to wait for completion of io that may
                 *     be active against this page, and
                 *
                 *     (2) xll_page_get_try() doesn't call xll_balance_pages()
                 *     that may block waiting for ownership and io.
```

```
                 *
                 */
                page = xll_page_get_try(cache, i);
                if (page == NULL)
                        continue;
                /* obtain struct xll_async_page stored in page->private field
                   reserved by page cache to clients */
                xlap_from_page(page, &xlap);
                if (xlap->defer_uptodate)
                        continue;
                /* start asynchronous IO against @page.
                   XLL_PAGE_IO bit will be set. */
                issue_page_read(exp, xlap, oig, 1);
         }
        /*
         * a bunch of pages was created in the loop above and we couldn't
         * balance page cache there (see comment in the loop). Balance it now.
         */
        xll_balance_pages(xnode->cache.domain);
}
```

5.4.5 State specification.

Diagram of external state transitions



Informal description of external states:

NEW page was just created. All fields are zeroed.

NO-INIT page is already owned by thread that created it, but page initialization method (->page_init()) from page operations vector wasn't yet run.

USED page is referenced and owned by thread. Pages in this state are returned by xll_page_{get,try, look}(). This is „normal" state to work with page.

HEARD-BANSHEE owner decided to get rid of a page. Page will be removed when last reference to it is released.

IO-LAUNCHED owner started IO on a page.

IO-IN-FLIGHT IO is in progress, but owner released ownership on a page (asynchronous IO like read-ahead).

REFERENCED page is referenced by some thread and/or data-structure, but nor owned.

CACHED when last reference to page is released, page usually continues to live in the cache (unless it heard banshee already).

KILLED when last reference to page that heard banshee is removed, page is immediately destroyed. This is the only way to get rid of a page.

OWNED this is „multi-state“ for all states where page is owned by some thread.

NO-REFS this is „multi-state“ for all states where there are no references to the page.

Mapping from external states to internal state of struct xll_page.

NEW `page->flags == 0 && page->ref == 1`

NO-INIT `page->flags == XLL_PAGE_BUSY && page->ref == 1`

USED `page->flags == XLL_PAGE_BUSY | XLL_PAGE_INIT && page->ref > 0`

HEARD-BANSHEE `page->flags == XLL_PAGE_BUSY | XLL_PAGE_INIT | XLL_PAGE_HEARD_BANSHEE && page->ref > 0`

IO-LAUNCHED `page->flags == XLL_PAGE_BUSY | XLL_PAGE_INIT | XLL_PAGE_IO && page->ref > 0`

IO-IN-FLIGHT `page->flags == XLL_PAGE_INIT | XLL_PAGE_IO && page->ref > 0`

REFERENCED `page->flags == XLL_PAGE_INIT && page->ref == 1`

CACHED `page->flags == XLL_PAGE_INIT && page->ref == 0`

KILLED `page->flags == XLL_PAGE_INIT | XLL_PAGE_HEARD_BANSHEE && page->ref == 0`

OWNED `page->flags == XLL_PAGE_BUSY && page->ref > 0`

NO-REFS `page->flags == 0 && page->ref == 0`

Note: bit-flags XLL_PAGE_DIRTY, XLL_PAGE_UPTODATE, and XLL_PAGE_ERROR are omitted from the internal state definition, because these bit-flags never used or inspected by page cache itself, and are reserved for page cache clients to implement page handling logic (*i.e.*, when to start IO, *etc.*)
State transition description:

0 NEW -> NO-INIT page_own()

1 NO-INIT -> USED page_call_init()

2 USED -> HEARD-BANSHEE xll_page_del()

3 USED -> IO-LAUNCHED many places in xll_io.c where XLL_PAGE_IO bit is set

4 IO-LAUNCHED -> USED synchronous IO completed (see xll_io.c:xll_fetch_page() for example)

5 IO-LAUNCHED -> IO-IN-FLIGHT thread that started asynchronous on a page, releases ownership on it (xll_io.c:readahead())

6 IO-IN-FLIGHT -> REFERENCED asynchronous IO completes (xll_io.c:completion())

7 REFERENCED -> CACHED xll_page_release(), at this moment ->page_store() method of page operations vector is invoked.

8 CACHED -> REFERENCED xll_page_{own,own_try,get_try}(), at this moment ->page_load() method of page operations vector is invoked.

9 HEARD-BANSHEE -> KILLED page_recycle(), last reference to page that heard banshee released.

XLL page cache invariants:

– invariant of page cache object (checked by xll_page.c:xll_cache_invariant()): under cache->guard lock, number of elements in each page queue equals corresponding page count, and total number of pages in all queues equals cache->nr_pages.

– invariant of page (checked by xll_page.c:xll_page_invariant()): if page is owned, its reference count is greater than 0 (more should be added here).

5.5 Vnode/xnode handling

XNU VFS operates on vnodes which represent file system objects at the VFS level. struct vnode contains fields for reference counting, various flags, UBC (Universal Buffer Cache) data, and does not contain usual UNIX file attributes (like size, nlink, times, credentials, *etc.*). Instead it contains opaque pointer (->v_data) to which file system back-end is free to attach additional object with object attributes and file system specific data.

XLL attaches to each vnode instance of struct xnode: there is 1-to-1 mapping between vnodes and xnodes for „normal“ types of files (XLL also uses so called *pseudo* files (5.9 on page 58) that only have vnodes). Similarly UFS/FFS has inode, NFS has nfsnode, HFS+ has cnode.

Vnodes are not kept in any global index, VFS relies on file system back-end to somehow keep track of vnodes (that file system returns to VFS through various VOP_ and VFS_ calls: VFS_ROOT(), VOP_LOOKUP(), VOP_CREATE(), *etc.*). Usual way to do this, is to keep file system specific nodes in some sort of global data-structure keyed by unique key, locate file system nodes there, and return corresponding vnode to VFS.

[WARNING: Following is not how things are implemented now. Current xnode is identified by md->body->ino.]

Unique identifier for Lustre file system object is its *fid* (struct lustre_fid), and therefore xnodes are kept in global hash-table, indexed by super-block (pointer to struct xll_fs) and fid. This is quite standard hash-table usage. Two main entry points are:

