



> Version 1.5 Q1 2008

LECTURE 5.1

Sample Lustre Implementation

Contents

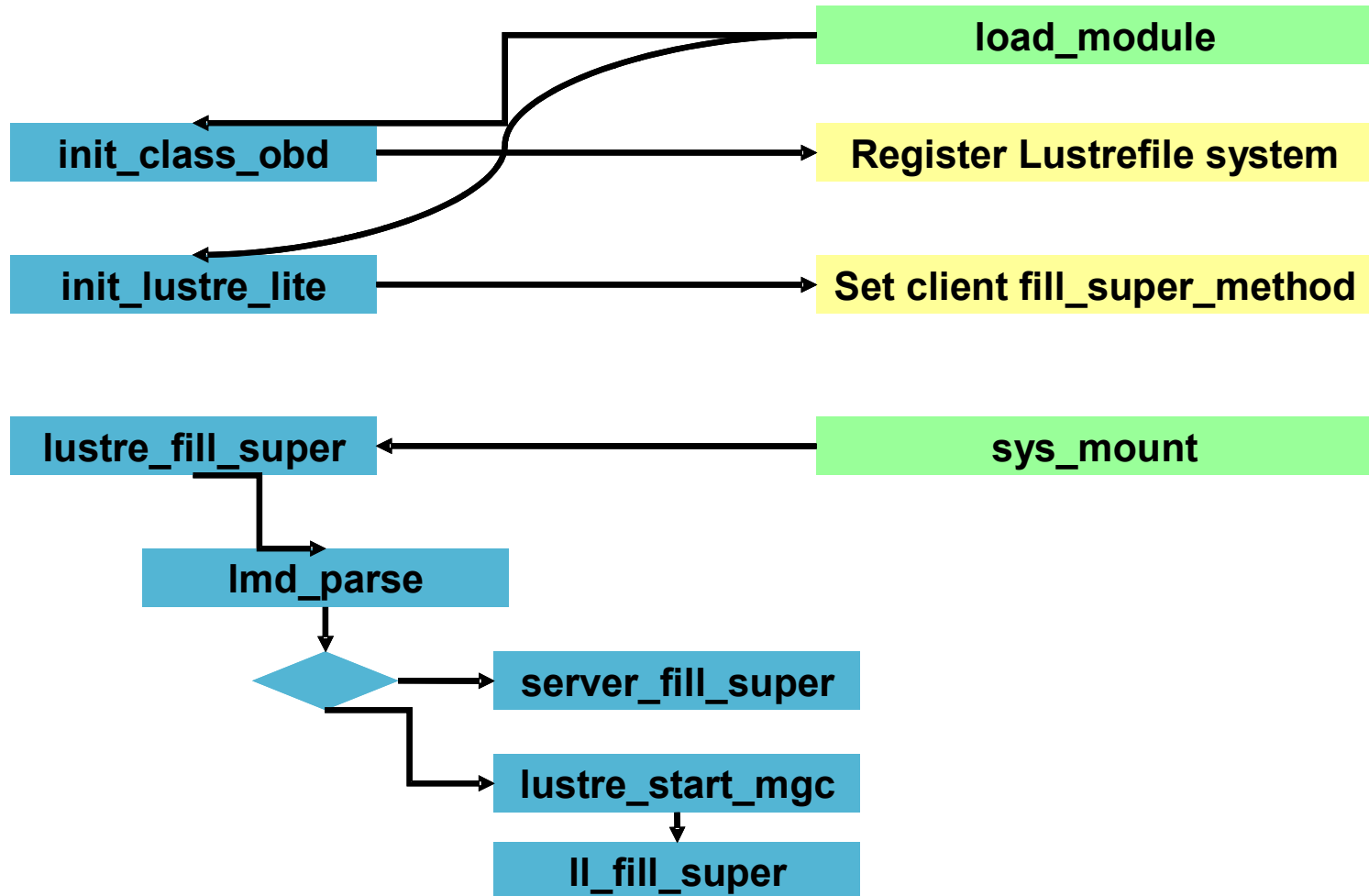
- Mount in general
- Mount on clients
- Mount on servers
- Open O_CREAT
- File Size
- File I/O

Mount in general

obd_mount.c

- Remember, in 1.6 server and clients both mount
- The file system that mounts is declared in obd_mount.c
 - > During the init call of the obdclass module
 - > Inside class_obd.c
- Then sys_mount calls lustre_fill_super
 - > lmd_parse parses the mount options from mount.lustre
 - If ‘:’ is in the mount path we are dealing with a client!
- Client
 - > Set up an MGC client in lustre_start_mgc
 - Analyze names & nids for the MGS and connect
 - Deal with failover MGS’s here!
 - > The client_fill_super, which goes to ll_fill_super in llite_lib.c
- Server
 - > Call server_fill_super for servers
 - > Details follow later

