Lustre* Software Release 2.x

Operations Manual

Lustre* Software Release 2.x: Operations Manual

Important Notice from Intel

Important Notice from Oracle

Table of Contents

List of Figures

List of Tables

Preface

The *Lustre*Software Release 2.x Operations Manual* provides detailed information and procedures to install, configure and tune a Lustre file system. The manual covers topics such as failover, quotas, striping, and bonding. This manual also contains troubleshooting information and tips to improve the operation and performance of a Lustre file system.

1. About this Document

This document is maintained by Intel in Docbook format. The canonical version is available at http://wiki.hpdd.intel.com/display/PUB/Documentation.

1.1. UNIX* Commands

This document may not contain information about basic UNIX* operating system commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following for this information:

- Software documentation that you received with your system
- Red Hat* Enterprise Linux* documentation, which is at: http://docs.redhat.com/docs/en-US/index.html

Note

The Lustre client module is available for many different Linux* versions and distributions. The Red Hat Enterprise Linux distribution is the best supported and tested platform for Lustre servers.

1.2. Shell Prompts

Shell	Prompt
C shell	*machine-name*%
C shell superuser	*machine-name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

1.3. Related Documentation

Application	Title	Format	Location
Latest information	*Lustre Software Release 2.x Change Logs*	Wiki page	Online at http://wiki.hpdd.intel.com/display/PUB/Documentation
Service	*Lustre Software Release 2.x Operations Manual*	PDF HTML	Online at http://wiki.hpdd.intel.com/display/PUB/Documentation

1.4. Documentation, Support, and Training

These web sites provide additional resources:

- Documentation http://wiki.hpdd.intel.com/display/PUB/Documentation
- Support http://www.hpdd.intel.com/
- Training http://www.hpdd.intel.com/

2. Revisions

The Lustre* File System Release 2.x Operations Manual is a community maintained work. Versions of the manual are continually built as suggestions for changes and improvements arrive. Suggestions for improvements can be submitted through the ticketing system maintained at https://jira.hpdd.intel.com/browse/LUDOC [http://jira.hpdd.intel.com/browse/LUDOC]. Instructions for providing a patch to the existing manual are available at: https://wiki.hpdd.intel.com/display/PUB/Making+changes+to+the+Lustre+Manual+source.

This manual currently covers all the 2.x Lustre software releases. Features that are specific to individual releases are identified within the table of contents using a short hand notation (i.e. 'L24' is a Lustre software release 2.4 specific feature), and within the text using a distinct box. For example:

Introduced in Lustre 2.4

Lustre software release version 2.4 includes support for multiple metadata servers.

Only the latest revision of this document is made readily available because changes are continually arriving. The current and latest revision of this manual is available from links maintained at: http://lustre.opensfs.org/documentation/.

Revision History

Revision 0	Built 09 April 2014 09:01:49-07:00	onIntel Corporation

Continuous build of Manual.

Part I. Introducing the Lustre* File System

Part I provides background information to help you understand the Lustre file system architecture and how the major components fit together. You will find information in this section about:

- Understanding Lustre Architecture
- Understanding Lustre Networking (LNET)
- Understanding Failover in a Lustre File System

Chapter 1. Understanding Lustre Architecture

This chapter describes the Lustre architecture and features of the Lustre file system. It includes the following sections:

- Section 1.1, “What a Lustre File System Is (and What It Isn't)”
- Section 1.2, “Lustre Components”
- Section 1.3, “ Lustre File System Storage and I/O”

1.1. What a Lustre File System Is (and What It Isn't)

The Lustre architecture is a storage architecture for clusters. The central component of the Lustre architecture is the Lustre file system, which is supported on the Linux operating system and provides a POSIX* standard-compliant UNIX file system interface.

The Lustre storage architecture is used for many different kinds of clusters. It is best known for powering many of the largest high-performance computing (HPC) clusters worldwide, with tens of thousands of client systems, petabytes (PB) of storage and hundreds of gigabytes per second (GB/sec) of I/O throughput. Many HPC sites use a Lustre file system as a site-wide global file system, serving dozens of clusters.

The ability of a Lustre file system to scale capacity and performance for any need reduces the need to deploy many separate file systems, such as one for each compute cluster. Storage management is simplified by avoiding the need to copy data between compute clusters. In addition to aggregating storage capacity of many servers, the I/O throughput is also aggregated and scales with additional servers. Moreover, throughput and/or capacity can be easily increased by adding servers dynamically.

While a Lustre file system can function in many work environments, it is not necessarily the best choice for all applications. It is best suited for uses that exceed the capacity that a single server can provide, though in some use cases, a Lustre file system can perform better with a single server than other file systems due to its strong locking and data coherency.

A Lustre file system is currently not particularly well suited for "peer-to-peer" usage models where clients and servers are running on the same node, each sharing a small amount of storage, due to the lack of data replication at the Lustre software level. In such uses, if one client/server fails, then the data stored on that node will not be accessible until the node is restarted.

1.1.1. Lustre Features

Lustre file systems run on a variety of vendor's kernels. For more details, see the Lustre Test Matrix Section 8.1, “ Preparing to Install the Lustre Software”.

A Lustre installation can be scaled up or down with respect to the number of client nodes, disk storage and bandwidth. Scalability and performance are dependent on available disk and network bandwidth and the processing power of the servers in the system. A Lustre file system can be deployed in a wide variety of configurations that can be scaled well beyond the size and performance observed in production systems to date.

Table 1.1, “Lustre File System Scalability and Performance” shows the practical range of scalability and performance characteristics of a Lustre file system and some test results in production systems.

Table 1.1. Lustre File System Scalability and Performance

Feature	Current Practical Range	Tested in Production
Client Scalability	100-100000	50000+ clients, many in the 10000 to 20000 range
Client Performance	*Single client:* I/O 90% of network bandwidth *Aggregate:* 2.5 TB/sec I/O	*Single client:* 2 GB/sec I/O, 1000 metadata ops/sec *Aggregate:* 240 GB/sec I/O
OSS Scalability	*Single OSS:* 1-32 OSTs per OSS, 128TB per OST *OSS count:* 500 OSSs, with up to 4000 OSTs	*Single OSS:* 8 OSTs per OSS, 16TB per OST *OSS count:* 450 OSSs with 1000 4TB OSTs 192 OSSs with 1344 8TB OSTs
OSS Performance	*Single OSS:* 5 GB/sec *Aggregate:* 2.5 TB/sec	*Single OSS:* 2.0+ GB/sec *Aggregate:* 240 GB/sec
MDS Scalability	*Single MDS:* 4 billion files *MDS count:* 1 primary + 1 backup Introduced in Lustre 2.4 *Since Lustre software release 2.4:* Introduced in Lustre 2.4 Up to 4096 MDSs and up to 4096 MDTs	*Single MDS:* 750 million files *MDS count:* 1 primary + 1 backup
MDS Performance	35000/s create operations, 100000/s metadata stat operations	15000/s create operations, 35000/s metadata stat operations
File system Scalability	*Single File:* 2.5 PB max file size	*Single File:* multi-TB max file size

Feature	Current Practical Range	Tested in Production
	Aggregate: 512 PB space, 4 billion files	*Aggregate:* 10 PB space, 750 million files

Other Lustre software features are:

- **Performance-enhanced ext4 file system:** The Lustre file system uses an improved version of the ext4 journaling file system to store data and metadata. This version, called *ldiskfs*, has been enhanced to improve performance and provide additional functionality needed by the Lustre file system.

- **POSIX standard compliance:** The full POSIX test suite passes in an identical manner to a local ext4 file system, with limited exceptions on Lustre clients. In a cluster, most operations are atomic so that clients never see stale data or metadata. The Lustre software supports mmap() file I/O.

- **High-performance heterogeneous networking:** The Lustre software supports a variety of high performance, low latency networks and permits Remote Direct Memory Access (RDMA) for InfiniBand* (utilizing OpenFabrics Enterprise Distribution (OFED*) and other advanced networks for fast and efficient network transport. Multiple RDMA networks can be bridged using Lustre routing for maximum performance. The Lustre software also includes integrated network diagnostics.

- **High-availability:** The Lustre file system supports active/active failover using shared storage partitions for OSS targets (OSTs). Lustre software release 2.3 and earlier releases offer active/passive failover using a shared storage partition for the MDS target (MDT).

  Introduced in Lustre 2.4

  With Lustre software release 2.4 or later servers and clients it is possible to configure active/active failover of multiple MDTs. This allows application transparent recovery. The Lustre file system can work with a variety of high availability (HA) managers to allow automated failover and has no single point of failure (NSPF). Multiple mount protection (MMP) provides integrated protection from errors in highly-available systems that would otherwise cause file system corruption.

- **Security:** By default TCP connections are only allowed from privileged ports. UNIX group membership is verified on the MDS.

- **Access control list (ACL), extended attributes:** the Lustre security model follows that of a UNIX file system, enhanced with POSIX ACLs. Noteworthy additional features include root squash.

- **Interoperability:** The Lustre file system runs on a variety of CPU architectures and mixed-endian clusters and is interoperable between successive major Lustre software releases.

- **Object-based architecture:** Clients are isolated from the on-disk file structure enabling upgrading of the storage architecture without affecting the client.

- **Byte-granular file and fine-grained metadata locking:** Many clients can read and modify the same file or directory concurrently. The Lustre distributed lock manager (LDLM) ensures that files are coherent between all clients and servers in the file system. The MDT LDLM manages locks on inode permissions and pathnames. Each OST has its own LDLM for locks on file stripes stored thereon, which scales the locking performance as the file system grows.

- **Quotas:** User and group quotas are available for a Lustre file system.

- **Capacity growth:** The size of a Lustre file system and aggregate cluster bandwidth can be increased without interruption by adding a new OSS with OSTs to the cluster.

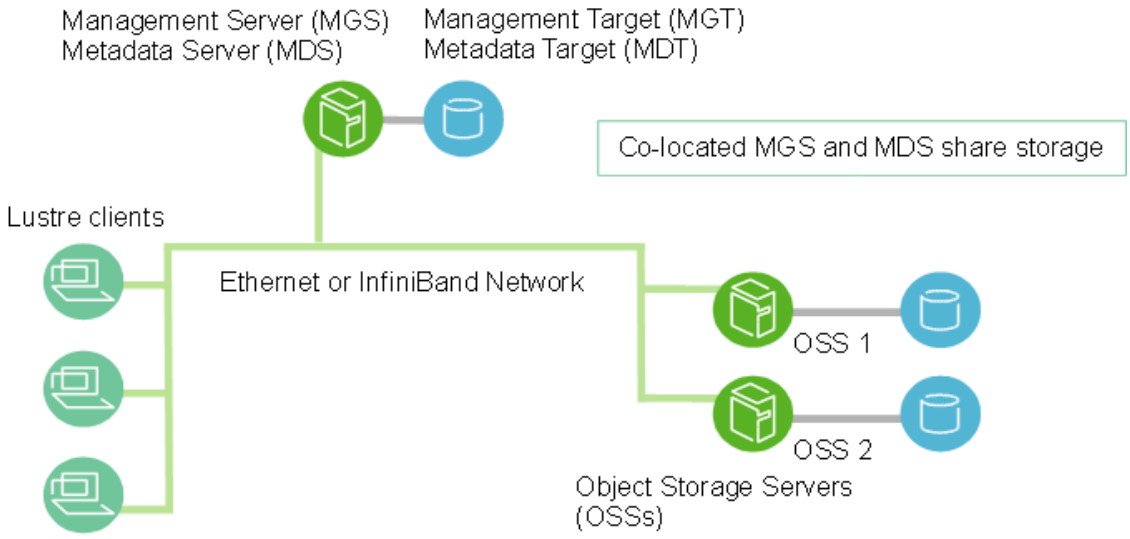
- **Controlled striping:** The layout of files across OSTs can be configured on a per file, per directory, or per file system basis. This allows file I/O to be tuned to specific application requirements within a single file system. The Lustre file system uses RAID-0 striping and balances space usage across OSTs.
- **Network data integrity protection:** A checksum of all data sent from the client to the OSS protects against corruption during data transfer.
- **MPI I/O:** The Lustre architecture has a dedicated MPI ADIO layer that optimizes parallel I/O to match the underlying file system architecture.
- **NFS and CIFS export:** Lustre files can be re-exported using NFS (via Linux knfsd) or CIFS (via Samba) enabling them to be shared with non-Linux clients, such as Microsoft* Windows* and Apple* Mac OS X*.
- **Disaster recovery tool:** The Lustre file system provides an online distributed file system check (LFSCK) that can restore consistency between storage components in case of a major file system error. A Lustre file system can operate even in the presence of file system inconsistencies, and LFSCK can run while the filesystem is in use, so LFSCK is not required to complete before returning the file system to production.
- **Performance monitoring:** The Lustre file system offers a variety of mechanisms to examine performance and tuning.
- **Open source:** The Lustre software is licensed under the GPL 2.0 license for use with the Linux operating system.

1.2. Lustre Components

An installation of the Lustre software includes a management server (MGS) and one or more Lustre file systems interconnected with Lustre networking (LNET).

A basic configuration of Lustre file system components is shown in Figure 1.1, “Lustre file system components in a basic cluster ”.

Figure 1.1. Lustre file system components in a basic cluster



1.2.1. Management Server (MGS)

The MGS stores configuration information for all the Lustre file systems in a cluster and provides this information to other Lustre components. Each Lustre target contacts the MGS to provide information, and Lustre clients contact the MGS to retrieve information.

It is preferable that the MGS have its own storage space so that it can be managed independently. However, the MGS can be co-located and share storage space with an MDS as shown in Figure 1.1, “Lustre file system components in a basic cluster ”.

1.2.2. Lustre File System Components

Each Lustre file system consists of the following components:

- **Metadata Server (MDS)** - The MDS makes metadata stored in one or more MDTs available to Lustre clients. Each MDS manages the names and directories in the Lustre file system(s) and provides network request handling for one or more local MDTs.

- **Metadata Target (MDT**) - For Lustre software release 2.3 and earlier, each file system has one MDT. The MDT stores metadata (such as filenames, directories, permissions and file layout) on storage attached to an MDS. Each file system has one MDT. An MDT on a shared storage target can be available to multiple MDSs, although only one can access it at a time. If an active MDS fails, a standby MDS can serve the MDT and make it available to clients. This is referred to as MDS failover.

  Introduced in Lustre 2.4

  Since Lustre software release 2.4, multiple MDTs are supported. Each file system has at least one MDT. An MDT on a shared storage target can be available via multiple MDSs, although only one MDS can export the MDT to the clients at one time. Two MDS machines share storage for two or more MDTs. After the failure of one MDS, the remaining MDS begins serving the MDT(s) of the failed MDS.

- **Object Storage Servers (OSS)** : The OSS provides file I/O service and network request handling for one or more local OSTs. Typically, an OSS serves between two and eight OSTs, up to 16 TB each. A typical configuration is an MDT on a dedicated node, two or more OSTs on each OSS node, and a client on each of a large number of compute nodes.

- **Object Storage Target (OST)** : User file data is stored in one or more objects, each object on a separate OST in a Lustre file system. The number of objects per file is configurable by the user and can be tuned to optimize performance for a given workload.

