Lustre: some protocol basics & debugging

LUG, April 2009

Johann Lombardi
Lustre Group
Sun Microsystems
Topics

> Building Lustre
> Some protocol basics
  - Request lifecycle
  - llog
  - I/O in the OST
  - ldlm
  - quota
> Debugging
Topics

> Building Lustre
> Some protocol basics
  - Request lifecycle
  - Ilog
  - I/O in the OST
  - Idlm
  - quota
> Debugging
Pre-built rpms

• We provide pre-built rpms
  > For 1.6, RHEL4/5, SLES9/10
  > For 1.8 & 2.0, RHEL5/SLES10 and RHEL6/SLES11 (when available)

• Include OFED & TCP support

• Rebuilding rpms is needed if:
  > Need support for another interconnect (Myrinet, ...)
  > Need to apply kernel or lustre patches
Building lustre (server side)

• Kernel patches needed
  > Re-add journal callback support in jbd
  > Jbd fixes & statistics
  > scsi disk statistics
    - could be removed if blktrace enabled
  > Export some symbols used by lustre
  > API for setting block device read-only
  > ...

• First step is to apply those patches & build the patched kernel
  > Use quilt to manage patches
  > Patch series available in lustre/kernel_patches/series
  > Quilt setup /path/to/series, quilt push -a
  > kernel config files in lustre/kernel_patches/kernel_configs
Building lustre (server side)

• Once the kernel is built, we are ready to build the lustre rpms:
  > Get the lustre source
  > ./configure --with-linux=/path/to/kernel ..
  > make rpms

• This produces serveral rpms:
  > lustre-modules: the lustre kernel module
  > lustre-ldiskfs: ext3+patches
  > lustre-$version: utils (mkfs.lustre, mount.lustre, ...)

• Install the patched kernel + lustre/ldiskfs rpms on the servers (OSSs/MDSs)
Building lustre (client side)

• No kernel patches needed
  > except for RHEL4/SLES9
  > You can run the patched kernel on the clients if you wish

• Get the lustre source
  > ./configure --with-linux=/path/to/kernel --disable-server ..
  > make rpms

• Build the lustre rpms as previously:
  > ./configure --with-linux=/path/to/kernel
    --disable-server ..
  > Generate rpms with client only support

• Install the lustre rpms on the client nodes
Building lustre with DMU support

- No change
- ldiskfs rpm replaces by kDMU rpm
- kDMU integrated in lustre source
  > built as part of lustre, like ldiskfs today
  > only needed on OSS/MDS (again as ldiskfs)
Topics

> Building Lustre
> Some protocol basics
  - Request lifecycle
  - llog
  - I/O in the OST
  - ldlm
  - quota
> Debugging
Systems in a Lustre Cluster

Lustre Components

- Clients
  - LOV
  - File open, Directory Operations, metadata, and concurrency
  - Recovery, file status & creation, quota acq/rel
- MDS
- OSS
  - File I/O and file locking
Striping

File open & write

Lustre Client

Linux VFS
Lustre client FS
LOV

OSC 1
OSC 3
MDC

Write (obj 1)
Write (obj 2)

File open request
File metadata
Inode A (obj1, obj2)

OST 1
OST 2
OST 3

Parallel Bandwidth

Odd blocks, even blocks

Metadata Server
Topics

> Building Lustre
> Some protocol basics
  - Request lifecycle
  - llog
  - I/O in the OST
  - ldlm
  - quota
> Debugging
MDS execution

• MDS executes transactions
  > In parallel by multiple threads

• Two stage commit:
  > Commit in memory – after this results are visible
  > Commit on disk – in same order but later
  > This batches the transactions

• Key recovery issue
  > Lustre MDS can lose some transactions
  > Clients need to replay in precisely same order
Request lifecycle

1. Request
2. Execute the request
3a. Reply Ack
3. Reply
4. Commit callback from FS
5. Commit
Client MDS interaction

• Send request
• Request is allocated a transno
• Send reply which includes transno
• Clients acknowledge reply
  > Purpose: MDS knows clients has transno
• Clients keep request & reply
  > Until MDS confirms a disk commit
  > That's where we need commit callback
  > Purpose: client can compensate for lost trans
• MDS has disk data per client
  > Last executed request, last reply information
Commit callbacks