Diagram – initial part of mount

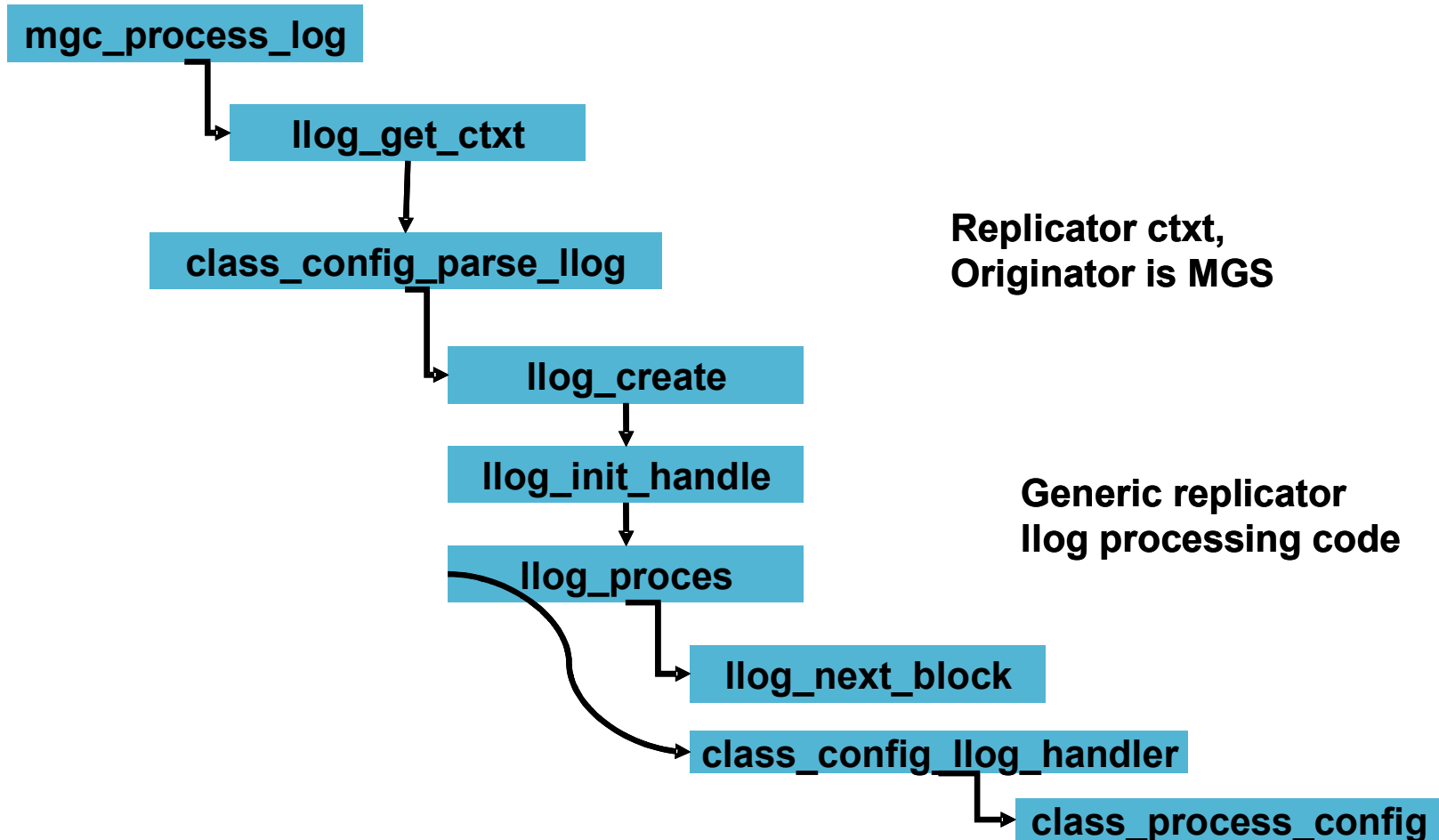


Client mount

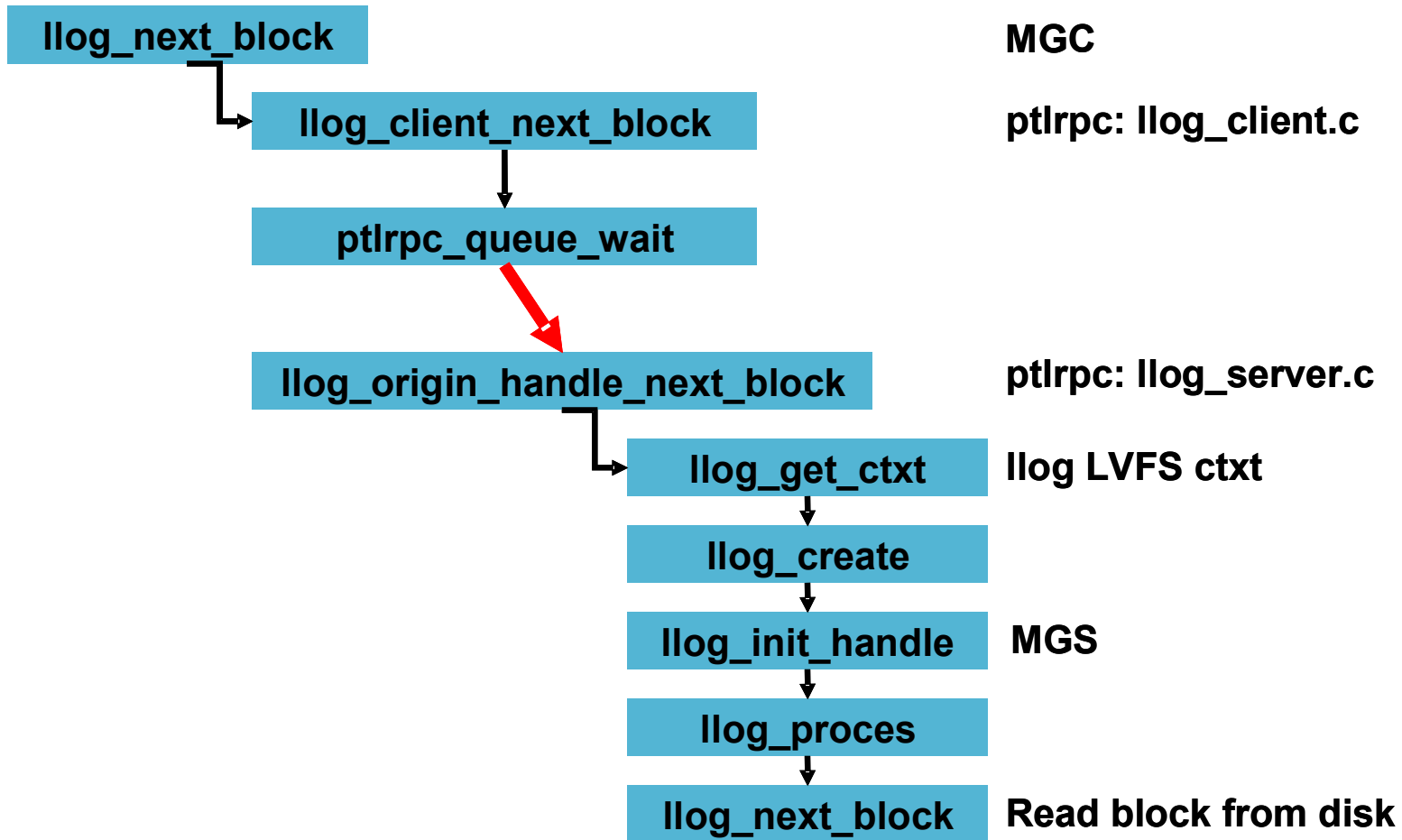
Mount on clients

- `ll_fill_super` calls into `lustre_process_log`
 - > Set up buffers to hold struct `lustre_cfg`
 - > Then call `obd_process_log`
 - Enqueue the configuration lock
 - Copy log to a local file on a server, unless we are the MGS
 - Call `class_config_parse_lllog`
 - Wrapper for `lllog_process` with iterator callback `class_config_lllog_handler`
 - This calls a critical function: `class_process_config`
- `class_process_config` performs
 - > OBD device creation, attach, setup, adding connections
- Unwind the stack, and end up in
 - > `client_common_fill_super`

Diagram – processing the llog



llog_next_block



Client common fill super

- `obd_connect` to the lov and all the osc's then ost's
- `obd_connect` to the mdc
- `mdc_get_status` – fetch root fid
- `mdc_getattr` – get root inode attributes
- Instantiate the root inode and the root directory
- You have a client file system!

Mount on servers

Inode bits lock DEBUG message

- Server mounts – two super blocks
 - > The server data is in an Idiskfs file system
 - This disk file system is not visible from user space
 - A lot of setup: recovery, services etc.
 - > The mounted file system is a fake simple file system
 - Very simple setup, does almost nothing, just statfs/mount/unmount
- Keeping complex mount information
 - > The servers disk file system is mounted twice
 - > First as Idiskfs / ext3 to read the mount options (extents, etc)
 - > The again, with the right options
 - Just to keep server mounts “trivial”

Open O_CREAT

Open(“foo”, O_EXCL|O_CREAT|O_RDWR)