- **Lustre clients** : Lustre clients are computational, visualization or desktop nodes that are running Lustre client software, allowing them to mount the Lustre file system.

The Lustre client software provides an interface between the Linux virtual file system and the Lustre servers. The client software includes a management client (MGC), a metadata client (MDC), and multiple object storage clients (OSCs), one corresponding to each OST in the file system.

A logical object volume (LOV) aggregates the OSCs to provide transparent access across all the OSTs. Thus, a client with the Lustre file system mounted sees a single, coherent, synchronized namespace. Several clients can write to different parts of the same file simultaneously, while, at the same time, other clients can read from the file.

Table 1.2, “Storage and hardware requirements for Lustre file system components” provides the requirements for attached storage for each Lustre file system component and describes desirable characteristics of the hardware used.

Table 1.2. Storage and hardware requirements for Lustre file system components

	Required attached storage	Desirable hardware characteristics
MDSs	1-2% of file system capacity	Adequate CPU power, plenty of memory, fast disk storage.
OSSs	1-16 TB per OST, 1-8 OSTs per OSS	Good bus bandwidth. Recommended that storage be balanced evenly across OSSs.
Clients	None	Low latency, high bandwidth network.

For additional hardware requirements and considerations, see Chapter 5, *Determining Hardware Configuration Requirements and Formatting Options*.

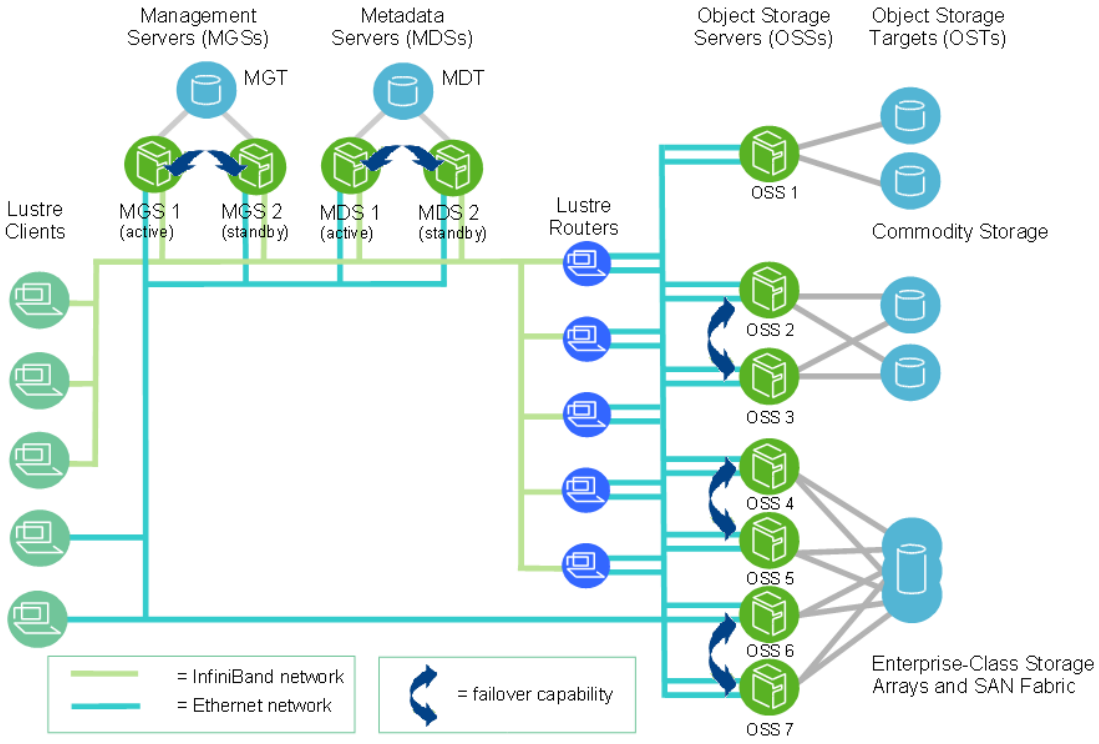
1.2.3. Lustre Networking (LNET)

Lustre Networking (LNET) is a custom networking API that provides the communication infrastructure that handles metadata and file I/O data for the Lustre file system servers and clients. For more information about LNET, see Chapter 2, *Understanding Lustre Networking (LNET)*.

1.2.4. Lustre Cluster

At scale, a Lustre file system cluster can include hundreds of OSSs and thousands of clients (see Figure 1.2, “Lustre cluster at scale”). More than one type of network can be used in a Lustre cluster. Shared storage between OSSs enables failover capability. For more details about OSS failover, see Chapter 3, *Understanding Failover in a Lustre File System*.

Figure 1.2. Lustre cluster at scale



1.3. Lustre File System Storage and I/O

In Lustre software release 2.0, Lustre file identifiers (FIDs) were introduced to replace UNIX inode numbers for identifying files or objects. A FID is a 128-bit identifier that contains a unique 64-bit sequence number, a 32-bit object ID (OID), and a 32-bit version number. The sequence number is unique across all Lustre targets in a file system (OSTs and MDTs). This change enabled future support for multiple MDTs (introduced in Lustre software release 2.3) and ZFS (introduced in Lustre software release 2.4).

Also introduced in release 2.0 is a feature call *FID-in-dirent* (also known as *dirdata*) in which the FID is stored as part of the name of the file in the parent directory. This feature significantly improves performance for `ls` command executions by reducing disk I/O. The FID-in-dirent is generated at the time the file is created.

Note

The FID-in-dirent feature is not compatible with the Lustre software release 1.8 format. Therefore, when an upgrade from Lustre software release 1.8 to a Lustre software release 2.x is performed, the FID-in-dirent feature is not automatically enabled. For upgrades from Lustre software release 1.8 to Lustre software releases 2.0 through 2.3, FID-in-dirent can be enabled manually but only takes effect for new files.

For more information about upgrading from Lustre software release 1.8 and enabling FID-in-dirent for existing files, see Chapter 16, *Upgrading a Lustre File System*Chapter 16 “Upgrading a Lustre File System”.

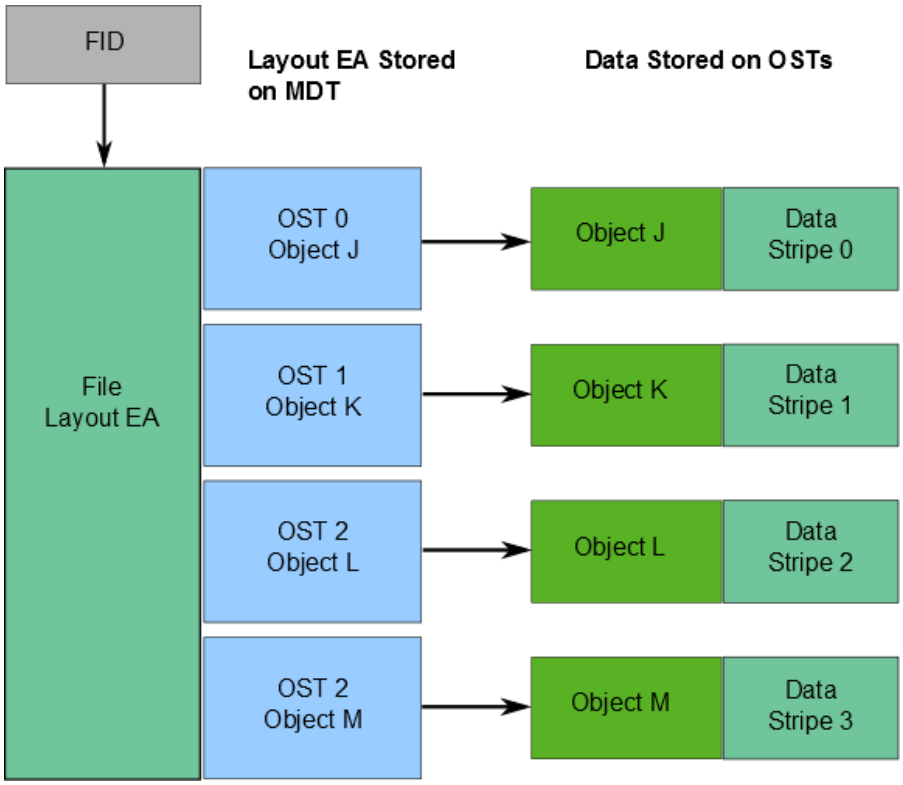
Introduced in Lustre 2.4

The LFSCK 1.5 file system administration tool released with Lustre software release 2.4 provides functionality that enables FID-in-dirent for existing files. It includes the following functionality:

- Generates IGIF mode FIDs for existing release 1.8 files.
- Verifies the FID-in-dirent for each file to determine when it doesn’t exist or is invalid and then regenerates the FID-in-dirent if needed.
- Verifies the linkEA entry for each file to determine when it is missing or invalid and then regenerates the linkEA if needed. The *linkEA* consists of the file name plus its parent FID and is stored as an extended attribute in the file itself. Thus, the linkEA can be used to parse out the full path name of a file from root.

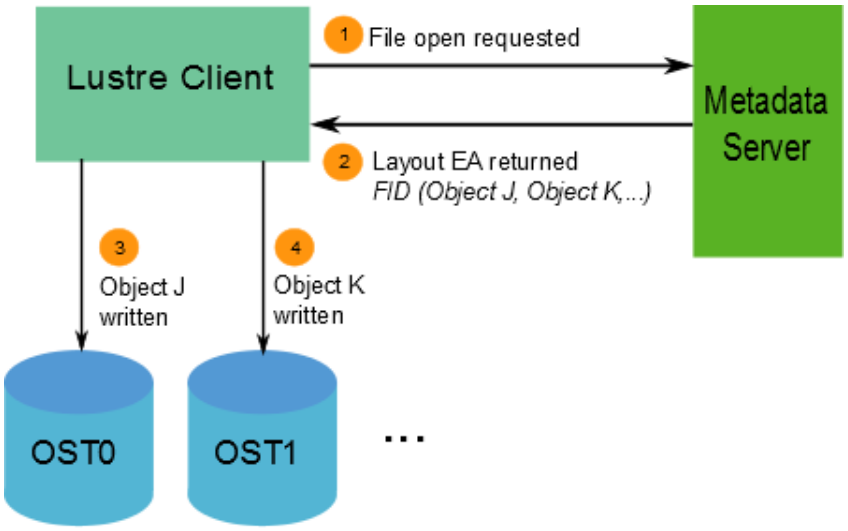
Information about where file data is located on the OST(s) is stored as an extended attribute called layout EA in an MDT object identified by the FID for the file (see Figure 1.3, “Layout EA on MDT pointing to file data on OSTs”). If the file is a data file (not a directory or symbol link), the MDT object points to 1-to-N OST object(s) on the OST(s) that contain the file data. If the MDT file points to one object, all the file data is stored in that object. If the MDT file points to more than one object, the file data is *striped* across the objects using RAID 0, and each object is stored on a different OST. (For more information about how striping is implemented in a Lustre file system, see Section 1.3.1, “ Lustre File System and Striping”.

Figure 1.3. Layout EA on MDT pointing to file data on OSTs



When a client wants to read from or write to a file, it first fetches the layout EA from the MDT object for the file. The client then uses this information to perform I/O on the file, directly interacting with the OSS nodes where the objects are stored. This process is illustrated in Figure 1.4, “Lustre client requesting file data”.

Figure 1.4. Lustre client requesting file data



The available bandwidth of a Lustre file system is determined as follows:

- The *network bandwidth* equals the aggregated bandwidth of the OSSs to the targets.

- The *disk bandwidth* equals the sum of the disk bandwidths of the storage targets (OSTs) up to the limit of the network bandwidth.
- The *aggregate bandwidth* equals the minimum of the disk bandwidth and the network bandwidth.
- The *available file system space* equals the sum of the available space of all the OSTs.

1.3.1. Lustre File System and Striping

One of the main factors leading to the high performance of Lustre file systems is the ability to stripe data across multiple OSTs in a round-robin fashion. Users can optionally configure for each file the number of stripes, stripe size, and OSTs that are used.

Striping can be used to improve performance when the aggregate bandwidth to a single file exceeds the bandwidth of a single OST. The ability to stripe is also useful when a single OST does not have enough free space to hold an entire file. For more information about benefits and drawbacks of file striping, see Section 18.2, “ Lustre File Layout (Striping) Considerations”.

Striping allows segments or 'chunks' of data in a file to be stored on different OSTs, as shown in Figure 1.5, “File striping on a Lustre file system”. In the Lustre file system, a RAID 0 pattern is used in which data is "striped" across a certain number of objects. The number of objects in a single file is called the `stripe_count`.

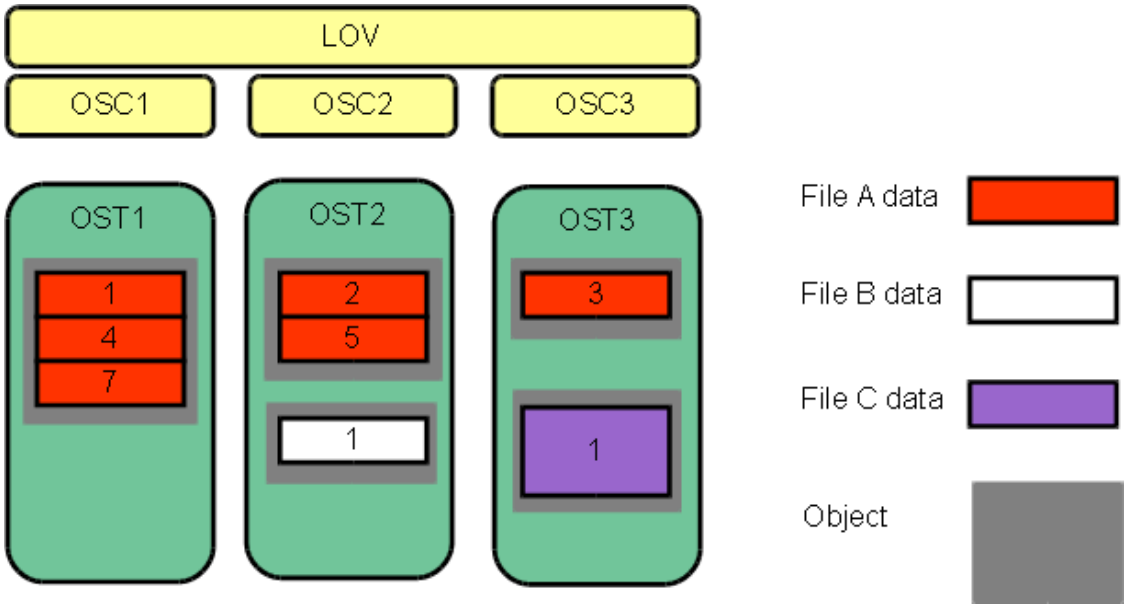
Each object contains a chunk of data from the file. When the chunk of data being written to a particular object exceeds the `stripe_size`, the next chunk of data in the file is stored on the next object.

Default values for `stripe_count` and `stripe_size` are set for the file system. The default value for `stripe_count` is 1 stripe for file and the default value for `stripe_size` is 1MB. The user may change these values on a per directory or per file basis. For more details, see Section 18.3, “Setting the File Layout/Striping Configuration (`lfs setstripe`)”.