```
int xll_vnode_get(struct xll_fs *sb,
                  obd_id oid, struct lustre_md *md, struct vnode **vnode);
struct vnode *xll_vnode_look(struct xll_fs *sb,
                             obd_id oid, __u32 generation,
                             int flags);
```

Xnode life-cycle has two „critical sections“: one during xnode initialization, and another during xnode recycling: xnode can be initialized and finalized only once (and, therefore, by one thread). Other threads has to wait while critical section finishes. This is achieved through standard UNIX busy-wanted-bits idiom (3.3 on page 8). To serialize xnode initialization LLI_IN_ALLOC and LLI_IN_WALLOC bits are used, to serialize xnode finalization LLI_IN_TRANSIT and LLI_IN_WTRANSIT bits (***WARNING***: these LLI_IN_ names are going to change). LLI_IN_STRANSIT is set and cleared during xll_vnode_get(), LLI_IN_TRANSIT—during xll_reclaim() (VOP_RECLAIM() implementation, see 3.1.2 on page 6).

5.6 DLM Lock handling

5.6.1 struct xll_lock_info

XLL introduces special *wrapper* type that includes Lustre DLM lock handle plus all other bits of information necessary to manipulate this lock as a unit:

```
enum xll_lock_info_type {
        XLL_LOCK_MDC    = 0x1000,
        XLL_LOCK_EXTENT = 0x1001
};
```

```
/*
 * structure, describing Lustre DLM lock.
 */
struct xll_lock_info {
        __u32  magic;
        enum xll_lock_info_type type;
        /* LDLM lock handle */
        struct lustre_handle lh;
        /* mode in which lock is used */
        __u32                mode;
        /* file system object this lock is on */
        struct xnode        *owner;
        /* linkage into list of all locks for a given session */
        struct list_head     linkage;
};
```

xll_lock_info is useful (among other things) to keep track of all locks within a session (5.2 on page 24): when session is finishing, all its locks are released. To release locks one needs some additional information like mode in which lock was acquired (for IBITS locks), or xnode/vnode this locks is is on (for extent locks). Following invariant (checked by xll_session.c:xll_lock_info_invariant()) is maintained for lock-info:

```
li->magic == XLL_LOCK_MAGIC &&
li->mode != 0 &&
(li->type == XLL_LOCK_MDC <=> xll_lock_is_mdc(li)) &&
(li->type == XLL_LOCK_EXTENT <=> xll_lock_is_osc(li)) &&
(li->type == XLL_LOCK_EXTENT => li->owner != NULL)
```

Following API is provided to manipulate xll_lock_info:

– xll_lock_info constructor/destructor

`void xll_lock_info_init(struct xll_lock_info *li);`

description. initialize @li to represent no lock.

parameters.

li. lock-info to initialize.

`void xll_lock_info_done(struct xll_lock_info *li);`

description. release lock associated with @li, if any

parameters.

li. lock-info to finalize

– MDC locks support. MDC (aka IBITS locks) come in two flavors: LOOKUP and UPDATE locks.

```
/*
 * locks on xnode/vnode
 */
enum xll_lock_type {
        XLL_LOCK_UPDATE = (1 << 0),
        XLL_LOCK_LOOKUP = (1 << 1)
};
```

Both protect some parts of file system object meta-data. LOOKUP lock by definition protects meta-data that are necessary perform lookup within given object. This includes: object existence, credentials (uid and gid), and permission bits.

XXX nikita: what about supplimentary groups? UPDATE lock protects the rest of file meta-data: nlink, size, times, *etc.* Lock acquisition functions take a bitmask of require locks (e.g., `XLL_LOCK_UPDATE | XLL_LOCK_LOOKUP`). To simply bitmask selection a map os provided from inode fields to required lock types:

```
/*
 * xnode/vnode fields protected by lock(s).
 */
enum xll_fields {
        XLL_MODE  = XLL_LOCK_LOOKUP,
        XLL_UID   = XLL_LOCK_LOOKUP,
        XLL_GID   = XLL_LOCK_LOOKUP,
        XLL_NLINK = XLL_LOCK_UPDATE,
        XLL_SIZE  = XLL_LOCK_UPDATE,
        XLL_ATIME = XLL_LOCK_UPDATE,
        XLL_CTIME = XLL_LOCK_UPDATE,
        XLL_MTIME = XLL_LOCK_UPDATE
};
```

so that user may specify bitmask as XLL_NLINK | XLL_CTIME explicitly enumerating object fileds that are affected by the current operation.

`int xnode_get_ldlm_lock(struct xnode *xn, int locks, struct xll_lock_info *li, int get_real);`

description. search for already existing IBITS lock. (Either LOOKUP or UPDATE.)

parameters

xn file to search lock for

locks bitmask of lock to search for (only one bit may be set)

li lock-info for found lock

get_real if this is set, call md_get_real_obd() to find name space in which lock is located. This is needed for „new“ MDAPI. Used by xll_get_readdir_lock() (Also should be used by flock(2) support code.).

return

0 no lock found

>0 lock found

<0 code of error that occured

`int xnode_has_ldlm_lock(struct xnode *xn, int locks);`

description. checks whether all locks from bitmap @locks are held by this client

parameters

xn file to check locks for

locks bitmap of locks that are checked

return

true iff all locks from @locks bitmap are present on the client

– Extent locks. These are handled through following functions:

`int xll_extent_lock(struct ll_file_data *fd, struct xnode *xnode, byte_off_t start, byte_off_t end, int flags, int mode, struct xll_lock_info *li);`

description. acquire extent lock of mode @mode on the $[start, end)$ byte-range within file @xnode. Take LL_FILE_IGNORE_LOCK bit-flag for the file descriptor @fd and LL_SBI_NOLCK flag for the super-block into account. Pass @flags to obd_enqueue().

parameters

fd file descriptor for @xnode. This is used to check for LL_FILE_IGNORE_LOCK flag. Maybe NULL.

xnode file to lock byte-range of.

start first byte to lock

end first byte after locked region

flags passed down to obd_enqueue(). typical values:

LDLM_FL_BLOCK_NOWAIT trylock: only return lock if it was available immediately (used to implement O_NONBLOCK IO);

LDLM_AST_DISCARD_DATA instruct lock cancellation call-back (that MDS will fire on a client that owns this lock) to discard all dirty data in the range instead of writing them back. (Used by truncate.)

mode locking mode: LOCK_PR or LOCK_PW

li lock-info to fill with information about acquired lock

return

0 succes, lock was acquired.

<0 error code

`int xll_extent_unlock(struct xnode *xnode, struct xll_lock_info *li);`

fd release extent lock

parameters

xnode file to release lock on

li lock-info describing lock to release

5.6.2 DLM resource names

In Lustre DLM locks (struct ldlm_lock) are always associated with some *resource* (struct ldlm_resource), and resource has a *name* (struct ldlm_res_id), which is just an array of opaque __u64 values. XLL uses the fact that for IBITS locks first two elements of resource name array are inode number and generation respectively of file this lock is for. Specifically, DLM provides a field ->l_ast_data in ldlm_lock reserved for DLM clients. Linux port stores pointer to inode in this field to map lock to inode during lock cancellation. XLL, on the other hand, stores pointer to the super block in ->l_ast_data and uses resource name to identify vnode/xnode. This (purportedly) allows to avoid races between lock cancellation and inode reclamation that are known to exist.

For extent locks, XLL does the same as Linux port: stores poiner to vnode/xnode in ->l_ast_data.

5.6.3 Lock revocation

- IBITS locks. To cancel IBITS lock, xll_lock.c:blocking_ast() first maps lock vnode/xnode (5.6.2). If LOOKUP lock is cancelled, name-cache entry for this object is removed (cache_purge()), because object existence is protected by LOOKUP lock. If UPDATE lock is cancelled for a directory, all cached pages are invalidated (that is, next readdir will have to re-fetch them from the server).

- Extent locks.

  * extent lock cancellation call-back is function xll_lock.c:extent_lock_callback(). It is mostly identical to the appropriate Linux function, except for additional optimization in page eviction code (xll_lock.c:remove_extent()). This function is responsible for sending all dirty pages covered by the lock being cancelled back to the server (or for just dropping them if LDLM_FL_DISCARD_DATA bit-flag is set). In Linux port of Lustre client this is implemented by scanning whole byte range of the lock, checking for existing dirty pages, and by sending them one by one to the OST. Problem with this is that sometimes client is asked to cancel locks with *huge* byte range (like [0, EOF)) and scanning whole possible range of page indices takes a lot of time in this case (in fact, server may easily time-out and evict client while waiting for lock cancellation). Situation is further complicated by stripping: if stripe lock is cancelled, range of pages to be scanned has holes. In the simple case of single stripe, XLL uses page iteration API (5.4.1 on page 42) to find all pages in the range. In the

case of a large sparsely populated range of pages, this API can be much more efficient than direct scanning, because it will scan page queues instead;

* glimpse callback handling is done exactly like in Linux.

When xnode is reclaimed (by xll_reclaim()), ->l_ast_data field of all locks associated with this xnode is reset to XLL_LOCK_NO_OBJECT value. Lock cancellation call-back checks for this value and returns ELDLM_NO_LOCK_DATA if xnode is already reclaimed. Races are avoided by synchronizing ->l_ast_data accesses through global xll_ast_data_lock semaphore. ***WARNING***: this may well be very contended!

5.7 File attributes handling

File attributes handling is very similar to Linux one. Attributes are fetched from server by call to md_getattr() and updated on server by call to md_setattr(). md_getattr() is called in 3 places:

- xll_super.c:fetch_root() function called during mount to obtain attributes of root directory;
- xll_vnops.c:xll_access() function that implements VOP_ACCESS() method. If object does not have LOOKUP lock, xll_access() fetches up-to-date attributes from the server;
- xll_vnops.c:getattr() function called only by xll_vnops.c:xll_getattr() (VOP_GETATTR()). If object does not have either LOOKUP or UPDATE lock, getattr() fetches attributes from the server. File size is obtained by glimpsing, because result of VOP_GETATTR() is inherently racy even on local file system.

md_setattr() is called only by VOP_SETATTR() implementation xll_vnops.c:xll_setattr().

As in Linux, attributes are sometimes updated *on occasion*, when struct lustre_md is piggy-backed to request sent by the server. This is done in:

- xll_lock.c:mk_op(): attributes of new file are returned by server;
- xll_lock.c:md_lock_get_it(): attributes of file found or created during intent handling are returned;
- xll_super.c:fetch_root(): attributes of root directory are returned;
- xll_vnops.c:mknod(): attributes of newly created special file (UNIX domain socket) are returned;
- xll_vnops.c:xll_setattr(): attributes are returned in reply to md_setattr().

5.8 XLL VOP methods

Following is a brief reference for all VOP methods implemented by XLL

name	implementation	intent handler	comments
VOP_DEFAULT	vn_default_error		
VOP_LOOKUP	xll_lookup		xll_lookup() calls resolve() to do actual work, and then adjusts VFS locks and return code to match XNU conventions. resolve() checks permissions (VOP_ACCESS()), handles name-cache, and calls xll_intent_lock().
VOP_CREATE	xll_create		if called to create UNIX domain socket, calls xll_lock.c:mknod(), otherwise all work is already done by open_op().
VOP_WHITEOUT			
VOP_MKNOD	xll_mknod		calls md_create()
VOP_MKCOMPLEX			

VOP_OPEN	xll_open	open_op	all work is already done by open_op(). xll_open() releases extra reference acquired by open_op() (see 5.2.3 on page 30). open_op() sends open intent to the server (md_lock_get_it()), obtains child vnode, and attaches XLL data to the file descriptor.
VOP_CLOSE	xll_close		calls md_close(). Also it should send dirty pages to the server, but this is temporary disabled (to match expectations of sanity.sh test 45).
VOP_ACCESS	xll_access		if vnode has no LOOKUP lock, fetch attributes from the server. Follow standard UNIX rwx logic to calculate access rights.
VOP_GETATTR	xll_getattr		if vnode does not have both UPDATE and LOOKUP locks, fetch attributes from the server. Glimpse file size.
VOP_SETATTR	xll_setattr		calls md_setattr() and obd_setattr()
VOP_GETATTRLIST			
VOP_SETATTRLIST			
VOP_READ	xll_read		acquires extent lock, and does page-by-page read
VOP_WRITE	xll_write		acquires extent lock, and does page-by-page write
VOP_LEASE			
VOP_IOCTL			
VOP_SELECT			
VOP_EXCHANGE			
VOP_REVOKE			
VOP_MMAP			
VOP_FSYNC	xll_fsync		Uses page-iteration API (5.4.1) to send all dirty pages back to the server, then calls md_sync() and obd_sync().
VOP_SEEK			
VOP_REMOVE	xll_remove	unlink_op	unlink_op() does all the work and by calling md_unlink() and ll_objects_destroy_generic(), then returns fake vnode to the VFS (5.1.2). xll_remove() is a no-op.
VOP_LINK	xll_link	link_op	link_op() always returns success and pretends child does not exist. xll_link() calls md_link() to do real work.
VOP_RENAME	xll_rename	rename_src_op	rename_src_op() only stores last path-name component in the current session. rename_dst_op() does real work by calling md_rename(). xll_rename() is a no-op. See 5.1.2
		rename_dst_op	
VOP_MKDIR	xll_mkdir	mkdir_op	mkdir_op() calls md_create(), creates vnode of new object and stores it into current session. xll_mkdir() takes it from there and returns to VFS.
VOP_RMDIR	xll_rmdir	unlink_op	much like VOP_REMOVE()
VOP_SYMLINK	xll_symlink	symlink_op	much like VOP_MKDIR()
VOP_READDIR	xll_readdir		fetches ext3 directory pages from the MDS, stuff them into VFS supplied uio.
VOP_READDIRATTR			
VOP_READLINK	xll_readlink		uses md_getattr() to obtain symlink body
VOP_ABORTOP	xll_abortop		cleans up aborted VFS operation (*e.g.*, when path-name resolution leaves Lustre name-space due to symlink)
VOP_INACTIVE	xll_inactive		called when vnode is put on the free list (3.1.2 on page 6). If file has no names (nlink == 0), vnode is destroyed immediately.

VOP_RECLAIM	xll_reclaim		called when XLL vnode is just about to be re-used. Tears down all xnode state: – removes name-cache binding – removes xnode from the hash-table – removes xnode from the per-super-block list of xnodes – sends all dirty pages to the server – invalidates all pages – updates all locks, so that their cancellation call-backs will detect that xbode is already gone – wakes up xnode waiters (5.5 on page 52) – frees xnode
VOP_LOCK	xll_lock		acquires VFS lock on xnode. If necessary creates current session (5.2).
VOP_UNLOCK	xll_unlock		releases VFS lock on xnode. Destroys current session if necessary.
VOP_BMAP			
VOP_STRATEGY			
VOP_PRINT	xll_print		print xnode fields to the console (debugging method)
VOP_ISLOCKED	xll_islocked		returns true iff VFS lock on the vnode is held by some thread.
VOP_PATHCONF			
VOP_ADVLOCK			
VOP_BLKATOFF			
VOP_VALLOC			
VOP_REALLOCBLKS			
VOP_VFREE			
VOP_TRUNCATE			
VOP_ALLOCATE			
VOP_UPDATE			
VOP_PGRD			
VOP_PGWR			
VOP_BWRITE			
VOP_PAGEIN	xll_pagein		acquires extent lock covering all pages to be paged in. Reads them through issue_page_read().
VOP_PAGEOUT	xll_pageout		for each page in range, tries to acquire ownership of a page, if that succeeds, sends page out.
VOP_DEVBLOCKSIZE			
VOP_SEARCHFS			
VOP_COPYFILE			
VOP_BLKTOOFF			
VOP_OFFTOBLK	xll_offtoblk		divides offset by blocksize

5.9 Debugging infrastructure: pseudo files and counters

This section is now well-structured. It's not actually more of a data-dump.

5.9.1 Assertions

XLL has its own assertion checking macro