- VFS in Linux would like to do:
 1. Lookup(foo)
 2. Lock the directory
 3. Perhaps creat, fail if already existed
 4. Ask the file system execute open method to get handle
 5. Unlock directory
- Lustre
 - > Does this ALL on the MDS server first as part of lookup()
 - At lookup time, go with full information for open to MDS
 - MDS follows above algorithm
 - Maintains a cache of all opens in the cluster
 - ETXTBSY – must communicated between nodes
- Intent lock call: intent == open,
 - > IN: pFID, “foo”, flags, name,
 - > OUT: file handle, and attributes and error info, no lock

Open ... ctd

- Client gets:
 - > No lock
- No error:
 - > Here is the new inode + the open file handle
 - > Client locally pretends to create the file
 - > Client locally “opens” the file
- If file existed, we pretend so locally
- Part of return
 - > Error code
 - > Disposition – where did the error happen
 - In server lookup, in server create, or in server open
 - > Correct replay on the client depends in the disposition

Open ... whoops, what about objects?

- File needs data objects to store file data
 - > Yes... mds inodes with file data are on the way.
 - > Not normally done
- During mds_open objects must be acquired.
 - > Danger of many RPCs
- mds_open will pre-create objects on ALL OSTs
 - > Some variable amount in one shot, 32 – 20,000
 - > osc_create calls: return a range of object id's (namely last id)
 - > osc pre-creation is async call, using rpcd.
- On MDS:
 - > mds_open calls lov_create which calls all osc_creates (each OST)
 - > OSC responds (normally) from pre-created pool

Open & objects

- Object id's are available:
 - > mds_md is formed, extended attribute for the MDS inode of file
 - > mds_md holds all objects that belong with that file
 - > Usually, this is a small EA inside inode, otherwise in block
- This is returned to the client in enqueue
 - > Client turns this into the "lsm" – LOV, Stripe, Metadata
 - Is used for getattr, read/write etc.
- Just a reminder:
 - > If you crash, you have clean up unused pre-created objects!!!

File Size

stat

- Chdir("/mnt/lustre") ; Stat("foo", &statbuf)
 - > Pathname "foo" must be resolved & with intent getattr
 - > Kernel queries dcache
 - Finds dentry
 - Doesn't find dentry
- Not cached:
 - > Lustre_lookup:
 - mdc_intent_lock() -> decides if there is any work to do
 - mdc_enqueue(resource="pFID = fid of /mnt/lustre", "foo", intent=getattr)
 - Idlm_cli_enqueue() -> send the RPC
 - MDS mds_handler, Idlm_handle_enqueue
 - Intent handler: mds_intent_policy; mds_getattr_lock
 - mds_getattr_lock: read data lock on pFID -> lookup for "foo"
 - Local VFS getattr call in MDS file system
 - Acquire a lock's child fid, returns attributes + lock to client
 - Client de-referenced, unwind

stat – ctd'

- Client receives attributes from MDS, includes stripe EA
- Attributes: lsm in client inode describes the objects/OSTs where the data of inode resides
- Get the file size!
 - > File size may come from any OST among the stripes
 - You don't know which one!
 - > Clients (more than just one!) may change the file size

KMS -- known minimum size

- KMS – each client has a “known minimum size”
 - > Managed per-object
 - > Client’s cached KMS must be \leq end of client’s largest lock
 - Having a lock guarantees that it can’t be truncated smaller
 - > Examples:
 - On-disk object size is 1M, client has a lock from [2M, 3M]
 - Client kms is 1M -- it could grow behind our back, but not shrink
 - On-disk object size is 1M, client has a lock from [0, 0.5M]
 - Client kms is 0.5M
 - Client has no lock -- KMS is 0
 - > ll_glimpse_size will ask OSTs for size data (also called from stat)
 - If the clients has a cached [0,-1] lock, no RPC needed
 - If glimpse_size manages to get locks, it will remember the new KMS
 - Otherwise, it must discard after use
 - > Lock cancellation may reduce the local KMS
 - > By combining the stripes’ KMSs, you get a known minimum i_size

Stack traces

- It avoids getting the size very often
- For reading, the end of a file a size is needed
 - > Read beyond the end of file returns a certain number of bytes
 - > KMS is often enough to avoid a size lock call

File size

- Intent lock request to the OST – give me the object size
 - > If object not busy – give client the read lock
 - > If object busy – just send me the size, not a lock
 - > File locks are by extent per object
- If the object busy:
 - > Server makes a list of PW lock holders – finds the extent closest to EOF
 - > Send that client a callback/AST: `glimpse_ast`
 - > Client:
 - (i) Sends a reply to the glimpse AST containing file size
 - (ii) If lock hasn't been recently used client begins to cancel asynchronously

Glimpse for file size

- Usage of Glimpse
 - > Used by Lustre filesystem clients to get KMS
 - Before issuing read/write requests on file data
 - During getattr
 - > It's a special Lustre extent lock with 'INTENT' flag bit
 - > OST has different execution
- Glimpse Protocol - delicate
 - > Server sends glimpse callback (AST)
 - But only to client which has furthest extent, or size / PW lock
 - > When receiving glimpse AST from server
 - Client returns the file size it keeps to server
 - Prepares to cancel locks if locks are unused