• Run a callback, when disk data commits
• Ability to register & run callbacks has been removed from JBD in 2.6.10
  > Added back by the jbd-jcberr* patches
• Similar mechanism needed for DMU support
Bulk write replay

• Before 1.6.7
  > No replay for bulk write
  > Once the write rpc is acknowledged, data are safely written to disk

• No longer true in versions >= 1.6.8
  > Including 1.8.0
  > Oleg's async journal patch
  > Same scheme as for MDS requests now
Topics

> Building Lustre
> Some protocol basics
  - Request lifecycle
  - llog
  - I/O in the OST
  - ldlm
  - quota
> Debugging
Problem Statement

- Lustre is distributed filesystem
- some POSIX calls change on-disk state on few nodes
- Examples:
  - unlink removes MDS and OST inodes
  - setuid changes owner on MDS and OST
- need to maintain consistent state after failure
Maintaining Consistency: llog

• For distributed transaction commits

• Terminology
  > Initiator – where the transaction is started
  > Replicators – other nodes participating

• Normal operation
  > Write a replay record for each replicator on the initiator
  > Cancel that record after the replicators commit, in bulk
    - Commit callback needed here

• Recovery
  > Process the log entries on the initiator
Use case: unlink

- llog_add()
- llog_cancel()

**LOGCANCEL** => MDS
recovery: READ_LLOG => MDS

MDS ➔ client ➔ OST

- REINT_UNLINK => MDS
- COOKIE => client
- OST_DESTROY + cookie => OST
Use case: unlink (cont'd)

- OST commits objects destroy
  - Then it's time to cancel the MDS llog records
  - Add the cookies to the llog cancel page
  - … truncate the object
  - Start a transaction (fsfilt_start_{log})
  - Remove the object (filter_destroy_internal)
  - Add the commit callback (fsfilt_add_journal_cb)
    - CB is filter_cancel_cookies_cb
  - Finish the transaction (fsfilt_finish_transno)
Topics

> Building Lustre
> Some protocol basics
  - Request lifecycle
  - llog
  - I/O in the OST
  - ldlm
  - quota
> Debugging
I/O in the OST

• The page cache made things too slow in Linux 2.4
• Reserved memory registered for DMA can help
• In 1.6, OSS does non-cached direct IO
  > Nothing ends up in the OSS page cache
• OSS page cache has been resurrected in 1.8
  > For now, only for read
  > Huge performance increase when reading small files back
Topics

> Building Lustre
> Some protocol basics
  - Request lifecycle
  - llog
  - I/O in the OST
  - Idlm
  - quota
> Debugging
Lustre Distributed Lock Manager

- A lock protects a resource
  > Typically, a lock protects something a client caches
- A client enqueues a lock to get it
- An enqueued lock has a client and server copy
- Servers send blocking callbacks to revoke locks
- Servers send completion callbacks to grant locks
- Processes reference granted client locks for use
- Processes de-reference client locks after use
- Clients cancel locks upon callbacks or LRU overflow

- Callbacks were called AST’s in VAX-VMS lingo
- Cancel was de-queue in VAX-VMS lingo
LDLM history

• Basic ideas are similar to VAX DLM
  > You get locks on resources in a namespace
  > All lock calls are asynchronous and get completions
  > There are 6 lock modes with compatibility
  > There are server to client callbacks for notification
  > There are master locks on the “server” and client locks

• Differences
  > We don’t migrate server lock data, except during failover
    - LDLM is more like a collection of lock servers
  > There are extensions to:
    - Handle intents – interpret what the caller wants
    - Handle extents – protect ranges of files
    - Handle lock bits – lock parts of metadata attributes
Client Lock Usage