Figure 1.5, “File striping on a Lustre file system”, the `stripe_size` for File C is larger than the `stripe_size` for File A, allowing more data to be stored in a single stripe for File C. The `stripe_count` for File A is 3, resulting in data striped across three objects, while the `stripe_count` for File B and File C is 1.

No space is reserved on the OST for unwritten data. File A in Figure 1.5, “File striping on a Lustre file system”.

Figure 1.5. File striping on a Lustre file system



The maximum file size is not limited by the size of a single target. In a Lustre file system, files can be striped across multiple objects (up to 2000), and each object can be up to 16 TB in size with ldiskfs. This leads to a maximum file size of 31.25 PB. (Note that a Lustre file system can support files up to 2^64 bytes depending on the backing storage used by OSTs.)

Note

Versions of the Lustre software prior to Release 2.2 limited the maximum stripe count for a single file to 160 OSTs.

Although a single file can only be striped over 2000 objects, Lustre file systems can have thousands of OSTs. The I/O bandwidth to access a single file is the aggregated I/O bandwidth to the objects in a file, which can be as much as a bandwidth of up to 2000 servers. On systems with more than 2000 OSTs, clients can do I/O using multiple files to utilize the full file system bandwidth.

For more information about striping, see Chapter 18, *Managing File Layout (Striping) and Free Space*.

Chapter 2. Understanding Lustre Networking (LNET)

This chapter introduces Lustre networking (LNET). It includes the following sections:

- Section 2.1, " Introducing LNET"
- Section 2.2, "Key Features of LNET"
- Section 2.3, "Lustre Networks"
- Section 2.4, "Supported Network Types"

2.1. Introducing LNET

In a cluster using one or more Lustre file systems, the network communication infrastructure required by the Lustre file system is implemented using the Lustre networking (LNET) feature.

LNET supports many commonly-used network types, such as InfiniBand and IP networks, and allows simultaneous availability across multiple network types with routing between them. Remote direct memory access (RDMA) is permitted when supported by underlying networks using the appropriate Lustre network driver (LND). High availability and recovery features enable transparent recovery in conjunction with failover servers.

An LND is a pluggable driver that provides support for a particular network type, for example `ksocklnd` is the driver which implements the TCP Socket LND that supports TCP networks. LNDs are loaded into the driver stack, with one LND for each network type in use.

For information about configuring LNET, see Chapter 9, *Configuring Lustre Networking (LNET)* .

For information about administering LNET, see Part III, "Administering Lustre".

2.2. Key Features of LNET

Key features of LNET include:

- RDMA, when supported by underlying networks
- Support for many commonly-used network types
- High availability and recovery
- Support of multiple network types simultaneously
- Routing among disparate networks

LNET permits end-to-end read/write throughput at or near peak bandwidth rates on a variety of network interconnects.

2.3. Lustre Networks

A Lustre network is comprised of clients and servers running the Lustre software. It need not be confined to one LNET subnet but can span several networks provided routing is possible between the networks. In a similar manner, a single network can have multiple LNET subnets.

The Lustre networking stack is comprised of two layers, the LNET code module and the LND. The LNET layer operates above the LND layer in a manner similar to the way the network layer operates above the data link layer. LNET layer is connectionless, asynchronous and does not verify that data has been transmitted while the LND layer is connection oriented and typically does verify data transmission.

LNETs are uniquely identified by a label comprised of a string corresponding to an LND and a number, such as tcp0, o2ib0, or o2ib1, that uniquely indentifies each LNET. Each node on an LNET has at least one network identifier (NID). A NID is a combination of the address of the network interface and the LNET label in the form:*address@LNET_label*.

Examples:

```
192.168.1.2@tcp0
10.13.24.90@o2ib1
```

In certain circumstances it might be desirable for Lustre file system traffic to pass between multiple LNETs. This is possible using LNET routing. It is important to realize that LNET routing is not the same as network routing. For more details about LNET routing, see Chapter 9, *Configuring Lustre Networking (LNET)*

2.4. Supported Network Types

The LNET code module includes LNDs to support many network types including:

- InfiniBand: OpenFabrics OFED (o2ib)
- TCP (any network carrying TCP traffic, including GigE, 10GigE, and IPoIB)
- Cray: Seastar
- Myrinet: MX
- RapidArray: ra
- Quadrics: Elan

Chapter 3. Understanding Failover in a Lustre File System

This chapter describes failover in a Lustre file system. It includes:

- Section 3.1, " What is Failover?"
- Section 3.2, "Failover Functionality in a Lustre File System"

3.1. What is Failover?

In a high-availability (HA) system, unscheduled downtime is minimized by using redundant hardware and software components and software components that automate recovery when a failure occurs. If a failure condition occurs, such as the loss of a server or storage device or a network or software fault, the system's services continue with minimal interruption. Generally, availability is specified as the percentage of time the system is required to be available.

Availability is accomplished by replicating hardware and/or software so that when a primary server fails or is unavailable, a standby server can be switched into its place to run applications and associated resources. This process, called *failover*, is automatic in an HA system and, in most cases, completely application-transparent.

A failover hardware setup requires a pair of servers with a shared resource (typically a physical storage device, which may be based on SAN, NAS, hardware RAID, SCSI or Fibre Channel (FC) technology). The method of sharing storage should be essentially transparent at the device level; the same physical logical unit number (LUN) should be visible from both servers. To ensure high availability at the physical storage level, we encourage the use of RAID arrays to protect against drive-level failures.

Note

The Lustre software does not provide redundancy for data; it depends exclusively on redundancy of backing storage devices. The backing OST storage should be RAID 5 or, preferably, RAID 6 storage. MDT storage should be RAID 1 or RAID 10.

3.1.1. Failover Capabilities

To establish a highly-available Lustre file system, power management software or hardware and high availability (HA) software are used to provide the following failover capabilities:

- **Resource fencing** - Protects physical storage from simultaneous access by two nodes.
- **Resource management** - Starts and stops the Lustre resources as a part of failover, maintains the cluster state, and carries out other resource management tasks.
- **Health monitoring** - Verifies the availability of hardware and network resources and responds to health indications provided by the Lustre software.

These capabilities can be provided by a variety of software and/or hardware solutions. For more information about using power management software or hardware and high availability (HA) software with a Lustre file system, see Chapter 11, *Configuring Failover in a Lustre File System.*

HA software is responsible for detecting failure of the primary Lustre server node and controlling the failover.The Lustre software works with any HA software that includes resource (I/O) fencing. For proper resource fencing, the HA software must be able to completely power off the failed server or disconnect it from the shared storage device. If two active nodes have access to the same storage device, data may be severely corrupted.

3.1.2. Types of Failover Configurations

Nodes in a cluster can be configured for failover in several ways. They are often configured in pairs (for example, two OSTs attached to a shared storage device), but other failover configurations are also possible. Failover configurations include:

- **Active/passive** pair - In this configuration, the active node provides resources and serves data, while the passive node is usually standing by idle. If the active node fails, the passive node takes over and becomes active.
- **Active/active** pair - In this configuration, both nodes are active, each providing a subset of resources. In case of a failure, the second node takes over resources from the failed node.

In Lustre software releases previous to Lustre software release 2.4, MDSs can be configured as an active/passive pair, while OSSs can be deployed in an active/active configuration that provides redundancy without extra overhead. Often the standby MDS is the active MDS for another Lustre file system or the MGS, so no nodes are idle in the cluster.

Introduced in Lustre 2.4

Lustre software release 2.4 introduces metadata targets for individual sub-directories. Active-active failover configurations are available for MDSs that serve MDTs on shared storage.

3.2. Failover Functionality in a Lustre File System

The failover functionality provided by the Lustre software can be used for the following failover scenario. When a client attempts to do I/O to a failed Lustre target, it continues to try until it receives an answer from any of the configured failover nodes for the Lustre target. A user-space application does not detect anything unusual, except that the I/O may take longer to complete.

Failover in a Lustre file system requires that two nodes be configured as a failover pair, which must share one or more storage devices. A Lustre file system can be configured to provide MDT or OST failover.

- For MDT failover, two MDSs can be configured to serve the same MDT. Only one MDS node can serve an MDT at a time.

  Introduced in Lustre 2.4

  Lustresoftware release 2.4 allows multiple MDTs. By placing two or more MDT partitions on storage shared by two MDSs, one MDS can fail and the remaining MDS can begin serving the unserved MDT. This is described as an active/active failover pair.

- For OST failover, multiple OSS nodes can be configured to be able to serve the same OST. However, only one OSS node can serve the OST at a time. An OST can be moved between OSS nodes that have access to the same storage device using `umount/mount` commands.

The `--servicenode` option is used to set up nodes in a Lustre file system for failover at creation time (using `mkfs.lustre`) or later when the Lustre file system is active (using `tunefs.lustre`). For explanations of these utilities, see Section 37.14, “ mkfs.lustre” and Section 37.18, “ tunefs.lustre”.

Failover capability in a Lustre file system can be used to upgrade the Lustre software between successive minor versions without cluster downtime. For more information, see Chapter 16, *Upgrading a Lustre File System*.

For information about configuring failover, see Chapter 11, *Configuring Failover in a Lustre File System*.

Note

The Lustre software provides failover functionality only at the file system level. In a complete failover solution, failover functionality for system-level components, such as node failure detection or power control, must be provided by a third-party tool.

Caution

OST failover functionality does not protect against corruption caused by a disk failure. If the storage media (i.e., physical disk) used for an OST fails, it cannot be recovered by functionality provided in the Lustre software. We strongly recommend that some form of RAID be used for OSTs. Lustre functionality assumes that the storage is reliable, so it adds no extra reliability features.

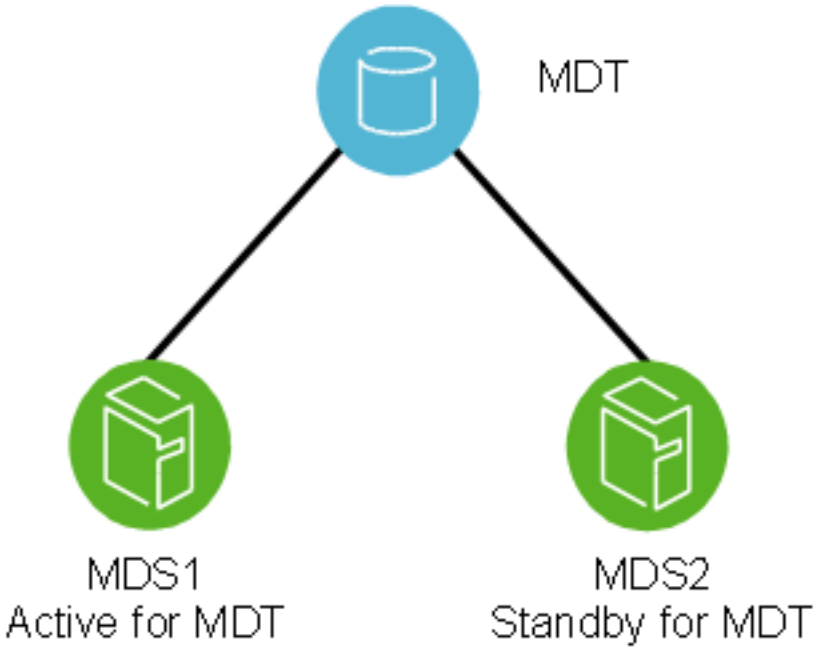
3.2.1. MDT Failover Configuration (Active/Passive)

Two MDSs are typically configured as an active/passive failover pair as shown in Figure 3.1, " Lustre failover configuration for a active/passive MDT". Note that both nodes must have access to shared storage for the MDT(s) and the MGS. The primary (active) MDS manages the Lustre system metadata resources. If the primary MDS fails, the secondary (passive) MDS takes over these resources and serves the MDTs and the MGS.

Note

In an environment with multiple file systems, the MDSs can be configured in a quasi active/active configuration, with each MDS managing metadata for a subset of the Lustre file system.

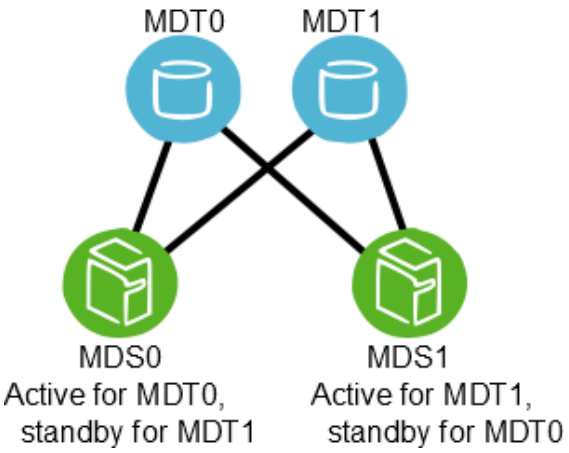
Figure 3.1. Lustre failover configuration for a active/passive MDT



Introduced in Lustre 2.4

Multiple MDTs became available with the advent of Lustre software release 2.4. MDTs can be setup as an active/active failover configuration. A failover cluster is built from two MDSs as shown in Figure 3.2, " Lustre failover configuration for a active/active MDTs ".

Figure 3.2. Lustre failover configuration for a active/active MDTs



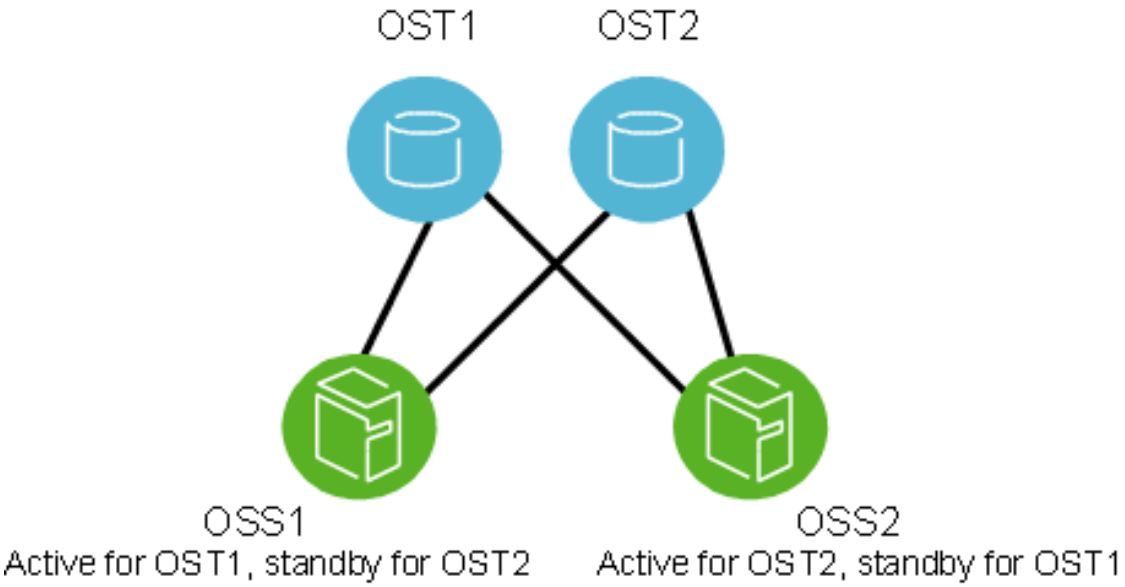
3.2.3. OST Failover Configuration (Active/Active)

OSTs are usually configured in a load-balanced, active/active failover configuration. A failover cluster is built from two OSSs as shown in Figure 3.3, " Lustre failover configuration for an OSTs ".