Glimpse AST processing

- Send size back with the reply
 - > Lock was not used in a few seconds
 - Lock is canceled with a separate cancellation thread and RPC
 - > Lock was busy recently: we keep it

File size - redux

- Glimpse AST can fail if it crosses with client cancellation
 - > Any AST for a non-existent lock replies with -EINVAL
 - > We can handle this without additional RPCs
 - > Before we send the glimpse AST, the resource knows a file size
 - This is stored in the resource lock value block (LVB)
 - This may be smaller than the actual size, but not larger
 - If there are no writers above this, we return without a glimpse
 - > We send a glimpse, this fails because it crosses with a cancel
 - We know that as part of the cancel, the LVB may get updated
 - > After a failed glimpse, we refresh the LVB from the disk inode
- This LVB is returned to the original client as object size

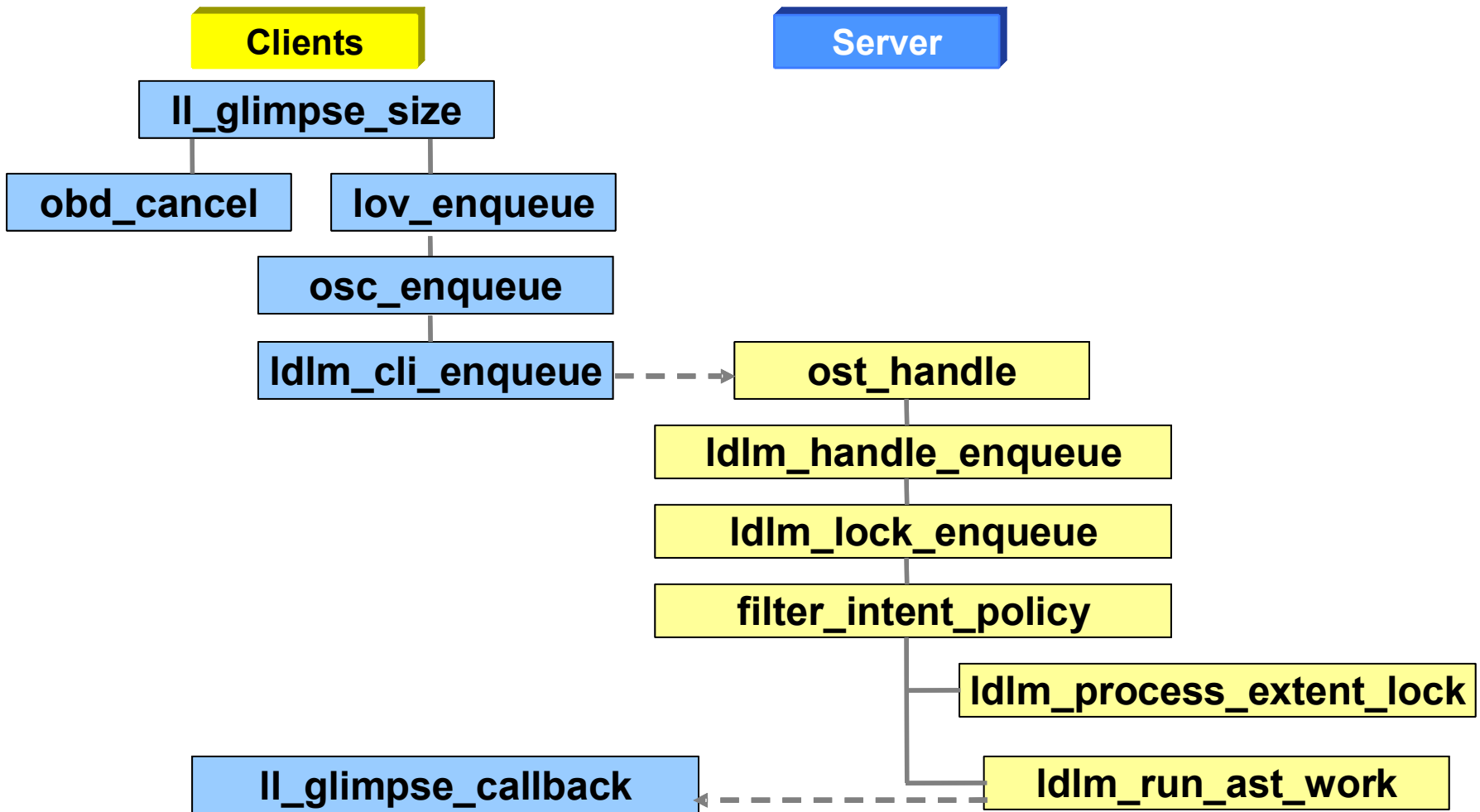
Critical Functions

- `ll_glimpse_size(inode)`
 - > Description
 - Clients use it to get the file size from OST. OST may forward the file size which is fetched from another client who currently own the lock of 'file size'
 - It's a lock request with 'INTENT' flag bit
 - > Code walk through
 - Client side
 - `llite/file.c::ll_glimpse_size()`
 - `llite/file.c::ll_glimpse_callback()`
 - `lov/lov_obd.c::lov_cancel()`