- DLM locks are acquired over the network
  > The locks are owned by clients of the DLM
    - MGC, OSC & MDC are examples
- Use of locks
  > Locks are given to a particular lock client
  > Processes reference the locks
  > Locks can be canceled only when idle
- Differences
  > Locks are not owned by processes (VAX)
- Servers can take locks also
Lustre Lock Namespaces

- **OST**: namespace to protect object extents.
  - Resources are object ids
  - Extents in the object are “policy data”
- **MDS**: namespace to protect inodes and names
  - FIDs are the resources
  - Lock bits are policy data
  - Intents bundle a VFS operation with its lock requests
- **MGS**: namespace for configuration locks
  - Presently only one resource
  - Protects the entire configuration data
File I/O locks and lock callbacks

• Clients must acquire a read-lock to cache data for read
  > Locks cover an optimistically large file extent
  > Locks are cached on clients

• Before writing, a client obtains a write lock

• Upon concurrent access by another client
  > Client locks see a callback when others want a conflicting lock
  > After the revocation callback arrives, dirty data is flushed
  > Cached data is removed
  > Then the lock is dropped
Client Lock Callback Handling

• Callback function is bound to lock
  > upon client side lock enqueue
  > RPC’s made to the client ldIm service by servers
  > Handed by client lock callback thread : ldIm_cbd

• Completion callback
  > When lock is granted

• Blocking callback
  > Called when servers try to cancel locks in clients
  > Causes cache flush
**Typical Simple Lock Sequence**

- **Sys A:** has Lock on R
- **Sys B:** need Lock on R

**Blocking callback**
- **Sys A:**
  - reply to svc
  - await that lock is inactive
  - flush dirty data,
  - remove cached data
  - cancel the lock

**Lock Enqueue**
- **Lock svc:**
  - reply ASYNC to requestor
  - send callback to holder
  - monitor arrival of cancel

**Lock cancellation**
- **Lock svc:**
  - receive flushed dirty data
  - receive cancellation

**Completion Callback**
- **Sys B:**
  - acquire lock on R
I/O & Locking

• Stripe locking
  > Change from
    - Lock all stripe extents, do all IO in parallel, unlock all
  > To
    - For all stripes in parallel: lock, do IO, unlock
  > Holding locks from multiple servers
    - Can lead to cascading aborts
    - Is necessary for truncate and O_APPEND writes

• Disallow client locks under contention
  > When an extent in a file sees concurrent access
    - Ask the client to write through to the server
  > This eliminates callback traffic and cache flushes
File size and glimpses

• Normal case
  > Only one client does IO to a file, this client knows the size

• Size of file without active IO from any client
  > Currently file size derived from object sizes
  > Will be on the MDS in the future (SOM) - optimal for quiescent files

• Size of a file under active IO
  > Now any client with “far write lock” maybe growing the file
  > A full file write lock would protect the size, but flushes all caches!
    - Lustre does NOT DO THIS, unless the file is not busy
  > In Lustre the OSS’s ask the clients with furthest locks for the size
    - This is a glimpse callback - gives one view of file size
    - A glimpse callback causes clients to cancel locks if they are not using them
  > Glimpsing is the optimal method to get file size during active IO
Configuration Lock

• The central configuration server is the MGS
• When a client fetches a log it also gets a lock
  > The lock gets callbacks when the configuration changes
• Callback triggering events
  > Online addition of OST devices
  > Setting timeouts is global now
  > Many others usage (OST pools creation, quota setup, ....)
  > More robustness fixes
Timeouts and Eviction

- Client requests time out unless a reply is received.
- Client-originated RPC timeouts will cause the client to:
  - Disconnect from the affected server.
  - Ping, reconnect to server or failover and retry/complete operations.
- Server callback RPC timeouts *evict* the affected client.
  - Reconnects to server like an evicted NFS client (not a perfect solution, but OK).
  - The client will learn of eviction during its next request.
  - Upon eviction the client must purge its cache:
    - if data is dirty, this means a small amount of data loss!
  - In-flight network ops will return -EIO to application.
  - Eviction prevents one bad client halting the whole cluster.
Topics

> Building Lustre
> Some protocol basics
  - Request lifecycle
  - llog
  - I/O in the OST
  - ldlm
  - quota
> Debugging
Quota Architecture Primer