Note

OSSs configured as a failover pair must have shared disks/RAID.

Figure 3.3. Lustre failover configuration for an OSTs



In an active configuration, 50% of the available OSTs are assigned to one OSS and the remaining OSTs are assigned to the other OSS. Each OSS serves as the primary node for half the OSTs and as a failover node for the remaining OSTs.

In this mode, if one OSS fails, the other OSS takes over all of the failed OSTs. The clients attempt to connect to each OSS serving the OST, until one of them responds. Data on the OST is written synchronously, and the clients replay transactions that were in progress and uncommitted to disk before the OST failure.

For more information about configuring failover, see Chapter 11, *Configuring Failover in a Lustre File System*.

Part II. Installing and Configuring Lustre

Part II describes how to install and configure a Lustre file system. You will find information in this section about:

- Installation Overview
- Determining Hardware Configuration Requirements and Formatting Options
- Configuring Storage on a Lustre File System
- Setting Up Network Interface Bonding
- Installing the Lustre Software
- Configuring Lustre Networking (LNET)
- Configuring a Lustre File System
- Configuring Failover in a Lustre File System

Chapter 4. Installation Overview

This chapter provides on overview of the procedures required to set up, install and configure a Lustre file system.

Note

If the Lustre file system is new to you, you may find it helpful to refer to Part I, "Introducing the Lustre* File System" for a description of the Lustre architecture, file system components and terminology before proceeding with the installation procedure.

4.1. Steps to Installing the Lustre Software

To set up Lustre file system hardware and install and configure the Lustre software, refer the the chapters below in the order listed:

1. *(Required)* **Set up your Lustre file system hardware.**

   See Chapter 5, *Determining Hardware Configuration Requirements and Formatting Options* - Provides guidelines for configuring hardware for a Lustre file system including storage, memory, and networking requirements.

2. *(Optional - Highly Recommended)* **Configure storage on Lustre storage devices.**

   See Chapter 6, *Configuring Storage on a Lustre File System* - Provides instructions for setting up hardware RAID on Lustre storage devices.

3. *(Optional)* **Set up network interface bonding.**

   See Chapter 7, *Setting Up Network Interface Bonding* - Describes setting up network interface bonding to allow multiple network interfaces to be used in parallel to increase bandwidth or redundancy.

4. *(Required)* **Install Lustre software.**

   See Chapter 8, *Installing the Lustre Software* - Describes preparation steps and a procedure for installing the Lustre software.

5. *(Optional)* **Configure Lustre networking (LNET).**

   See Chapter 9, *Configuring Lustre Networking (LNET)* - Describes how to configure LNET if the default configuration is not sufficient. By default, LNET will use the first TCP/IP interface it discovers on a system. LNET configuration is required if you are using InfiniBand or multiple Ethernet interfaces.

6. *(Required)* **Configure the Lustre file system.**

   See Chapter 10, *Configuring a Lustre File System* - Provides an example of a simple Lustre configuration procedure and points to tools for completing more complex configurations.

7. *(Optional)* **Configure Lustre failover.**

   See Chapter 11, *Configuring Failover in a Lustre File System* - Describes how to configure Lustre failover.

Chapter 5. Determining Hardware Configuration Requirements and Formatting Options

This chapter describes hardware configuration requirements for a Lustre file system including:

- Section 5.1, “ Hardware Considerations”
- Section 5.2, “ Determining Space Requirements”
- Section 5.3, “ Setting File System Formatting Options”
- Section 5.4, “Determining Memory Requirements”
- Section 5.5, “Implementing Networks To Be Used by the Lustre File System”

5.1. Hardware Considerations

A Lustre file system can utilize any kind of block storage device such as single disks, software RAID, hardware RAID, or a logical volume manager. In contrast to some networked file systems, the block devices are only attached to the MDS and OSS nodes in a Lustre file system and are not accessed by the clients directly.

Since the block devices are accessed by only one or two server nodes, a storage area network (SAN) that is accessible from all the servers is not required. Expensive switches are not needed because point-to-point connections between the servers and the storage arrays normally provide the simplest and best attachments. (If failover capability is desired, the storage must be attached to multiple servers.)

For a production environment, it is preferable that the MGS have separate storage to allow future expansion to multiple file systems. However, it is possible to run the MDS and MGS on the same machine and have them share the same storage device.

For best performance in a production environment, dedicated clients are required. For a non-production Lustre environment or for testing, a Lustre client and server can run on the same machine. However, dedicated clients are the only supported configuration.

Performance and other issues can occur when an MDS or OSS and a client are running on the same machine:

- Running the MDS and a client on the same machine can cause recovery and deadlock issues and impact the performance of other Lustre clients.
- Running the OSS and a client on the same machine can cause issues with low memory and memory pressure. If the client consumes all the memory and then tries to write data to the file system, the OSS will need to allocate pages to receive data from the client but will not be able to perform this operation due to low memory. This can cause the client to hang.

Only servers running on 64-bit CPUs are tested and supported. 64-bit CPU clients are typically used for testing to match expected customer usage and avoid limitations due to the 4 GB limit for RAM size, 1 GB low-memory limitation, and 16 TB file size limit of 32-bit CPUs. Also, due to kernel API limitations,

performing backups of Lustre software release 2.x. file systems on 32-bit clients may cause backup tools to confuse files that have the same 32-bit inode number.

The storage attached to the servers typically uses RAID to provide fault tolerance and can optionally be organized with logical volume management (LVM), which is then formatted as a Lustre file system. Lustre OSS and MDS servers read, write and modify data in the format imposed by the file system.

The Lustre file system uses journaling file system technology on both the MDTs and OSTs. For a MDT, as much as a 20 percent performance gain can be obtained by placing the journal on a separate device.

The MDS can effectively utilize a lot of CPU cycles. A minimum of four processor cores are recommended. More are advisable for files systems with many clients.

Note

Lustre clients running on architectures with different endianness are supported. One limitation is that the PAGE_SIZE kernel macro on the client must be as large as the PAGE_SIZE of the server. In particular, ia64 or PPC clients with large pages (up to 64kB pages) can run with x86 servers (4kB pages). If you are running x86 clients with ia64 or PPC servers, you must compile the ia64 kernel with a 4kB PAGE_SIZE (so the server page size is not larger than the client page size).

5.1.1. MGT and MDT Storage Hardware Considerations

MGT storage requirements are small (less than 100 MB even in the largest Lustre file systems), and the data on an MGT is only accessed on a server/client mount, so disk performance is not a consideration. However, this data is vital for file system access, so the MGT should be reliable storage, preferably mirrored RAID1.

MDS storage is accessed in a database-like access pattern with many seeks and read-and-writes of small amounts of data. High throughput to MDS storage is not important. Storage types that provide much lower seek times, such as high-RPM SAS or SSD drives can be used for the MDT.

For maximum performance, the MDT should be configured as RAID1 with an internal journal and two disks from different controllers.

If you need a larger MDT, create multiple RAID1 devices from pairs of disks, and then make a RAID0 array of the RAID1 devices. This ensures maximum reliability because multiple disk failures only have a small chance of hitting both disks in the same RAID1 device.

Doing the opposite (RAID1 of a pair of RAID0 devices) has a 50% chance that even two disk failures can cause the loss of the whole MDT device. The first failure disables an entire half of the mirror and the second failure has a 50% chance of disabling the remaining mirror.

Introduced in Lustre 2.4

If multiple MDTs are going to be present in the system, each MDT should be specified for the anticipated usage and load.

Introduced in Lustre 2.4

MDT0 contains the root of the Lustre file system. If MDT0 is unavailable for any reason, the file system cannot be used.

Introduced in Lustre 2.4

Additional MDTs can be dedicated to sub-directories off the root file system provided by MDT0. Subsequent directories may also be configured to have their own MDT. If an MDT serving a subdirectory

becomes unavailable this subdirectory and all directories beneath it will also become unavailable. Configuring multiple levels of MDTs is an experimental feature for the Lustre software release 2.4.

5.1.2. OST Storage Hardware Considerations

The data access pattern for the OSS storage is a streaming I/O pattern that is dependent on the access patterns of applications being used. Each OSS can manage multiple object storage targets (OSTs), one for each volume with I/O traffic load-balanced between servers and targets. An OSS should be configured to have a balance between the network bandwidth and the attached storage bandwidth to prevent bottlenecks in the I/O path. Depending on the server hardware, an OSS typically serves between 2 and 8 targets, with each target up to 128 terabytes (TBs) in size.

Lustre file system capacity is the sum of the capacities provided by the targets. For example, 64 OSSs, each with two 8 TB targets, provide a file system with a capacity of nearly 1 PB. If each OST uses ten 1 TB SATA disks (8 data disks plus 2 parity disks in a RAID 6 configuration), it may be possible to get 50 MB/sec from each drive, providing up to 400 MB/sec of disk bandwidth per OST. If this system is used as storage backend with a system network, such as the InfiniBand network, that provides a similar bandwidth, then each OSS could provide 800 MB/sec of end-to-end I/O throughput. (Although the architectural constraints described here are simple, in practice it takes careful hardware selection, benchmarking and integration to obtain such results.)

5.2. Determining Space Requirements

The desired performance characteristics of the backing file systems on the MDT and OSTs are independent of one another. The size of the MDT backing file system depends on the number of inodes needed in the total Lustre file system, while the aggregate OST space depends on the total amount of data stored on the file system. If MGS data is to be stored on the MDT device (co-located MGT and MDT), add 100 MB to the required size estimate for the MDT.

Each time a file is created on a Lustre file system, it consumes one inode on the MDT and one inode for each OST object over which the file is striped. Normally, each file's stripe count is based on the system-wide default stripe count. However, this can be changed for individual files using the `lfs setstripe` option. For more details, see Chapter 18, *Managing File Layout (Striping) and Free Space*.

In a Lustre ldiskfs file system, all the inodes are allocated on the MDT and OSTs when the file system is first formatted. The total number of inodes on a formatted MDT or OST cannot be easily changed, although it is possible to add OSTs with additional space and corresponding inodes. Thus, the number of inodes created at format time should be generous enough to anticipate future expansion.

When the file system is in use and a file is created, the metadata associated with that file is stored in one of the pre-allocated inodes and does not consume any of the free space used to store file data.

Note

By default, the ldiskfs file system used by Lustre servers to store user-data objects and system data reserves 5% of space that cannot be used by the Lustre file system. Additionally, a Lustre file system reserves up to 400 MB on each OST for journal use and a small amount of space outside the journal to store accounting data. This reserved space is unusable for general storage. Thus, at least 400 MB of space is used on each OST before any file object data is saved.

5.2.1. Determining MGT Space Requirements

Less than 100 MB of space is required for the MGT. The size is determined by the number of servers in the Lustre file system cluster(s) that are managed by the MGS.

5.2.2. Determining MDT Space Requirements

When calculating the MDT size, the important factor to consider is the number of files to be stored in the file system. This determines the number of inodes needed, which drives the MDT sizing. To be on the safe side, plan for 2 KB per inode on the MDT, which is the default value. Attached storage required for Lustre file system metadata is typically 1-2 percent of the file system capacity depending upon file size.

For example, if the average file size is 5 MB and you have 100 TB of usable OST space, then you can calculate the minimum number of inodes as follows:

(100 TB * 1024 GB/TB * 1024 MB/GB) / 5 MB/inode = 20 million inodes

We recommend that you use at least twice the minimum number of inodes to allow for future expansion and allow for an average file size smaller than expected. Thus, the required space is:

2 KB/inode * 40 million inodes = 80 GB

If the average file size is small, 4 KB for example, the Lustre file system is not very efficient as the MDT uses as much space as the OSTs. However, this is not a common configuration for a Lustre environment.

Note

If the MDT is too small, this can cause all the space on the OSTs to be unusable. Be sure to determine the appropriate size of the MDT needed to support the file system before formatting the file system. It is difficult to increase the number of inodes after the file system is formatted.

5.2.3. Determining OST Space Requirements

For the OST, the amount of space taken by each object depends on the usage pattern of the users/ applications running on the system. The Lustre software defaults to a conservative estimate for the object size (16 KB per object). If you are confident that the average file size for your applications will be larger than this, you can specify a larger average file size (fewer total inodes) to reduce file system overhead and minimize file system check time. See Section 5.3.2, “Setting Formatting Options for an OST” for more details.

5.3. Setting File System Formatting Options

By default, the `mkfs.lustre` utility applies these options to the Lustre backing file system used to store data and metadata in order to enhance Lustre file system performance and scalability. These options include:

- `flex_bg` - When the flag is set to enable this flexible-block-groups feature, block and inode bitmaps for multiple groups are aggregated to minimize seeking when bitmaps are read or written and to reduce read/modify/write operations on typical RAID storage (with 1 MB RAID stripe widths). This flag is enabled on both OST and MDT file systems. On MDT file systems the `flex_bg` factor is left at the default value of 16. On OSTs, the `flex_bg` factor is set to 256 to allow all of the block or inode bitmaps in a single `flex_bg` to be read or written in a single I/O on typical RAID storage.

- `huge_file` - Setting this flag allows files on OSTs to be larger than 2 TB in size.

- `lazy_journal_init` - This extended option is enabled to prevent a full overwrite of the 400 MB journal that is allocated by default in a Lustre file system, which reduces the file system format time.

To override the default formatting options, use arguments to `mkfs.lustre` to pass formatting options to the backing file system:

```
--mkfsoptions='backing fs options'
```

For other `mkfs.lustre` options, see the Linux man page for `mke2fs(8)`.

5.3.1. Setting Formatting Options for an MDT

The number of inodes on the MDT is determined at format time based on the total size of the file system to be created. The default *bytes-per-inode* ratio ("inode ratio") for an MDT is optimized at one inode for every 2048 bytes of file system space. It is recommended that this value not be changed for MDTs.

This setting takes into account the space needed for additional metadata, such as the journal (up to 400 MB), bitmaps and directories, and a few files that the Lustre file system uses to maintain cluster consistency.

5.3.2. Setting Formatting Options for an OST

When formatting OST file systems, it is normally advantageous to take local file system usage into account. When doing so, try to minimize the number of inodes on each OST, while keeping enough margin for potential variations in future usage. This helps reduce the format and file system check time and makes more space available for data.

The table below shows the default *bytes-per-inode* ratio ("inode ratio") used for OSTs of various sizes when they are formatted.

Table 5.1. Inode Ratios Used for Newly Formatted OSTs

LUN/OST size	Inode ratio	Total inodes
over 10GB	1 inode/16KB	640 - 655k
10GB - 1TB	1 inode/68kiB	153k - 15.7M
1TB - 8TB	1 inode/256kB	4.2M - 33.6M
over 8TB	1 inode/1MB	8.4M - 134M

In environments with few small files, the default inode ratio may result in far too many inodes for the average file size. In this case, performance can be improved by increasing the number of *bytes-per-inode*.To set the inode ratio, use the `-i` argument to `mkfs.lustre` to specify the *bytes-per-inode* value.