Critical Functions(cont')

- ll_glimpse_size()
 - > Code walk through
 - Server side
 - ost/ost_handler.c::ost_handle()
 - ldlm/ldlm_lockd.c::ldlm_handle_enqueue()
 - ldlm/ldlm_lock.c::ldlm_lock_enqueue()
 - obdfilter/filter.c::filter_intent_policy()
 - ldlm/ldlm_extent.c::ldlm_process_extent_lock()
 - ldlm/ldlm_lockd.c::ldlm_server_glimpse_ast()

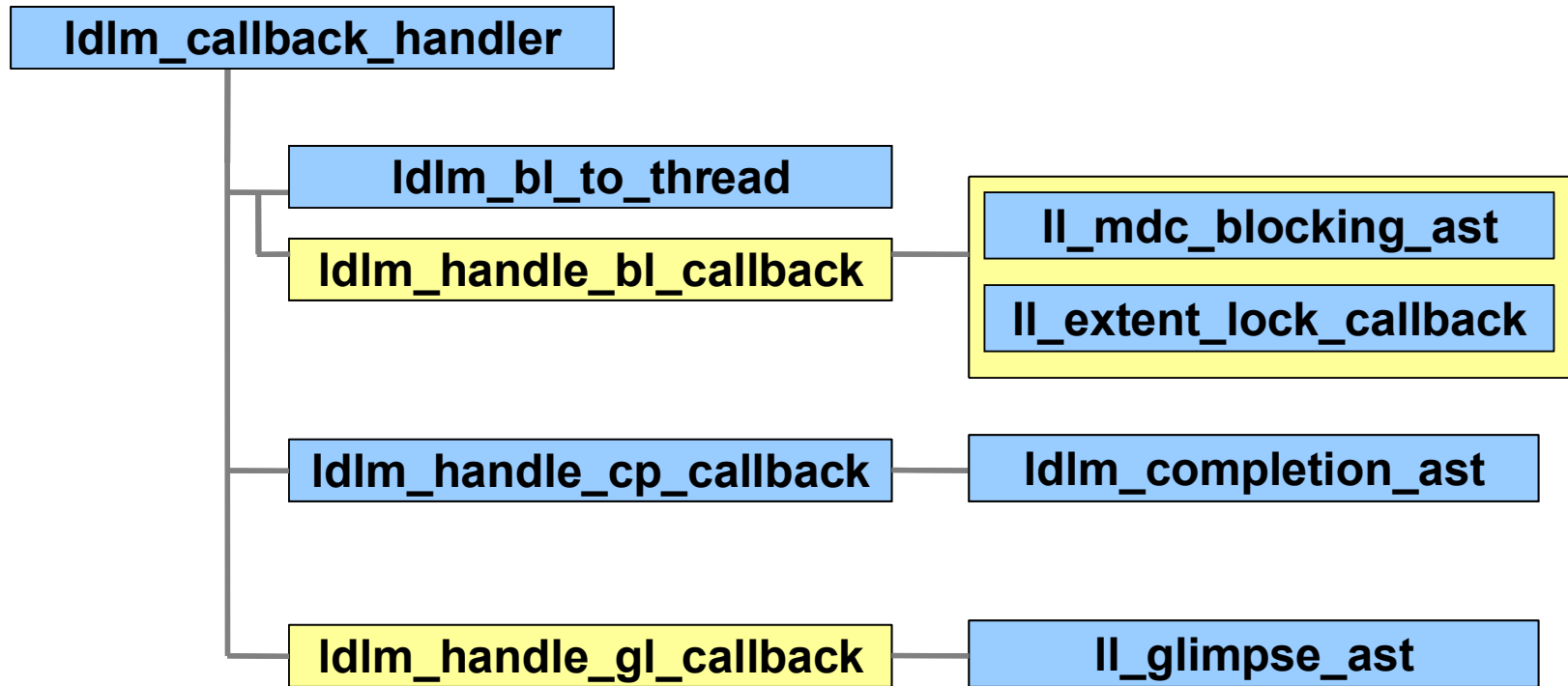
ll_glimpse_size() diagram



Critical Functions(cont')