```
XLASSERT(condition);
```

This is so, because standard LASSERT() macro in Lustre is allowed to have side-effects (like LASSERT(a ++ != 0)). As a result, it is impossible to safely turn assertion checking off in core Lustre code. XLL, on the other hand, uses XLASSERT for computationally expensive checks that should be disableable.

In addition compilation time assertion macro

```
XCASSERT(condition);
```

is provided. Condition in this macro should be constant expression (in C language sense), and compilation will be aborted with this expression evaluates to 0.

5.9.2 Stack back-tracing

XLL provides simple API for stack back-trace recording:

```
enum {
        XLL_STACK_TRACE_DEPTH = 16
};
struct xll_stack_trace {
        void *frame[XLL_STACK_TRACE_DEPTH];
};
void xll_stack_trace_fill(struct xll_stack_trace *trace);
```

5.9.3 Pseudo files.

xll_pseudo.h privides an interface (and xll_pseudo.c an implementation) to *pseudo* files. Pseudo files are objects visible in XLL file system name-space, that respond to normal system calls like read(2) and write(2), but instead of returning or storing data somewhere, they return some internal kernel information, or change kernel parameters. Pseudo files are much like procfs or sysfs (and many other pseudo file systems in Linux), they are needed, because XNU lacks built-in functionality to easily export some piece of kernel information or some tunable knob to the user level.

Pseudo files

- are attached to the root directory of mounted XLL instance, and
- by convention have names of the form __something__, and
- are invisble for readdir.

API:

```
struct xll_pseudo_desc {
        /*
         * name of pseudo file. Should have the form "__something__"
         */
        char                        *name;
        /*
         * vector of VOP method descriptions for this pseudo files. Usually it
         * contains non-trivial VOP_READ(), and VOP_WRITE() (if pseudo file
         * supports writes).
         *
         * VOP_GETATTR() should be set to xll_pseudo_getattr().
         *
         * VOP_OPEN(), VOP_CLOSE(), VOP_ACCESS(), VOP_SETATTR(),
         * VOP_ABORTOP(), VOP_INACTIVE(), VOP_RECLAIM(), VOP_LOCK(),
         * VOP_UNLOCK(), VOP_ISLOCKED() should return success, all other
         * methods should return failure (or can be left unset).
```

```
         */
        struct vnodeopv_entry_desc *opdesc;
        /* type of file. Usually VREG */
        enum vtype                  type;
        /*
         * the rest of the fields is used by implementation and shouldn't be
         * touched by user.
         */
        int                        (**vector)(void *);
        int                          index;
        struct list_head             linkage;
};
/*
 * registers pseudo file. Has to be called before first mount.
 */
int xll_pseudo_register(struct xll_pseudo_desc *desc);
```

Following pseudo files are implemented currently:

/__oscio__ read-only pseudo file to export OSC IO statistics. Sample output:

```
snapshot_time:         1108061255:494239 (secs:usecs)
RPCs in flight:        4
pending write pages:   1010
pending read pages:    0
                       read                    write
pages per rpc         rpcs  % cum % |       rpcs   % cum %
1:                      0   0   0   |        213  63  63
2:                      0   0   0   |         66  19  82
4:                      0   0   0   |         36  10  93
8:                      0   0   0   |         12   3  96
16:                     0   0   0   |          2   0  97
32:                     0   0   0   |          6   1  99
64:                     0   0   0   |          2   0  99
128:                    0   0   0   |          1   0 100
                       read                    write
rpcs in flight        rpcs   % cum % |       rpcs   % cum %
0:                       0   0   0   |          5   1   1
1:                       0   0   0   |         13   3   5
2:                       0   0   0   |         10   2   8
3:                       0   0   0   |        307  90  99
4:                       0   0   0   |          3   0 100
```

/__stats__ read-only pseudo file to export statistical counters (5.9.4 on the next page). Sample output:

```
refresh_count_call:             868
refresh_count:          2759376
submit-sync:            1
submit-skip:            0
submit:         1878
page-out-sync:          0
page-out:               365
lock_glimpse:           0
extent_time:            194209962000
extent_page_del:                333
extent_page_out:                333
lock_osc_cancel:                332
lock_mdc_cancel:                1187
cache-miss:             3760
cache-hit:              0
dirty:          1020
pages:          1877
reclaimed:              0
washen:         0
invalidated:            335
obd allocator: 0 0
portals memory: 116424
xnodes: 931
```

/__debug__ read-write pseudo file that shows and allows to manipulate portal_subsystem_debug and portal_debug Lustre variables that control what goes into debugging log. Sample output:

```
subsystem:      0xffffffff
mask:   0xffffffff
```

Tuning:

```
echo 'ffffffa 0' > __debug__
```

/__alloc__ read-only pseudo file that shows current histogram of OBD_{ALLOC,FREE} allocations. Sample output:

```
2       38
3       126
4       301
6       80
...
183     19
...
```

This means that currently 38 blocks of size 2, 126 blocks of size 3, *etc.* are allocated through OBD_ALLOC (and its variants). This file is useful to find and track memory leaks.

/__zones__ read-only pseudo file that shows information about *zones*: XNU analogues of Solaris/Linux slab caches. Sample output:

```
NAME            ADDR     COUNT FREE     CUR MAX ELEM ALLOC
zones           0xa7b000 94    10994968 12168 20480 72 20480 [ e ]
vm objects      0xa7df88 15561 29877056 2494240 2654208 160 20480 [ ce ]
vm object hash e 0xa7df40 12465 46707192 253920 524288 20 20480 [ ce ]
maps            0xa7def8 64    28120636 5796 40960 84 16384 [ e ]
non-kernel map e 0xa7deb0 3285  29630668 139176 1048576 36 16384 [ ce ]
kernel map entri 0xa7de68 2081  10891968 569340 569344 36 16384 [ ]
map copies      0xa7de20 0     47255512 20480 20480 40 20480 [ ce ]
pmap            0xa7ddd8 55    29241344 28672 204800 512 4096 [ ce ]
kalloc.16       0xa7dd90 11428 25268832 262144 279936 16 4096 [ ce ]
kalloc.32       0xa7dd48 16993 25786176 606208 663552 32 4096 [ ce ]
kalloc.64       0xa7dd00 25264 31625024 2633728 2985984 64 4096 [ ce ]
...
xll_locks       0xa7c638 0     30814176  4096 4096 32 4096 [ ce ]
xll_requests    0xa7c5f0 0     24555508 12288 12288 12 12288 [ ce ]
xll_sessions    0xa7c5a8 0     30752704 4096 8192 64 4096 [ ce ]
xll_dir_pages   0xa7c560 0     30822400 8192 4194304 4096 4096 [ ce ]
```

/__vnode__ read-only pseudo file that lists vnodes currently cached on the client. Sample output:

```
0x2cf8f70 1 3722840 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/bsd/kern/kern_proc.c
0x2cf9008 1 3722839 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/bsd/kern/kern_physio.c
0x2cf90a0 1 3722838 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/bsd/kern/kern_pcsamples.c
0x2cf9138 1 3722837 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/bsd/kern/kern_panicinfo.c
0x2cf91d0 1 3722836 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/bsd/kern/kern_newsysctl.c
0x2cf9268 1 3722835 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/bsd/kern/kern_mman.c
0x2cf9300 1 3722834 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/bsd/kern/kern_mib.c
0x2cf9398 1 3722833 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/bsd/kern/kern_malloc.c
...
0x1d6dc90 1 196802 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/pexpert/conf/files
0x1d6dd28 1 213202 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/pexpert/conf/tools/Makefile
0x1d6ddc0 1 246003 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/pexpert/conf/tools/newvers/newvers.csh
0x1d6de58 1 246002 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/pexpert/conf/tools/newvers/Makefile
0x1d6e020 2 246001 [VSTANDARD|] 2489 0 VDIR (2) ... /mnt/lustre/pexpert/conf/tools/newvers
0x1d6e0b8 1 229603 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/pexpert/conf/tools/doconf/Makefile
0x1d6e150 1 229602 [VSTANDARD|VUINACTIVE|] 0 0 VREG (1) ... /mnt/lustre/pexpert/conf/tools/doconf/doconf.csh
0x1d6e318 2 229601 [VSTANDARD|] 2484 0 VDIR (2) ... /mnt/lustre/pexpert/conf/tools/doconf
0x1d6e578 3 213201 [VSTANDARD|] 2491 0 VDIR (2) ... /mnt/lustre/pexpert/conf/tools
0x1d6ee60 15 196801 [VSTANDARD|] 2519 0 VDIR (2) ... /mnt/lustre/pexpert/conf
0x1d0f098 6 180401 [VSTANDARD|] 2626 0 VDIR (2) ... /mnt/lustre/pexpert
0x1d0fe40 1 0 [VSYSTEM|VSTANDARD|] 0 0 VREG (1) ... /mnt/lustre/__stats__
0x1d0fed8 1 0 [VSYSTEM|VSTANDARD|] 0 0 VREG (1) ... /mnt/lustre/__debug__
0x1d0ff70 1 0 [VSYSTEM|VSTANDARD|] 0 0 VREG (1) ... /mnt/lustre/__alloc__
0x1d10008 1 0 [VSYSTEM|VSTANDARD|] 0 0 VREG (1) ... /mnt/lustre/__zones__
0x1d100a0 2 0 [VSYSTEM|VSTANDARD|] 0 0 VREG (1) ... /mnt/lustre/__vnode__
0x1d10138 14 65601 [VROOT|VSTANDARD|] 0 0 VDIR (2) ...
```

5.9.4 Statistical counters.

XLL provides an API allowing its user to define a 64-bitper-mount counter that can be manipulated (incremented, decremented, *etc.*) and that is automatically exported through pseudo file /__stats__ (5.9.3 on the preceding page). Such counters are convenient to collect statistics. API:

```
typedef u_int64_t  xll_stat_cnt_t;
/*
```

```
 * description of statistical counter. Single instance exists for each
 * counter.
 */
struct xll_stat_cnt_desc {
        /* counter name, used to identify counter in /__stats__ */
        char *name;
        /*
         * fields below are initialized and used by counters API
         * implementation.
         */
        /* ordinal number of counter in stats array */
        int   index;
        /* list of all counters */
        struct list_head linkage;
};
/*
 * register counter. This is only allowed until first mount.
 */
int xll_stat_cnt_desc_register(struct xll_stat_cnt_desc *desc);
```

And (obvious) API to use counter:

```
void xll_stat_inc(struct xll_fs *sb, struct xll_stat_cnt_desc *desc);
void xll_stat_dec(struct xll_fs *sb, struct xll_stat_cnt_desc *desc);
void xll_stat_add(struct xll_fs *sb, struct xll_stat_cnt_desc *desc, int val);
void xll_stat_max(struct xll_fs *sb, struct xll_stat_cnt_desc *desc, int val);
void xll_stat_min(struct xll_fs *sb, struct xll_stat_cnt_desc *desc, int val);
```

There is a lot of counters already defined, mostly covering various xll page cache activity. Search code for xll_stat_cnt_desc_register().