Note

File system check time on OSTs is affected by a number of variables in addition to the number of inodes, including the size of the file system, the number of allocated blocks, the distribution of allocated blocks on the disk, disk speed, CPU speed, and the amount of RAM on the server. Reasonable file system check times are 5-30 minutes per TB.

For more details about formatting MDT and OST file systems, see Section 6.4, “Formatting Options for RAID Devices”.

5.3.3. File and File System Limits

Table 5.2, “File and file system limits” describes file and file system size limits. These limits are imposed by either the Lustre architecture or the Linux virtual file system (VFS) and virtual memory subsystems. In a few cases, a limit is defined within the code and can be changed by re-compiling the Lustre software

(see Chapter 30, *Installing a Lustre File System from Source Code*). In these cases, the indicated limit was used for testing of the Lustre software.

Table 5.2. File and file system limits

Limit	Value	Description
Maximum number of MDTs	1 Introduced in Lustre 2.4 4096	The Lustre software release 2.3 and earlier allows a maximum of 1 MDT per file system, but a single MDS can host multiple MDTs, each one for a separate file system. Introduced in Lustre 2.4 The Lustre software release 2.4 and later requires one MDT for the root. Upto 4095 additional MDTs can be added to the file system and attached into the namespace with remote directories.
Maximum number of OSTs	8150	The maximum number of OSTs is a constant that can be changed at compile time. Lustre file systems with up to 4000 OSTs have been tested.
Maximum OST size	128TB	This is not a *hard* limit. Larger OSTs are possible but today typical production systems do not go beyond 128TB per OST.
Maximum number of clients	131072	The maximum number of clients is a constant that can be changed at compile time.
Maximum size of a file system	512 PB	Each OST or MDT on 64-bit kernel servers can have a file system up to 128 TB. On 32-bit systems, due to page cache limits, 16TB is the maximum block device size, which in turn applies to the size of OST on 32-bit kernel servers. You can have multiple OST file systems on a single OSS node.
Maximum stripe count	2000	This limit is imposed by the size of the layout that needs to be stored on disk and sent in RPC requests, but is not a hard limit of the protocol.
Maximum stripe size	< 4 GB	The amount of data written to each object before moving on to next object.
Minimum stripe size	64 KB	Due to the 64 KB PAGE_SIZE on some 64-bit machines, the minimum stripe size is set to 64 KB.
Maximum object size	16 TB	The amount of data that can be stored in a single object. An object corresponds to a stripe. The ldiskfs limit of 16 TB for a single object applies. Files can consist of up to 2000 stripes, each 16 TB in size.
Maximum file size	16 TB on 32-bit systems	Individual files have a hard limit of nearly 16 TB on 32-bit systems imposed by the kernel

Limit	Value	Description
	31.25 PB on 64-bit systems	memory subsystem. On 64-bit systems this limit does not exist. Hence, files can be 64-bits in size. An additional size limit of up to the number of stripes is imposed, where each stripe is 16 TB. A single file can have a maximum of 2000 stripes, which gives an upper single file limit of 31.25 PB for 64-bit systems. The actual amount of data that can be stored in a file depends upon the amount of free space in each OST on which the file is striped.
Maximum number of files or subdirectories in a single directory	10 million files	The Lustre software uses the ldiskfs hashed directory code, which has a limit of about 10 million files depending on the length of the file name. The limit on subdirectories is the same as the limit on regular files. Lustre file systems are tested with ten million files in a single directory.
Maximum number of files in the file system	4 billion Introduced in Lustre 2.4 4096 * 4 billion	The ldiskfs file system imposes an upper limit of 4 billion inodes. By default, the MDS file system is formatted with 2KB of space per inode, meaning 1 billion inodes per file system of 2 TB. This can be increased initially, at the time of MDS file system creation. For more information, see Chapter 5, *Determining Hardware Configuration Requirements and Formatting Options*. Introduced in Lustre 2.4 Each additional MDT can hold up to 4 billion additional files, depending on available inodes and the distribution directories and files in the file system.
Maximum length of a filename	255 bytes (filename)	This limit is 255 bytes for a single filename, the same as in an ldiskfs file system.
Maximum length of a pathname	4096 bytes (pathname)	The Linux VFS imposes a full pathname length of 4096 bytes.
Maximum number of open files for a Lustre file system	None	The Lustre software does not impose a maximum for the number of open files, but the practical limit depends on the amount of RAM on the MDS. No "tables" for open files exist on the MDS, as they are only linked in a list to a given client's export. Each client process probably has a limit of several thousands of open files which depends on the ulimit.

Note

Introduced in Lustre 2.2

In Lustre software releases prior to release 2.2, the maximum stripe count for a single file was limited to 160 OSTs. In Lustre software release 2.2, the large `xattr` feature ("wide striping") was added to support up to 2000 OSTs. This feature is disabled by default at `mkfs.lustre` time. In order to enable this feature, set the "`-O large_xattr`" or "`-O ea_inode`" option on the MDT either by using `--mkfsoptions` at format time or by using `tune2fs`. Using either "`large_xattr`" or "`ea_inode`" results in "`ea_inode`" in the file system feature list.

5.4. Determining Memory Requirements

This section describes the memory requirements for each Lustre file system component.

5.4.1. Client Memory Requirements

A minimum of 2 GB RAM is recommended for clients.

5.4.2. MDS Memory Requirements

MDS memory requirements are determined by the following factors:

- Number of clients
- Size of the directories
- Load placed on server

The amount of memory used by the MDS is a function of how many clients are on the system, and how many files they are using in their working set. This is driven, primarily, by the number of locks a client can hold at one time. The number of locks held by clients varies by load and memory availability on the server. Interactive clients can hold in excess of 10,000 locks at times. On the MDS, memory usage is approximately 2 KB per file, including the Lustre distributed lock manager (DLM) lock and kernel data structures for the files currently in use. Having file data in cache can improve metadata performance by a factor of 10x or more compared to reading it from disk.

MDS memory requirements include:

- **File system metadata** : A reasonable amount of RAM needs to be available for file system metadata. While no hard limit can be placed on the amount of file system metadata, if more RAM is available, then the disk I/O is needed less often to retrieve the metadata.
- **Network transport** : If you are using TCP or other network transport that uses system memory for send/receive buffers, this memory requirement must also be taken into consideration.
- **Journal size** : By default, the journal size is 400 MB for each Lustre ldiskfs file system. This can pin up to an equal amount of RAM on the MDS node per file system.
- **Failover configuration** : If the MDS node will be used for failover from another node, then the RAM for each journal should be doubled, so the backup server can handle the additional load if the primary server fails.

5.4.2.1. Calculating MDS Memory Requirements

By default, 400 MB are used for the file system journal. Additional RAM is used for caching file data for the larger working set, which is not actively in use by clients but should be kept "hot" for improved access times. Approximately 1.5 KB per file is needed to keep a file in cache without a lock.

For example, for a single MDT on an MDS with 1,000 clients, 16 interactive nodes, and a 2 million file working set (of which 400,000 files are cached on the clients):

Operating system overhead = 512 MB

File system journal = 400 MB

1000 * 4-core clients * 100 files/core * 2kB = 800 MB

16 interactive clients * 10,000 files * 2kB = 320 MB

1,600,000 file extra working set * 1.5kB/file = 2400 MB

Thus, the minimum requirement for a system with this configuration is at least 4 GB of RAM. However, additional memory may significantly improve performance.

For directories containing 1 million or more files, more memory may provide a significant benefit. For example, in an environment where clients randomly access one of 10 million files, having extra memory for the cache significantly improves performance.

5.4.3. OSS Memory Requirements

When planning the hardware for an OSS node, consider the memory usage of several components in the Lustre file system (i.e., journal, service threads, file system metadata, etc.). Also, consider the effect of the OSS read cache feature, which consumes memory as it caches data on the OSS node.

In addition to the MDS memory requirements mentioned in Section 5.2.2, “ Determining MDT Space Requirements”, the OSS requirements include:

- **Service threads** : The service threads on the OSS node pre-allocate a 4 MB I/O buffer for each ost_io service thread, so these buffers do not need to be allocated and freed for each I/O request.
- **OSS read cache** : OSS read cache provides read-only caching of data on an OSS, using the regular Linux page cache to store the data. Just like caching from a regular file system in the Linux operating system, OSS read cache uses as much physical memory as is available.

The same calculation applies to files accessed from the OSS as for the MDS, but the load is distributed over many more OSSs nodes, so the amount of memory required for locks, inode cache, etc. listed under MDS is spread out over the OSS nodes.

Because of these memory requirements, the following calculations should be taken as determining the absolute minimum RAM required in an OSS node.

5.4.3.1. Calculating OSS Memory Requirements

The minimum recommended RAM size for an OSS with two OSTs is computed below:

Ethernet/TCP send/receive buffers (4 MB * 512 threads) = 2048 MB

400 MB journal size * 2 OST devices = 800 MB

1.5 MB read/write per OST IO thread * 512 threads = 768 MB

600 MB file system read cache * 2 OSTs = 1200 MB

1000 * 4-core clients * 100 files/core * 2kB = 800MB

16 interactive clients * 10,000 files * 2kB = 320MB

1,600,000 file extra working set * 1.5kB/file = 2400MB

DLM locks + file system metadata TOTAL = 3520MB

Per OSS DLM locks + file system metadata = 3520MB/6 OSS = 600MB (approx.)

Per OSS RAM minimum requirement = 4096MB (approx.)

This consumes about 1,400 MB just for the pre-allocated buffers, and an additional 2 GB for minimal file system and kernel usage. Therefore, for a non-failover configuration, the minimum RAM would be 4 GB for an OSS node with two OSTs. Adding additional memory on the OSS will improve the performance of reading smaller, frequently-accessed files.

For a failover configuration, the minimum RAM would be at least 6 GB. For 4 OSTs on each OSS in a failover configuration 10GB of RAM is reasonable. When the OSS is not handling any failed-over OSTs the extra RAM will be used as a read cache.

As a reasonable rule of thumb, about 2 GB of base memory plus 1 GB per OST can be used. In failover configurations, about 2 GB per OST is needed.

5.5. Implementing Networks To Be Used by the Lustre File System

As a high performance file system, the Lustre file system places heavy loads on networks. Thus, a network interface in each Lustre server and client is commonly dedicated to Lustre file system traffic. This is often a dedicated TCP/IP subnet, although other network hardware can also be used.

A typical Lustre file system implementation may include the following:

- A high-performance backend network for the Lustre servers, typically an InfiniBand (IB) network.
- A larger client network.
- Lustre routers to connect the two networks.

Lustre networks and routing are configured and managed by specifying parameters to the Lustre networking (`lnet`) module in `/etc/modprobe.d/lustre.conf`.

To prepare to configure Lustre networking, complete the following steps:

1. **Identify all machines that will be running Lustre software and the network interfaces they will use to run Lustre file system traffic. These machines will form the Lustre network .**

   A network is a group of nodes that communicate directly with one another. The Lustre software includes Lustre network drivers (LNDs) to support a variety of network types and hardware (see Chapter 2, *Understanding Lustre Networking (LNET)* for a complete list). The standard rules for specifying networks applies to Lustre networks. For example, two TCP networks on two different subnets (`tcp0` and `tcp1`) are considered to be two different Lustre networks.

2. **If routing is needed, identify the nodes to be used to route traffic between networks.**

   If you are using multiple network types, then you will need a router. Any node with appropriate interfaces can route Lustre networking (LNET) traffic between different network hardware types or topologies --the node may be a server, a client, or a standalone router. LNET can route messages between different network types (such as TCP-to-InfiniBand) or across different topologies (such as bridging two InfiniBand or TCP/IP networks). Routing will be configured in Chapter 9, *Configuring Lustre Networking (LNET)* .

3. **Identify the network interfaces to include in or exclude from LNET.**

   If not explicitly specified, LNET uses either the first available interface or a pre-defined default for a given network type. Interfaces that LNET should not use (such as an administrative network or IP-over-IB), can be excluded.

   Network interfaces to be used or excluded will be specified using the lnet kernel module parameters networks and `ip2netsas` described in Chapter 9, *Configuring Lustre Networking (LNET)* .

4. **To ease the setup of networks with complex network configurations, determine a cluster-wide module configuration.**

   For large clusters, you can configure the networking setup for all nodes by using a single, unified set of parameters in the `lustre.conf` file on each node. Cluster-wide configuration is described in Chapter 9, *Configuring Lustre Networking (LNET)* .

   ## Note

   We recommend that you use 'dotted-quad' notation for IP addresses rather than host names to make it easier to read debug logs and debug configurations with multiple interfaces.

Chapter 6. Configuring Storage on a Lustre File System

This chapter describes best practices for storage selection and file system options to optimize performance on RAID, and includes the following sections:

- Section 6.1, " Selecting Storage for the MDT and OSTs"
- Section 6.2, "Reliability Best Practices"
- Section 6.3, "Performance Tradeoffs"
- Section 6.4, "Formatting Options for RAID Devices"
- Section 6.5, "Connecting a SAN to a Lustre File System"

Note

It is strongly recommended that storage used in a Lustre file system be configured with hardware RAID. The Lustre software does not support redundancy at the file system level and RAID is required to protect against disk failure.

6.1. Selecting Storage for the MDT and OSTs

The Lustre architecture allows the use of any kind of block device as backend storage. The characteristics of such devices, particularly in the case of failures, vary significantly and have an impact on configuration choices.

This section describes issues and recommendations regarding backend storage.

6.1.1. Metadata Target (MDT)

I/O on the MDT is typically mostly reads and writes of small amounts of data. For this reason, we recommend that you use RAID 1 for MDT storage. If you require more capacity for an MDT than one disk provides, we recommend RAID 1 + 0 or RAID 10.

6.1.2. Object Storage Server (OST)

A quick calculation makes it clear that without further redundancy, RAID 6 is required for large clusters and RAID 5 is not acceptable:

> For a 2 PB file system (2,000 disks of 1 TB capacity) assume the mean time to failure (MTTF) of a disk is about 1,000 days. This means that the expected failure rate is 2000/1000 = 2 disks per day. Repair time at 10% of disk bandwidth is 1000 GB at 10MB/sec = 100,000 sec, or about 1 day.
>
> For a RAID 5 stripe that is 10 disks wide, during 1 day of rebuilding, the chance that a second disk in the same array will fail is about 9/1000 or about 1% per day. After 50 days, you have a 50% chance of a double failure in a RAID 5 array leading to data loss.
>
> Therefore, RAID 6 or another double parity algorithm is needed to provide sufficient redundancy for OST storage.

For better performance, we recommend that you create RAID sets with 4 or 8 data disks plus one or two parity disks. Using larger RAID sets will negatively impact performance compared to having multiple independent RAID sets.

To maximize performance for small I/O request sizes, storage configured as RAID 1+0 can yield much better results but will increase cost or reduce capacity.

6.2. Reliability Best Practices

RAID monitoring software is recommended to quickly detect faulty disks and allow them to be replaced to avoid double failures and data loss. Hot spare disks are recommended so that rebuilds happen without delays.

Backups of the metadata file systems are recommended. For details, see Chapter 17, *Backing Up and Restoring a File System*.

6.3. Performance Tradeoffs

A writeback cache can dramatically increase write performance on many types of RAID arrays if the writes are not done at full stripe width. Unfortunately, unless the RAID array has battery-backed cache (a feature only found in some higher-priced hardware RAID arrays), interrupting the power to the array may result in out-of-sequence writes or corruption of RAID parity and future data loss.