- `ldlm_callback_handler(req)`
 - > Description
 - Client side ldlm lock callback server thread
 - Handling AST messages from servers
 - > Code walk through
 - `ldlm/ldlm_lockd.c::ldlm_callback_handler()`
 - `ldlm/ldlm_lockd.c::ldlm_bl_to_thread()`
 - `ldlm/ldlm_lockd.c::ldlm_handle_bl_callback()`
 - `llite/file.c::ll_extent_lock_callback()`
 - `llite/namei.c::ll_mdc_blocking_ast()`
 - `ldlm/ldm_lockd.c::ldlm_handle_cp_callback()`
 - `ldlm/ldlm_request.c::ldlm_completion_ast()`
 - `ldlm/ldlm_lockd.c::ldlm_handle_gl_callback()`
 - `llite/file.c::ll_glimpse_ast()`

Idlm_callback_handler() diagram



File I/O

Write – case 1

- Client1 has done open
 - > Write(fd, “foobar”, 1.5MB)
- Example LSM:
 - > Stripe size is 1M, stripe count is 3 objects
 - > E.g., if file is 3.5M in size: obj1= 1.5M, obj2=1M, obj3=1M
- Enter write system call:
 - > sys_write --> ll_file_write --> ll_tree_lock --> ...
 - > ll_extent_lock --> lov_enqueue -->
 - Get locks on all needed stripes for this write()
 - As before, lov_enqueue will call osc_enqueue serially, in stripe order
 - osc_enqueue --> round the request to page boundaries
 - > What happens on the OST lock server?

OST lock handling

- Let us assume that only one client is using the object
 - > Extent policy:
 - Requesting a lock on [0, 1M]
 - Policy function checks for conflicting locks, grows our request
 - Return lock: [0, -1]
 - Paradigm:
 - > Node gets a DLM lock, NOT the process
 - > Users on the node
 - Use the lock by increasing the refcount
 - Drop the lock
 - > Normal Linux VFS semaphores
 - Control concurrency on one node

Write ll_file_write ctd

- We now have the locks
- Now we call `generic_file_write` inside `ll_file_write`
 - > Breaks down into two `address_space` operations PER PAGE
 - Asks VM to find (in the cache) or allocate a page
 - `prepare_write` --> calls into Lustre
 - Copies the data into the page
 - `commit_write` --> calls into Lustre
 - Rinse/repeat until the whole write is complete
 - Returns to Lustre

prepare_write and commit_write

- prepare_write's job is to read the page, if necessary
 - > If PageUptodate(), do nothing, return.
 - > Are we overwriting the entire page? If yes, return.
 - > Do we know enough about the file size? If not, update.
 - > Are we writing past EOF?
 - If yes, possibly zero a partial page
 - mmap gives access to whole pages, so tail of page is 0-d also
 - Any subsequent reads must return zeros for unwritten part
 - > Read this page from OST and mark it Uptodate
- commit_write marks the ENTIRE page dirty in the cache
 - > Nothing happens immediately, normally
 - > The OSC manages cached pages (except mmap-written pages)
 - > KMS and file size is updated by commit write
 - This is the not necessarily real file size (stripe 3 is unlocked!)

Read ... more complicated!

- `ll_file_read(fd, buf, len)`, at offset: 1M,
- Read up to offset $3M = 1M + len$: 3 cases
 - > Read inside file, i.e. $EOF > 3M$, return 2M
 - > Short read: $1M < EOF < 3M$: return $size - 1M$
 - > $EOF < 1M$: return 0 (indicating we've reached EOF)
- `extent_lock(LCK_PR, [1M, 1M+len = 3M])`
- `lov_enqueue` split it up per OSC:
 - > Select OSC's in the stripe: stripe 2, `osc2` & stripe 3, `osc3`.
 - > Client 1 has lock on the whole object for `osc2`
 - Policy denies growing: lock granted as requested `[0, 1M]`
 - > Stripe 3 is empty, and nobody else has a lock
 - We get `[0, -1]`. Size of stripe 3 is 0 and is returned to the client.

Read ...

- Only way to determine what to return is file size
 - > Unfortunately many clients can change the file size
- Before getting the file size, all we know is:
 - > In this case KMS:
 - Stripe 1: no lock, no KMS info
 - Stripe 2: have lock [0, 1M] -- KMS = 1M
 - Stripe 3: have lock [0, -1], -- KMS = 0
 - Combined KMS for the entire file = 2M (based on stripe 2)
 - > We don't know anything about stripe 1
 - Other clients could be writing in stripe 1 at large offsets
 - > We're reading up to file offset 3M -- so if KMS > 3M, we're done
 - > But KMS == 2M so we MUST get the file size to know what to return (`ll_glimpse_size`)

Lock LRU

- Each client namespace has an LRU, and a max-size
- Locks with nonzero refcounts get removed from the LRU
 - > Added back to the end when they fall to 0
- If the `lru_list` grows larger than the LRU max-size
 - > Start to cancel locks from the top of the list
- Default is 100 locks -- to limit 1000 node cluster to 100K
 - > 100k locks is probably ~100MB of server RAM
- Makes sense to substantially raise limit on interactive nodes