• A centralized server hold the cluster wide limits: the quota master(s)
  > guarantees that per-uid/gid global quota limits are not exceeded
  > track quota usage on slaves
  > 1.6/1.8/2.0: single quota master
  > 3.0: multiple quota master required for CMD

• Quota slaves
  > all the OSTs and MDT(s)
  > manage local quota usage/hardlimit
  > acquire/release quota space from the master

• Acquire/release RPC to grant space to slave
  > initiated by slaves & processed by master(s)
  > Early space acquisition to prevent blocking write/create rpcs
Quota Master(s)

- 1.6/1.8/2.0: 1 single master running on the MDS
- 3.0: multiple master required for CMD
- In charge of:
  > storing the quota limits for each uid/gid
  > accounting how much quota space has been granted to slaves
- quota information are stored in administrative quota files
  > files proper to Lustre (admin_quotafile.usr/grp)
  > format identical to the one used in the VFS
Generic Flow of a write request

1. **write request**

2. **sync or async?**
   - **sync**
   - **async**

3. **send write RPC**

4. **is the uid/gid known to be already over quotas?**
   - **no**
   - **yes**

5. **send write RPC**

6. **write data**

7. **write from the grant cache?**
   - **yes**
   - **no**

8. **enough local quota space to satisfy the request?**
   - **yes**
   - **no**

9. **any quota limits for this uid/gid?**
   - **yes**
   - **no**

10. **enough left quota space to grant one more qunit?**
    - **yes**
    - **no**

11. **write RPC completed**

12. **OSS (quota slave)**

13. **MDS (quota master)**

14. **write acknowledged**

15. **send reply**

16. **send dqacq RPC**

17. **trigger early acquisition if needed**

18. **writeback queued**
Quotas support with kDMU

• ZFS currently doesn't support per-uid/gid quotas
  > uses quotas on fileset instead
  > Per-uid/gid quotas is under development

• Future plan
  > Currently lustre quota relies on the linux quota module
  > Implement quota inside lustre instead
  > Relies on ldiskfs/dmu only for block/inode usage accounting
  > Using standard dlm mechanisms to manage both quotas & grant space
Topics

> Building Lustre
> Some protocol basics
  - Request lifecycle
  - Ilog
  - I/O in the OST
  - ldlm
  - quota
> Debugging
Manifestations of trouble

- Problems manifest themselves in multiple ways:
  - An LBUG / Oops / Panic
    - Messages on consoles
    - Modules will not unload
  - A timeout of a client RPC or bulk data transfer
    - Systems are stuck, clients can report timeouts
    - Server threads being stuck (no progress)
  - A timeout of a lock callback (formerly AST)
    - Servers report timeouts
  - Incorrect results
  - Performance is awful
General actions

• Diagnose
  > Check local console and server consoles
    – Oopses, LBUGS and timeouts are found here
  > Server problems can have different manifestation
    – Hung threads, high load, no cpu usage – server thread is stuck
  > /var/log/messages
    – Less common errors may end up here

• Check your network
  > lctl ping
  > LNET Self Test (LST)

• If there is trouble
  > Collect information – Lustre Diagnostics
  > Reboot some nodes to get cluster moving again
LBUG / Oops / Panic

• An LBUG *always* requires a reboot
  > We intentionally hang the thread that LBUGs -- it will never return
  > We do this to make it easy to gather stack traces
    - or if you have a crash dump utility, to examine kernel structures on that task
  > That thread may have locks held
  > In any case, it found something bad

• Oops - a failed kernel assertion
  > an oops will usually kill the thread
  > it may or may not have been fatal to the node
  > you should reboot at your earliest convenience

• Report oops/LBUG output and the events leading to it
LBUG Example

LustreError: 6596:0:(rw.c:159:ll_truncate())
  ASSERTION(atomic_read(&lli->lli_size_sem.count) <= 0) failed
LustreError: 6596:0:(module.c:46:kportal_assertion_failed()) LBUG
LustreError: dumping log to
  /tmp/lustre-log-b2.boston.clusterfs.com.1108864884.6596