If writeback cache is enabled, a file system check is required after the array loses power. Data may also be lost because of this.

Therefore, we recommend against the use of writeback cache when data integrity is critical. You should carefully consider whether the benefits of using writeback cache outweigh the risks.

6.4. Formatting Options for RAID Devices

When formatting a file system on a RAID device, it is beneficial to ensure that I/O requests are aligned with the underlying RAID geometry. This ensures that Lustre RPCs do not generate unnecessary disk operations which may reduce performance dramatically. Use the `--mkfsoptions` parameter to specify additional parameters when formatting the OST or MDT.

For RAID 5, RAID 6, or RAID 1+0 storage, specifying the following option to the `--mkfsoptions` parameter option improves the layout of the file system metadata, ensuring that no single disk contains all of the allocation bitmaps:

`-E stride =` *`chunk_blocks`*

The *`chunk_blocks`* variable is in units of 4096-byte blocks and represents the amount of contiguous data written to a single disk before moving to the next disk. This is alternately referred to as the RAID stripe size. This is applicable to both MDT and OST file systems.

For more information on how to override the defaults while formatting MDT or OST file systems, see Section 5.3, “ Setting File System Formatting Options”.

6.4.1. Computing file system parameters for mkfs

For best results, use RAID 5 with 5 or 9 disks or RAID 6 with 6 or 10 disks, each on a different controller. The stripe width is the optimal minimum I/O size. Ideally, the RAID configuration should allow 1 MB

Lustre RPCs to fit evenly on a single RAID stripe without an expensive read-modify-write cycle. Use this formula to determine the *stripe_width*, where *number_of_data_disks* does *not* include the RAID parity disks (1 for RAID 5 and 2 for RAID 6):

stripe_width_blocks = *chunk_blocks* * *number_of_data_disks* = 1 MB

If the RAID configuration does not allow *chunk_blocks* to fit evenly into 1 MB, select *stripe_width_blocks*, such that is close to 1 MB, but not larger.

The *stripe_width_blocks* value must equal *chunk_blocks* * *number_of_data_disks*. Specifying the *stripe_width_blocks* parameter is only relevant for RAID 5 or RAID 6, and is not needed for RAID 1 plus 0.

Run `--reformat` on the file system device (`/dev/sdc`), specifying the RAID geometry to the underlying ldiskfs file system, where:

--mkfsoptions "*other_options* -E stride=*chunk_blocks*, stripe_width=*stripe_width_blo*

A RAID 6 configuration with 6 disks has 4 data and 2 parity disks. The *chunk_blocks* <= 1024KB/4 = 256KB.

Because the number of data disks is equal to the power of 2, the stripe width is equal to 1 MB.

--mkfsoptions "*other_options* -E stride=*chunk_blocks*, stripe_width=*stripe_width_blo*

6.4.2. Choosing Parameters for an External Journal

If you have configured a RAID array and use it directly as an OST, it contains both data and metadata. For better performance, we recommend putting the OST journal on a separate device, by creating a small RAID 1 array and using it as an external journal for the OST.

In a Lustre file system, the default journal size is 400 MB. A journal size of up to 1 GB has shown increased performance but diminishing returns are seen for larger journals. Additionally, a copy of the journal is kept in RAM. Therefore, make sure you have enough memory available to hold copies of all the journals.

The file system journal options are specified to `mkfs.lustre` using the `--mkfsoptions` parameter. For example:

--mkfsoptions "*other_options* -j -J device=/dev/mdJ"

To create an external journal, perform these steps for each OST on the OSS:

1. Create a 400 MB (or larger) journal partition (RAID 1 is recommended).

   In this example, `/dev/sdb` is a RAID 1 device.

2. Create a journal device on the partition. Run:

   oss# mke2fs -b 4096 -O journal_dev /dev/sdb *journal_size*

   The value of *journal_size* is specified in units of 4096-byte blocks. For example, 262144 for a 1 GB journal size.

3. Create the OST.

   In this example, `/dev/sdc` is the RAID 6 device to be used as the OST, run:

```
[oss#] mkfs.lustre --mgsnode=mds@osib --ost --index=0 \
--mkfsoptions="-J device=/dev/sdb1" /dev/sdc
```

4. Mount the OST as usual.

6.5. Connecting a SAN to a Lustre File System

Depending on your cluster size and workload, you may want to connect a SAN to a Lustre file system. Before making this connection, consider the following:

- In many SAN file systems, clients allocate and lock blocks or inodes individually as they are updated. The design of the Lustre file system avoids the high contention that some of these blocks and inodes may have.
- The Lustre file system is highly scalable and can have a very large number of clients. SAN switches do not scale to a large number of nodes, and the cost per port of a SAN is generally higher than other networking.
- File systems that allow direct-to-SAN access from the clients have a security risk because clients can potentially read any data on the SAN disks, and misbehaving clients can corrupt the file system for many reasons like improper file system, network, or other kernel software, bad cabling, bad memory, and so on. The risk increases with increase in the number of clients directly accessing the storage.

Chapter 7. Setting Up Network Interface Bonding

This chapter describes how to use multiple network interfaces in parallel to increase bandwidth and/or redundancy. Topics include:

- Section 7.1, “Network Interface Bonding Overview”
- Section 7.2, “Requirements”
- Section 7.3, “Bonding Module Parameters”
- Section 7.4, “Setting Up Bonding”
- Section 7.5, “Configuring a Lustre File System with Bonding”
- Section 7.6, “Bonding References”

Note

Using network interface bonding is optional.

7.1. Network Interface Bonding Overview

Bonding, also known as link aggregation, trunking and port trunking, is a method of aggregating multiple physical network links into a single logical link for increased bandwidth.

Several different types of bonding are available in the Linux distribution. All these types are referred to as 'modes', and use the bonding kernel module.

Modes 0 to 3 allow load balancing and fault tolerance by using multiple interfaces. Mode 4 aggregates a group of interfaces into a single virtual interface where all members of the group share the same speed and duplex settings. This mode is described under IEEE spec 802.3ad, and it is referred to as either 'mode 4' or '802.3ad.'

7.2. Requirements

The most basic requirement for successful bonding is that both endpoints of the connection must be capable of bonding. In a normal case, the non-server endpoint is a switch. (Two systems connected via crossover cables can also use bonding.) Any switch used must explicitly handle 802.3ad Dynamic Link Aggregation.

The kernel must also be configured with bonding. All supported Lustre kernels have bonding functionality. The network driver for the interfaces to be bonded must have the ethtool functionality to determine slave speed and duplex settings. All recent network drivers implement it.

To verify that your interface works with ethtool, run:

```
# which ethtool
/sbin/ethtool

# ethtool eth0
```

```
Settings for eth0:
          Supported ports: [ TP MII ]
          Supported link modes:   10baseT/Half 10baseT/Full
                                  100baseT/Half 100baseT/Full
          Supports auto-negotiation: Yes
          Advertised link modes:  10baseT/Half 10baseT/Full
                                  100baseT/Half 100baseT/Full
          Advertised auto-negotiation: Yes
          Speed: 100Mb/s
          Duplex: Full
          Port: MII
          PHYAD: 1
          Transceiver: internal
          Auto-negotiation: on
          Supports Wake-on: pumbg
          Wake-on: d
          Current message level: 0x00000001 (1)
          Link detected: yes

# ethtool eth1

Settings for eth1:
   Supported ports: [ TP MII ]
   Supported link modes:   10baseT/Half 10baseT/Full
                           100baseT/Half 100baseT/Full
   Supports auto-negotiation: Yes
   Advertised link modes:  10baseT/Half 10baseT/Full
   100baseT/Half 100baseT/Full
   Advertised auto-negotiation: Yes
   Speed: 100Mb/s
   Duplex: Full
   Port: MII
   PHYAD: 32
   Transceiver: internal
   Auto-negotiation: on
   Supports Wake-on: pumbg
   Wake-on: d
   Current message level: 0x00000007 (7)
   Link detected: yes
   To quickly check whether your kernel supports bonding, run:
   # grep ifenslave /sbin/ifup
   # which ifenslave
   /sbin/ifenslave
```

7.3. Bonding Module Parameters

Bonding module parameters control various aspects of bonding.

Outgoing traffic is mapped across the slave interfaces according to the transmit hash policy. We recommend that you set the `xmit_hash_policy` option to the layer3+4 option for bonding. This policy uses upper layer protocol information if available to generate the hash. This allows traffic to a particular network peer to span multiple slaves, although a single connection does not span multiple slaves.

```
$ xmit_hash_policy=layer3+4
```

The `miimon` option enables users to monitor the link status. (The parameter is a time interval in milliseconds.) It makes an interface failure transparent to avoid serious network degradation during link failures. A reasonable default setting is 100 milliseconds; run:

```
$ miimon=100
```

For a busy network, increase the timeout.

7.4. Setting Up Bonding

To set up bonding:

1. Create a virtual 'bond' interface by creating a configuration file:

   ```
   # vi /etc/sysconfig/network-scripts/ifcfg-bond0
   ```

2. Append the following lines to the file.

   ```
   DEVICE=bond0
   IPADDR=192.168.10.79 # Use the free IP Address of your network
   NETWORK=192.168.10.0
   NETMASK=255.255.255.0
   USERCTL=no
   BOOTPROTO=none
   ONBOOT=yes
   ```

3. Attach one or more slave interfaces to the bond interface. Modify the eth0 and eth1 configuration files (using a VI text editor).

   a. Use the VI text editor to open the eth0 configuration file.

      ```
      # vi /etc/sysconfig/network-scripts/ifcfg-eth0
      ```

   b. Modify/append the eth0 file as follows:

      ```
      DEVICE=eth0
      USERCTL=no
      ONBOOT=yes
      MASTER=bond0
      SLAVE=yes
      BOOTPROTO=none
      ```

   c. Use the VI text editor to open the eth1 configuration file.

      ```
      # vi /etc/sysconfig/network-scripts/ifcfg-eth1
      ```

   d. Modify/append the eth1 file as follows:

      ```
      DEVICE=eth1
      USERCTL=no
      ONBOOT=yes
      MASTER=bond0
      SLAVE=yes
      BOOTPROTO=none
      ```

4. Set up the bond interface and its options in `/etc/modprobe.d/bond.conf`. Start the slave interfaces by your normal network method.

```
# vi /etc/modprobe.d/bond.conf
```

a. Append the following lines to the file.

```
alias bond0 bonding
options bond0 mode=balance-alb miimon=100
```

b. Load the bonding module.

```
# modprobe bonding
# ifconfig bond0 up
# ifenslave bond0 eth0 eth1
```

5. Start/restart the slave interfaces (using your normal network method).

Note

You must `modprobe` the bonding module for each bonded interface. If you wish to create bond0 and bond1, two entries in `bond.conf` file are required.

The examples below are from systems running Red Hat Enterprise Linux. For setup use: `/etc/sysconfig/networking-scripts/ifcfg-*` The website referenced below includes detailed instructions for other configuration methods, instructions to use DHCP with bonding, and other setup details. We strongly recommend you use this website.

http://www.linuxfoundation.org/collaborate/workgroups/networking/bonding

6. Check /proc/net/bonding to determine status on bonding. There should be a file there for each bond interface.

```
# cat /proc/net/bonding/bond0
Ethernet Channel Bonding Driver: v3.0.3 (March 23, 2006)

Bonding Mode: load balancing (round-robin)
MII Status: up
MII Polling Interval (ms): 0
Up Delay (ms): 0
Down Delay (ms): 0

Slave Interface: eth0
MII Status: up
Link Failure Count: 0
Permanent HW addr: 4c:00:10:ac:61:e0

Slave Interface: eth1
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:14:2a:7c:40:1d
```

7. Use ethtool or ifconfig to check the interface state. ifconfig lists the first bonded interface as 'bond0.'

```
ifconfig
bond0      Link encap:Ethernet  HWaddr 4C:00:10:AC:61:E0
   inet addr:192.168.10.79  Bcast:192.168.10.255 \     Mask:255.255.255.0
   inet6 addr: fe80::4e00:10ff:feac:61e0/64 Scope:Link
```

```
      UP BROADCAST RUNNING MASTER MULTICAST  MTU:1500 Metric:1
      RX packets:3091 errors:0 dropped:0 overruns:0 frame:0
      TX packets:880 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:314203 (306.8 KiB)  TX bytes:129834 (126.7 KiB)

  eth0       Link encap:Ethernet  HWaddr 4C:00:10:AC:61:E0
      inet6 addr: fe80::4e00:10ff:feac:61e0/64 Scope:Link
      UP BROADCAST RUNNING SLAVE MULTICAST  MTU:1500 Metric:1
      RX packets:1581 errors:0 dropped:0 overruns:0 frame:0
      TX packets:448 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:162084 (158.2 KiB)  TX bytes:67245 (65.6 KiB)
      Interrupt:193 Base address:0x8c00

  eth1       Link encap:Ethernet  HWaddr 4C:00:10:AC:61:E0
      inet6 addr: fe80::4e00:10ff:feac:61e0/64 Scope:Link
      UP BROADCAST RUNNING SLAVE MULTICAST  MTU:1500 Metric:1
      RX packets:1513 errors:0 dropped:0 overruns:0 frame:0
      TX packets:444 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:152299 (148.7 KiB)  TX bytes:64517 (63.0 KiB)
      Interrupt:185 Base address:0x6000
```

7.4.1. Examples

This is an example showing `bond.conf` entries for bonding Ethernet interfaces `eth1` and `eth2` to `bond0`:

```
# cat /etc/modprobe.d/bond.conf
alias eth0 8139too
alias eth1 via-rhine
alias bond0 bonding
options bond0 mode=balance-alb miimon=100

# cat /etc/sysconfig/network-scripts/ifcfg-bond0
DEVICE=bond0
BOOTPROTO=none
NETMASK=255.255.255.0
IPADDR=192.168.10.79 # (Assign here the IP of the bonded interface.)
ONBOOT=yes
USERCTL=no

ifcfg-ethx
# cat /etc/sysconfig/network-scripts/ifcfg-eth0
TYPE=Ethernet
DEVICE=eth0
HWADDR=4c:00:10:ac:61:e0
BOOTPROTO=none
ONBOOT=yes
USERCTL=no
IPV6INIT=no
PEERDNS=yes
MASTER=bond0
```

```
SLAVE=yes
```

In the following example, the `bond0` interface is the master (MASTER) while `eth0` and `eth1` are slaves (SLAVE).