Enqueue RPCs

- Client A sends LDLM_ENQUEUE (PW)
- Server sees that there are no conflicts
 - > Lock 1 added to the lr_granted list
 - > Server sends a reply that indicates the lock was granted
- Client B send LDLM_ENQUEUE (PR)
- Server checks for conflicts, finds lock 1
 - > Lock 2 added to the lr_waiting list
 - > Server sends a reply to client B indicating the lock NOT granted
 - > Server sends a blocking AST to client A for lock 1 (lock revocation)
- Client A sends a CANCEL for lock 1
- Server finishes:
 - > Lock 1 removed from lr_granted
 - > Lock 2 moved from lr_waiting to lr_granted
 - > Server sends a completion AST for lock 2

Blocking AST processing

- LDLM service threads on clients handle the callbacks
- Blocking AST – lock revocation message
 - > Lock is not used, (ie, l_readers, l_writers are 0)
 - Lock is immediately marked as being destroyed
 - Clients cancels in the reply to the callback -- avoid extra CANCEL rpc
 - > Lock is busy (readers or writers is nonzero)
 - Reply to AST– I'm working on it, but lock is in use
 - Lock is marked for cancellation – CB_PENDING (callback pending)
 - Lock cannot be matched by other threads
 - When refcount falls to 0, send CANCEL rpc
 - If refcount doesn't fall to 0: server evicts client.

What else can go wrong during enqueue?

- The initial enqueue RPC can timeout
 - > Client A never receives a reply
 - > Client A will timeout and reconnect
 - > Likely caused by a server crash or network outage
 - > Or highly overloaded server (more rare)
- One or more blocking ASTs can timeout (client A evicted)
 - > Server never receives a reply to the AST from client A
 - > Could be caused by a client crash / reboot (most likely)
 - > Less likely now that we have proactive eviction
- Cancels may not arrive on time, after blocking AST
 - > Server never receives a CANCEL rpc from client B (client B evicted)
 - > Could again be caused by client crash
 - > Client has too much dirty data to flush quickly (most likely)
- Completion AST may not arrive on time from server
 - > Causes client A to timeout and reconnect
 - > Cascading evictions / timeouts can cause pathologically long waits

Enqueue replies and completion ASTs

- May contain a flag that is a substitute for blocking AST
- If so, client lock gets immediately marked with `CB_PENDING`
- The Lustre DLM does try to guarantee forward progress
 - > Each enqueue is guaranteed to be granted for at least one operation

Canceling a write lock...

- We have dirty data in the VM page cache for the inode
 - > All pages dirtied by write() have an oap = obd asynchronous page
 - > i.e., OSC has list of dirty OAPs
- Lock callback (blocking AST) comes in ...
 - > Callback arrives in the OSC
 - > Bubbles up to file system – OSC shouldn't muck with the VM
 - > File system calls ll_pgcache_remove_extent(inode, extent)
- Find all the pages that lie in the extent
 - > Initiate write back on each dirty page covered by the lock
 - ll_writepage (calls address space operation, writepage)
 - Calls through LOV into OSC, marks OAP urgent
 - > Try to lock the page -- when this returns, the I/O is finished
 - > Can we throw the pages away: not if other locks cover this page
 - Read locks can overlap

Client/OST bulk data transfer (write)

- Client maps its buffers to a Portals MD
 - > With the same match bits as the RPC that it's about to send
- Client OSC prepares and sends OST_BRW rpc
 - > Contains a write flag, obj #, & sorted list of [offset,length] tuples
- The OST will map and attach the buffers to a Portals MD
- OST does a bulk_get from the OSC
 - > This internally becomes a bulk_put
 - > This actually moves the data
- OST replies to the original OST_BRW rpc
 - > Contains per-page write return code

Client/OST bulk data transfer (read)

- Client maps its buffers to a Portals MD
 - > With the same match bits as the RPC that it's about to send
- Client OSC prepares and sends OST_BRW rpc
 - > Contains a read flag, obj #, an [offset,length] tuple
- The OST will map and attach the buffers to a Portals MD
- OST does a bulk_put to the OSC
 - > This actually moves the data
- OST replies to the original OST_BRW rpc
 - > Contains per-page read return code

OST I/O processing

- OST handles all of the network I/O
 - > Either the ost or ...
- obdfilter gets called via `preprw` and `commitrw` for disk I/O
 - > We don't use the standard VFS interfaces
 - > Grabs a set of preallocated pages
 - > `ext3_map_inode_pages` --> turn logical pages into physical blocks
 - Allocate new disk blocks through `ext3`
 - > `filter_direct_io`
 - Must write out and flush the buffer & page caches
 - Then do a direct write to/read from disk with pages locked
- If `preprw` succeeds, you *must* `commitrw`



THANK YOU