• Post-process the log:
  lctl df /tmp/lustre-log-b2.boston.clusterfs.com.1108864884.6596
  /tmp/foo

• Collect other information
  > See next section
  > File in a bug at Sun
Reboot some nodes

- Reboot OSS or MDS – minor if any consequences
  > Before you do this collect the bug information – see later
- Reboot a stuck client – quite safe
  > `umount` – unmount client
    - may hang & disconnects only once
  > `umount -f` – client will not attempt to disconnect
Check the Console First

- It might have your answer
- Include messages with any bug report or support request
- In many cases, this is Lustre’s only way to communicate
  - `dmesg`
  - `/var/log/messages`
Check *Other* Consoles

- Lustre is an enormous distributed system
- Most problems involve multiple nodes
- Chances are, the log will tell you which nodes:

```
LustreError: Connection to service ost2_svc (on 192.168.0.107) was lost (timeout waiting for reply); in progress operations using this service will wait for reconnection
```

```
LustreError: This client was evicted by ost2_svc (on 192.168.0.107); in progress operations using this service will fail.
```
Lkcd / kdump / netdump

- These tools were an amazing benefit early on
- They pay for themselves with the first 1-in-a-million crash
- Historically, its stack traces are more trustworthy than SysRq-T
- You can also examine data structures in the kernel
- You can also examine live, running kernels
- We sometimes ask customer to upload crash dumps to our ftp site (if possible)
  > We are very familiar with crash/lcrash
Gathering and filing bug information

SysRq

• Turn it on:
  > /etc/sysctl.conf, add “kernel.sysrq=1”
  > sysctl -w kernel.sysrq=1
  > Trigger it with /proc/sysrq-trigger

• SysRq-P (one stack trace) is usually uninteresting
• SysRq-T (all stack traces) is voluminous but very useful
  > Especially if a process is hung and won’t make progress

• SysRq-M (memory info) is sometimes enlightening
  > Is the system essentially out of memory?
  > Are any of the counters impossible values?

• ps is often useless – the kernel “D” state is not unique
  > It means “uninterruptible sleep”
  > It’s interesting to know, but could be *anything*.
  > Get the Sysrq-T traces!
Collecting Lustre Debug Logs

• Lustre keeps a ring-buffer of pages in the kernel
  > by default, 5 MB/CPU
  > /proc/sys/portals/debug_mb

• /proc/sys/lnet/debug is a bitmask
  > Let's turn on and off some kinds of messages
  > We may ask you to modify this before reproducing a problem
  > The default is not bad for production use, but you might try others

• These logs are extremely user-unfriendly
Getting a Debug Log

- Sometimes the system volunteers a debug log after a LBUG
- Other times we’ll ask you to generate one
- If we do, please clear the buffers before you reproduce:
  > lctl clear
- 5 MB sounds like a lot, but it’s usually not.
  > These logs are incredibly verbose.
  > Try to have as little running alongside your test as possible.
Post-processing

• If you get a log the normal way:
  `lctl dk [filename]`
  ...then `lctl` post-processes it for you.
• If the kernel dumps it on its own (e.g., an LBUG):
  `lctl df INFILE [OUTFILE]`
• Please do this before you send it to us.
Lock Dump

• You can get a complete lock dump in the logs
• Only visible if DLMTRACE is enabled in portals/debug

```bash
echo > /proc/fs/lustre/ldlm/dump_namespaces
```

• Sometimes need a lock snapshot from several nodes

--- Resource: c277aa80 (717958/0/0/0) (rc: 1)
Granted locks:

-- Lock dump: c8175280/0xa6f5f87dbc6b3693 (rc: 1) (pos: 1) Node: NID 0:192.168.0.3 on socknal (rhandle: 0x7899f232a33d8fb8)
Resource: c277aa80 (717958/0)
Req mode: PR, grant mode: PR, rc: 1, read: 0, write: 0
Extent: 0 -> 18446744073709551615 (req 253112320-253128703)
Questions?

johann@sun.com