Note

All slaves of `bond0` have the same MAC address (Hwaddr) - `bond0`. All modes, except TLB and ALB, have this MAC address. TLB and ALB require a unique MAC address for each slave.

```
$ /sbin/ifconfig

bond0Link encap:EthernetHwaddr 00:C0:F0:1F:37:B4
inet addr:XXX.XXX.XXX.YYY Bcast:XXX.XXX.XXX.255 Mask:255.255.252.0
UP BROADCAST RUNNING MASTER MULTICAST MTU:1500  Metric:1
RX packets:7224794 errors:0 dropped:0 overruns:0 frame:0
TX packets:3286647 errors:1 dropped:0 overruns:1 carrier:0
collisions:0 txqueuelen:0

eth0Link encap:EthernetHwaddr 00:C0:F0:1F:37:B4
inet addr:XXX.XXX.XXX.YYY Bcast:XXX.XXX.XXX.255 Mask:255.255.252.0
UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500  Metric:1
RX packets:3573025 errors:0 dropped:0 overruns:0 frame:0
TX packets:1643167 errors:1 dropped:0 overruns:1 carrier:0
collisions:0 txqueuelen:100
Interrupt:10 Base address:0x1080

eth1Link encap:EthernetHwaddr 00:C0:F0:1F:37:B4
inet addr:XXX.XXX.XXX.YYY Bcast:XXX.XXX.XXX.255 Mask:255.255.252.0
UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500  Metric:1
RX packets:3651769 errors:0 dropped:0 overruns:0 frame:0
TX packets:1643480 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:100
Interrupt:9 Base address:0x1400
```

7.5. Configuring a Lustre File System with Bonding

The Lustre software uses the IP address of the bonded interfaces and requires no special configuration. The bonded interface is treated as a regular TCP/IP interface. If needed, specify `bond0` using the Lustre `networks` parameter in `/etc/modprobe`.

```
options lnet networks=tcp(bond0)
```

7.6. Bonding References

We recommend the following bonding references:

- In the Linux kernel source tree, see `documentation/networking/bonding.txt`
- http://linux-ip.net/html/ether-bonding.html.
- http://www.sourceforge.net/projects/bonding.

- Linux Foundation bonding website: http://www.linuxfoundation.org/collaborate/workgroups/networking/bonding. This is the most extensive reference and we highly recommend it. This website includes explanations of more complicated setups, including the use of DHCP with bonding.

Chapter 8. Installing the Lustre Software

This chapter describes how to install the Lustre software from RPM packages. It includes:

- Section 8.1, " Preparing to Install the Lustre Software"
- Section 8.2, "Lustre Software Installation Procedure"

For hardware and system requirements and hardware configuration information, see Chapter 5, *Determining Hardware Configuration Requirements and Formatting Options*.

8.1. Preparing to Install the Lustre Software

You can install the Lustre software from downloaded packages (RPMs) or directly from the source code. This chapter describes how to install the Lustre RPM packages. For information about installing from source code, see Chapter 30, *Installing a Lustre File System from Source Code*.

The Lustre RPM packages have been tested on the Linux distributions listed in the table below.

Table 8.1. Lustre Test Matrix

Lustre Release	Servers Tested[1]	Clients Tested
2.0	RHEL 5, CentOS 5	RHEL 5, CentOS 5, SLES 11 SP0
2.1.x	RHEL 5, CentOS 5, RHEL 6, CentOS 6	RHEL 5, CentOS 5, RHEL 6, CentOS 6, SLES 11 SP1
2.2	RHEL 6, CentOS 6	RHEL 5, CentOS 5, RHEL 6, CentOS 6, SLES 11 SP1
2.3	RHEL 6.3, CentOS 6.3	RHEL 6.3, CentOS 6.3, RHEL 5.8, CentOS 5.8, SLES 11 SP1
2.4.x	RHEL 6.4, CentOS 6.4	RHEL 6.4, CentOS 6.4, SLES 11 SP2, FC18

[1]Red Hat Enterprise Edition (RHEL), CentOS Enterprise Linux Distribution (CentOS), SUSE Linux Enterprise Server (SLES), Fedora* F18 Linux kernel (FC18).

8.1.1. Software Requirements

To install the Lustre software from RPMs, the following are required:

- **Lustre server packages**. The required packages for Lustre servers are listed in the table below, where *ver* refers to the Linux kernel distribution (e.g., 2.6.32-358.6.2.el6) and *arch* refers to the processor architecture (e.g., x86_64). These packages are available in the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository.

Table 8.2. Packages Installed on Lustre Servers

Package Name	Description
kernel-*ver*_lustre.*arch*	Linux kernel with Lustre software patches (often referred to as "patched kernel")

Package Name	Description
`lustre-`*`ver`*`_lustre.`*`arch`*	Lustre software command line tools
`lustre-modules-`*`ver`*`_lustre.`*`arch`*	Lustre-patched kernel modules
`lustre-ldiskfs-`*`ver`*`_lustre.`*`arch`*	Lustre back-end file system tools
`e2fsprogs`	Utility to maintain Lustre back-end file system
`lustre-tests-`*`ver`*`_lustre.`*`arch`*	Lustre I/O Kit benchmarking tools *(Included in Lustre software as of release 2.2)*

- **Lustre client packages**. The required packages for Lustre clients are listed in the table below, where *`ver`* refers to the Linux distribution (e.g., 2.6.18-348.1.1.el5). These packages are available in the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository.

Table 8.3. Packages Installed on Lustre Clients

Package Name	Description
`lustre-client-modules-`*`ver`*	Patchless kernel modules for client
`lustre-client-`*`ver`*`_lustre`	Client command line tools
`lustre-client-tests-`*`ver`*	Lustre I/O Kit *(Included in Lustre software as of release 2.2)*

Note

The version of the kernel running on a Lustre client must be the same as the version of the `lustre-client-modules-`*`ver`* package being installed. If the kernel running on the client is not compatible, a kernel that is compatible must be installed on the client before the Lustre file system software is installed.

- **Lustre LNET network driver (LND)**. The Lustre LNDs provided with the Lustre software are listed in the table below. For more information about Lustre LNET, see Chapter 2, *Understanding Lustre Networking (LNET)*.

Table 8.4. Network Types Supported by Lustre LNDs

Supported Network Types	Notes
TCP	Any network carrying TCP traffic, including GigE, 10GigE, and IPoIB
InfiniBand network	OpenFabrics OFED (o2ib)
gni	Gemini (Cray)
Seastar	Cray
MX	Myrinet network
ra	RapidArray* interconnect
Elan	Quadrics

Note

The InfiniBand and TCP Lustre LNDs are routinely tested during release cycles. The other LNDs are maintained by their respective owners

- **High availability software**. If needed, install third party high-availability software. For more information, see Section 11.2, “Preparing a Lustre File System for Failover”.
- **Optional packages.** Optional packages provided in the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository may include the following (depending on the operating system and platform):
  - `kernel-debuginfo`, `kernel-debuginfo-common`, `lustre-debuginfo`, `lustre-ldiskfs-debuginfo` - Versions of required packages with debugging symbols and other debugging options enabled for use in troubleshooting.
  - `kernel-devel`, - Portions of the kernel tree needed to compile third party modules, such as network drivers.
  - `kernel-firmware` - Standard Red Hat Enterprise Linux distribution that has been recompiled to work with the Lustre kernel.
  - `kernel-headers` - Header files installed under /user/include and used when compiling user-space, kernel-related code.
  - `lustre-source` - Lustre software source code.
  - *(Recommended)* `perf`, `perf-debuginfo`, `python-perf`, `python-perf-debuginfo` - Linux performance analysis tools that have been compiled to match the Lustre kernel version.

8.1.2. Environmental Requirements

Before installing the Lustre software, make sure the following environmental requirements are met.

- *(Required)* ***Disable Security-Enhanced Linux* (SELinux) on all Lustre servers and clients.*** The Lustre software does not support SELinux. Therefore, the SELinux system extension must be disabled on all Lustre nodes. Also, make sure other security extensions (such as the Novell AppArmor* security system) and network packet filtering tools (such as iptables) do not interfere with the Lustre software.
- *(Required)* ***Use the same user IDs (UID) and group IDs (GID) on all clients.*** If use of supplemental groups is required, see Section 34.1, “User/Group Upcall” for information about supplementary user and group cache upcall (`identity_upcall`).
- *(Recommended)* ***Provide remote shell access to clients.*** It is recommended that all cluster nodes have remote shell client access to facilitate the use of Lustre configuration and monitoring scripts. Parallel Distributed SHell (pdsh) is preferable, although Secure SHell (SSH) is acceptable.
- *(Recommended)* ***Ensure client clocks are synchronized.*** The Lustre file system uses client clocks for timestamps. If clocks are out of sync between clients, files will appear with different time stamps when accessed by different clients. Drifting clocks can also cause problems by, for example, making it difficult to debug multi-node issues or correlate logs, which depend on timestamps. We recommend that you use Network Time Protocol (NTP) to keep client and server clocks in sync with each other. For more information about NTP, see: http://www.ntp.org [http://www.ntp.org/].

8.2. Lustre Software Installation Procedure

Caution

Before installing the Lustre software, back up ALL data. The Lustre software contains kernel modifications that interact with storage devices and may introduce security issues and data loss if not installed, configured, or administered properly.

To install the Lustre software from RPMs, complete the steps below.

1. Verify that all Lustre installation requirements have been met.
   - For hardware requirements, see Chapter 5, *Determining Hardware Configuration Requirements and Formatting Options*.
   - For software and environmental requirements, see the section Section 8.1, " Preparing to Install the Lustre Software"above.
2. Download the `e2fsprogs` RPMs for your platform from the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository.
3. Download the Lustre server RPMs for your platform from the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository. See Table 8.2, "Packages Installed on Lustre Servers" for a list of required packages.
4. Install the Lustre server and `e2fsprogs` packages on all Lustre servers (MGS, MDSs, and OSSs).
   a. Log onto a Lustre server as the `root` user
   b. Use the `yum` command to install the packages:
      ```
      # yum --nogpgcheck install pkg1.rpm pkg2.rpm ...
      ```
   c. Verify the packages are installed correctly:
      ```
      rpm -qa|egrep "lustre|wc"|sort
      ```
   d. Reboot the server.
   e. Repeat these steps on each Lustre server.
5. Download the Lustre client RPMs for your platform from the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository. See Table 8.3, "Packages Installed on Lustre Clients" for a list of required packages.
6. Install the Lustre client packages on all Lustre clients.

   Note

   The version of the kernel running on a Lustre client must be the same as the version of the `lustre-client-modules-`*`ver`* package being installed. If not, a compatible kernel must be installed on the client before the Lustre client packages are installed.

   a. Log onto a Lustre client as the root user.
   b. Use the `yum` command to install the packages:
      ```
      # yum --nogpgcheck install pkg1.rpm pkg2.rpm ...
      ```
   c. Verify the packages were installed correctly:
      ```
      # rpm -qa|egrep "lustre|kernel"|sort
      ```
   d. Reboot the client.
   e. Repeat these steps on each Lustre client.

To configure LNET, go to Chapter 9, *Configuring Lustre Networking (LNET)* . If default settings will be used for LNET, go to Chapter 10, *Configuring a Lustre File System*.

Chapter 9. Configuring Lustre Networking (LNET)

This chapter describes how to configure Lustre networking (LNET). It includes the following sections:

- Section 9.1, " Overview of `lnet` Module Parameters"
- Section 9.2, "`networks` Parameter"
- Section 9.3, "`ip2nets` Parameter"
- Section 9.4, "Using the `routes` Parameter"
- Section 9.5, "Testing the LNET Configuration"
- Section 9.6, "Best Practices for LNET Options"

Note

LNET will, by default, use the first TCP/IP interface it discovers on a system. If the default is sufficient, further configuration is not required.

9.1. Overview of `lnet` Module Parameters

The LNET kernel module parameters specify how LNET is to be configured to work with a Lustre file system, including details on which NICs are to be configured and how routing among LNET routers will take place.

Parameters for LNET can be specified in the `/etc/modprobe.d/lustre.conf` file. The file need not be named `lustre.conf`. However, this convention will be used in this manual. For distributions previous to Red Hat Enterprise Edition 5 and SUSE Linux Enterprise Server 10, LNET parameters may have been stored in `/etc/modprobe.conf`, but this method of specifying LNET parameters is now deprecated. Storing LNET parameters in a separate `/etc/modprobe.d/lustre.conf` file simplifies administration and distribution of the Lustre networking configuration. This file contains one or more entries with the syntax:

```
options lnet parameter=value
```

To specify the network interfaces that are to be used for the Lustre file system, set either the `networks` parameter or the `ip2nets` parameter (only one of these parameters can be used at a time):

- `networks` - Specifies the LNET network interfaces to be used.
- `ip2nets` - Lists globally-available networks, each with a range of IP addresses. LNET then identifies locally-available networks through address list-matching lookup.

See Section 9.2, "`networks` Parameter" and Section 9.3, "`ip2nets` Parameter" for more details.

To set up routing between networks, use the `routes` parameter to identify the LNET subnets and the NID(s) of the routers that forward to them. See Section 9.4, "Using the `routes` Parameter" for more details.

9.2. networks Parameter

The networks parameter maps a network interface to an LNET subnet. You can do this by including an entry or a comma separated list in the lustre.conf file.

```
options lnet networks=LNET(interface),LNET(interface),...
```

For example:

To map the LNET tcp0 to the Ethernet interface eth1:

```
options lnet networks=tcp0(eth1)
```

To map the LNET o2ib to the InfiniBand interface ib0:

```
options lnet networks=o2ib(ib0)
```

Omitting a network number in an LNET specification results in a network number of 0.

Depending on the network design, it may be necessary to specify explicit interfaces. To explicitly specify that interface eth2 be used for network tcp0 and eth3 be used for tcp1 , use this entry:

```
options lnet networks=tcp0(eth2),tcp1(eth3)
```

When more than one interface is available during the network setup, the Lustre software chooses the best route based on the hopcount. Once the network connection is established, the Lustre software expects the network to stay connected. In a Lustre network, connections do not fail over to another interface, even if multiple interfaces are available on the same node.

Introduced in Lustre 2.5

Since Lustre software release 2.5, the route priority is also used to decide which interface to use. The priority has precendence over hopcount so the route with the lower priority number will be selected regardless of the hopcount.

Note

LNET lines in lustre.conf are only used by the local node to determine what to call its interfaces. They are not used for routing decisions.

9.2.1. Multi-homed Server Example

If a server with multiple IP addresses (multi-homed server) is connected to a Lustre network, certain configuration settings are required. An example illustrating these setting consists of a network with the following nodes:

- Server svr1 with three Ethernet NIC s (eth0, eth1, and eth2) and an InfiniBand NIC.
- Server svr2 with three Ethernet NICs (eth0, eth1, and eth2) and an InfiniBand NIC. The interface eth2 will not be used for Lustre networking.
- TCP clients, each with a single Ethernet interface.
- InfiniBand clients, each with a single InfiniBand interface and an Ethernet interface for administration.

To set the `networks` option for this example:

- On each server, `svr1` and `svr2`, include the following line in the `lustre.conf` file:

```
options lnet networks=tcp0(eth0),tcp1(eth1),o2ib(ib0)
```

- For TCP-only clients, the first available non-loopback IP interface is used for `tcp0`. Thus, TCP clients with only one interface do not need to have options defined in the `lustre.conf` file.

- On the InfiniBand clients, include the following line in the `lustre.conf` file:

```
options lnet networks=o2ib(ib0)
```

If a node has only one InfiniBand interface, then the interface name need not be specified. `- ib0` will be assumed.

9.3. `ip2nets` Parameter

The `ip2nets` option is typically used when a single, universal `lustre.conf` file is run on all servers and clients. Each node identifies the locally available networks based on the listed IP address patterns that match the node's local IP addresses.

Note that the IP address patterns listed in the `ip2nets` option are *only* used to identify the networks that an individual node should instantiate. They are *not* used by LNET for any other communications purpose.

For the example below, the nodes in the network have these IP addresses:

- Server svr1: `eth0` IP address `192.168.0.2`, IP over InfiniBand (`o2ib`) address `132.6.1.2`.

- Server svr2: `eth0` IP address `192.168.0.4`, IP over InfiniBand (`o2ib`) address `132.6.1.4`.

- TCP clients have IP addresses `192.168.0.5-255.`

- InfiniBand clients have IP over InfiniBand (`o2ib`) addresses `132.6.[2-3].2, .4, .6, .8`.

The following entry is placed in the `lustre.conf` file on each server and client:

```
options lnet 'ip2nets="tcp0(eth0) 192.168.0.[2,4]; \
tcp0 192.168.0.*; o2ib0 132.6.[1-3].[2-8/2]"'
```

Each entry in `ip2nets` is referred to as a 'rule'.

The order of LNET entries is important when configuring servers. If a server node can be reached using more than one network, the first network specified in `lustre.conf` will be used.

Because `svr1` and `svr2` match the first rule, LNET uses `eth0` for `tcp0` on those machines. (Although `svr1` and `svr2` also match the second rule, the first matching rule for a particular network is used).

The `[2-8/2]` format indicates a range of 2-8 stepped by 2; that is 2,4,6,8. Thus, the clients at `132.6.3.5` will not find a matching `o2ib` network.

9.4. Using the `routes` Parameter

Before discussing the `routes` parameter, it is important to look at what LNET routing achieves and why it is needed.

The role of an LNET router is essentially that of a gateway that forwards Lustre traffic from one LNET network to one (or more) other LNET networks. This implies that at the LNET layer, an LNET router is the next hop between two nodes on different LNET networks. Note that this is orthogonal to whether or not the nodes in question are in the same IP subnet or are separated by one or more IP routers. Therefore, LNET routing is not required when clients and servers are all in the same LNET network.

However, an LNET router is more than just a regular gateway; it is an LNET intelligent gateway. It is important to remember that Lustre routing is static routing, where the LNET network topology is statically configured and parsed when the system initializes. Routers can die and also come back online during operation. These topics will be discussed in greater detail in the following sections.

9.4.1. `routes` Parameter

The `routes` parameter is used to tell a node which route to use when forwarding traffic by identifying LNET routers in a Lustre configuration. The routes parameter needs to be set in the `lustre.conf` file to specify the next hop router.

The `routes` parameter specifies a semi-colon separated list of router definitions.

```
routes=dest_lnet [hop] [priority] router_NID@src_lnet; \
       dest_lnet [hop] [priority] router_NID@src_lnet
```

An alternative syntax consists of a colon separated list of router definitions:

```
routes=dest_lnet: [hop] [priority] router_NID@src_lnet \
                  [hop] [priority] router_NID@src_lnet
```

When there are two or more LNET routers, it is possible to give weighted priorities to each router using the `priority` parameter. Some reasons for doing this are that one of the routers is more capable than the other, or one is a primary router and the other is a back up, or one is for one section of clients and the other is for another section, or each router is moving traffic to a different physical location.The `priority` parameter is optional and need not be specified if no priority exists.

The `hop` parameter specifies the number of hops to the destination. When a node forwards traffic, the route with the least hops is used. If multiple routes to the same destination network have the same number of hops, the traffic is distributed between these routes in a round-robin fashion. To reach/transmit to the LNET `dest_lnet`, the next hop for a given node is the LNET router with the NID `router_NID` in the LNET `src_lnet`.

Given a sufficiently well-architected system, it is possible to map the flow to and from every single client or server. This type of routing has also been called *fine-grained* routing.

9.4.2. Router Configurations

To get a router setup started and running, execute the following commands:

```
modprobe lnet
lctl network configure
```

OR

```
lctl network up
```

For a router to forward traffic, the configuration parameter `forwarding` must be set to `enabled` (it is set to `disabled` by default). If forwarding is not enabled, traffic destined for the node is dropped.

Note

Any changes to the `lustre.conf` modprobe file require unloading the `lnet` kernel module and repeating the above given commands. Thus any major changes, especially on the servers, should be made with due consideration.

Here are a few examples starting from a basic setup and increasing in complexity to better illustrate LNET routing:

- *A simple setup involving one TCP client, one router and servers(MGS,MDS,OSS) in an InfiniBand fabric.*

  The LNET router has two NIDs, 192.168.1.2@tcp0 and 10.13.24.90@o2ib0.

  The `lustre.conf` file for the client includes:

  ```
  options lnet networks="tcp0(eth0)" routes="o2ib0 192.168.1.2@tcp0"
  ```

  On the router nodes:

  ```
  options lnet networks="o2ib0(ib0),tcp0(eth0)" forwarding=enabled
  ```

  On the server nodes:

  ```
  options lnet networks="o2ib0(ib0)" routes="tcp0 10.13.24.90@o2ib0"
  ```

- *Adding more routers to above described case.*

  When two or more routers are specified in the file, they are considered equivalent and are not configured as primary or secondary. The load is directed to the routers based on the `route` parameter settings for `priority` and `hop` (see Section 9.4, “Using the `routes` Parameter”). The syntax on the client nodes is:

  ```
  options lnet networks="tcp0(eth0)" routes="o2ib0 192.168.1.2@tcp0;
   o2ib0 192.168.1.3@tcp0"
  ```

  The servers can be configured likewise. The number of LNET routers is not limited. Enough routers should be used to handle the required file serving bandwidth plus a 25 percent margin for headroom. The nodes are capable of recovering if one or more routers fail, provided, there is still at least one router left running. The nodes are also capable of recognizing when an offline routers comes online.

- *Assigning priorities to routers.*

  ```
  options lnet networks="tcp0(eth0)" routes="o2ib0 1 192.168.1.2@tcp0;
   o2ib0 2 192.168.1.3@tcp0"
  ```

  The above example specifies two routers with priorities 1 and 2. In this case, the node will always first send traffic via 192.168.1.2 and if and only if that router is down, will it send traffic via 192.168.1.3.

  It is important to note that the weights are pre-configured and do not or cannot be changed dynamically. This method gives more control to an administrator on deciding which is a better route than just the route with least number of hops.

- *Assigning equal priorities to routers.*

  ```
  options lnet networks="tcp0(eth0)" routes="o2ib0 1 192.168.1.2@tcp0;
   o2ib0 1 192.168.1.3@tcp0"
  ```

The above example has two routers of the same priority. This is treated the same as specifying no priority at all.

- *Creating classes of routers.*

```
options lnet networks="tcp0(eth0)" routes="o2ib0 1 192.168.1.2@tcp0;
  o2ib0 1 192.168.1.3@tcp0;o2ib0 2 192.168.1.4@tcp0;
  o2ib0 2 192.168.1.5@tcp0"
```

This example creates two classes of routers load balanced between themselves. In this case, the node will only use the routers with priority 2 only if both the routers with priority 1 are not running.

9.4.3. Kernel Configuration Parameters

In a Lustre configuration in which different types of LNET networks are connected by routers, several kernel module parameters can be set to monitor and improve routing performance.

The routing related parameters are:

- `auto_down` - Enables/disables (1/0) the automatic marking of router state as up or down. The default value is 1. To disable router marking, enter:

  ```
  options lnet auto_down=0
  ```

- `avoid_asym_router_failure` - Specifies that if even one interface of a router is down for some reason, the entire router is marked as down. This is important because if nodes are not aware that the interface on one side is down, they will still keep pushing data to the other side presuming that the router is healthy, when it really is not. To turn it on, enter:

  ```
  options lnet avoid_asym_router_failure=1
  ```

- `live_router_check_interval` - Specifies a time interval in seconds after which the router checker will ping the live routers. The default value is 60. To set the value to 50, enter:

  ```
  options lnet live_router_check_interval=50
  ```

- `dead_router_check_interval` - Specifies a time interval in seconds after which the router checker will check the dead routers. The default value is 60. To set the value to 50, enter:

  ```
  options lnet dead_router_check_interval=50
  ```

  Note

  A kernel process called `router_checker` is started when either of the two options above are set (thus, the `router_checker` is enabled by default). It otians a list of next-hops from the routes entered and checks whether next-hops are alive. In a multi-hop router configuration, these parameters can be set on the routers as well, or on other node types. If no routes are entered, no checking is done.

- `router_ping_timeout` - Specifies a timeout for the router checker when it checks live or dead routers. The router checker sends a ping message to each dead or live router once every `dead_router_check_interval` or `live_router_check_interval` respectively. The default value is 50. To set the value to 60, enter:

  ```
  options lnet router_ping_timeout=60
  ```

Note

The `router_ping_timeout` is consistent with the default LND timeouts. You may have to increase it on very large clusters if the LND timeout is also increased. For larger clusters, we suggest increasing the check interval.

- `check_routers_before_use` - Specifies that routers are to be checked before use. Set to off by default. If this parameter is set to on, the `dead_router_check_interval` parameter must be given a positive integer value.

```
options lnet check_routers_before_use=on
```

The `router_checker` obtains the following information from each router:

- Time the router was disabled
- Elapsed disable time

If the `router_checker` does not get a reply message from the router within `router_ping_timeout` seconds, it considers the router to be down.

When a router in a priority class goes down, the traffic stops intermittently until LNET safely marks the router which is down as 'down' and then proceeds on again, depending either on other routers of the same or a different priority class. The time it takes for LNET to recover is roughly based on the values for the `live/dead_router_checker` parameters provided.

If a router that is marked 'up' responds to a ping, the timeout is reset. If 100 packets have been sent successfully through a router, the sent-packets counter for that router will have a value of 100. The ping response also provides the status of the NIDs of the node being pinged. In this way, the pinging node knows whether to keep using this node as a next-hop or not. If one of the NIDs of the router is down and the `avoid_asym_router_failure = 1` is set, then that router is no longer used.

9.5. Testing the LNET Configuration

After configuring Lustre networking, it is highly recommended that you test your LNET configuration using the LNET Self-Test provided with the Lustre software. For more information about using LNET Self-Test, see Chapter 24, *Testing Lustre Network Performance (LNET Self-Test)*.

9.6. Best Practices for LNET Options

For the `networks`, `ip2nets`, and `routes` options, follow these best practices to avoid configuration errors.

9.6.1. Escaping commas with quotes

It is recommended that you use network addresses rather than host names to make it easier to read debug logs and debug configurations with multiple interfaces.

If the server has multiple interfaces on the same subnet, the Linux kernel will send all traffic using the first configured interface. This is a limitation in the Linux distribution, not the Lustre software. In this case, network interface bonding should be used. For more information about network interface bonding, see Chapter 7, *Setting Up Network Interface Bonding*.

Depending on the Linux distribution, commas may need to be escaped using single or double quotes. In the extreme case, the `options` entry would look like this:

```
options lnet'networks="tcp0,elan0"' 'routes="tcp [2,10]@elan0"'
```

Added quotes may confuse some distributions. Messages such as the following may indicate an issue related to added quotes:

```
lnet: Unknown parameter 'networks'
```

A `'Refusing connection - no matching NID'` message generally points to an error in the LNET module configuration.

9.6.2. Including comments

Place the semicolon terminating a comment immediately after the comment. LNET silently ignores everything between the # character at the beginning of the comment and the next semicolon.

In this *incorrect* example, LNET silently ignores `pt11 192.168.0.[92,96]`, resulting in these nodes not being properly initialized. No error message is generated.

```
options lnet ip2nets="pt10 192.168.0.[89,93]; # comment with semicolon \
      BEFORE comment pt11 192.168.0.[92,96];
```

This *correct* example shows the required syntax:

```
options lnet ip2nets="pt10 192.168.0.[89,93] \
# comment with semicolon AFTER comment; \
pt11 192.168.0.[92,96] # comment
```

Do not add an excessive number of comments. The Linux kernel limits the length of character strings used in module options (usually to 1KB, but this may differ between vendor kernels). If you exceed this limit, errors result and the specified configuration may not be processed correctly.

Chapter 10. Configuring a Lustre File System

This chapter shows how to configure a simple Lustre file system comprised of a combined MGS/MDT, an OST and a client. It includes:

- Section 10.1, " Configuring a Simple Lustre File System"
- Section 10.2, " Additional Configuration Options"

10.1. Configuring a Simple Lustre File System

A Lustre file system can be set up in a variety of configurations by using the administrative utilities provided with the Lustre software. The procedure below shows how to configure a simple Lustre file system consisting of a combined MGS/MDS, one OSS with two OSTs, and a client. For an overview of the entire Lustre installation procedure, see Chapter 4, *Installation Overview*.

This configuration procedure assumes you have completed the following:

- ***Set up and configured your hardware*** . For more information about hardware requirements, see Chapter 5, *Determining Hardware Configuration Requirements and Formatting Options*.
- **Downloaded and installed the Lustre software.** For more information about preparing for and installing the Lustre software, see Chapter 8, *Installing the Lustre Software*.

The following optional steps should also be completed, if needed, before the Lustre software is configured:

- *Set up a hardware or software RAID on block devices to be used as OSTs or MDTs.* For information about setting up RAID, see the documentation for your RAID controller or Chapter 6, *Configuring Storage on a Lustre File System*.
- *Set up network interface bonding on Ethernet interfaces.* For information about setting up network interface bonding, see Chapter 7, *Setting Up Network Interface Bonding*.
- *Set* lnet *module parameters to specify how Lustre Networking (LNET) is to be configured to work with a Lustre file system and test the LNET configuration.* LNET will, by default, use the first TCP/IP interface it discovers on a system. If this network configuration is sufficient, you do not need to configure LNET. LNET configuration is required if you are using InfiniBand or multiple Ethernet interfaces.

For information about configuring LNET, see Chapter 9, *Configuring Lustre Networking (LNET)* . For information about testing LNET, see Chapter 24, *Testing Lustre Network Performance (LNET Self-Test)*.

- *Run the benchmark script* `sgpdd-survey` *to determine baseline performance of your hardware.* Benchmarking your hardware will simplify debugging performance issues that are unrelated to the Lustre software and ensure you are getting the best possible performance with your installation. For information about running `sgpdd-survey`, see Chapter 25, *Benchmarking Lustre File System Performance (Lustre I/O Kit)*.

  ### Note

  The `sgpdd-survey` script overwrites the device being tested so it must be run before the OSTs are configured.

To configure a simple Lustre file system, complete these steps:

1. Create a combined MGS/MDT file system on a block device. On the MDS node, run:

   ```
   mkfs.lustre --fsname=fsname --mgs --mdt --index=0 /dev/block_device
   ```

   The default file system name (`fsname`) is `lustre`.

   Note

   If you plan to create multiple file systems, the MGS should be created separately on its own dedicated block device, by running:

   ```
   mkfs.lustre --fsname=fsname --mgs /dev/block_device
   ```

   See Section 13.7, “Running Multiple Lustre File Systems” for more details.

2. Optional for Lustre software release 2.4 and later. Add in additional MDTs.

   ```
   mkfs.lustre --fsname=fsname --mgsnode=nid --mdt --index=1 /dev/block_device
   ```

   Note

   Up to 4095 additional MDTs can be added.

3. Mount the combined MGS/MDT file system on the block device. On the MDS node, run:

   ```
   mount -t lustre /dev/block_device /mount_point
   ```

   Note

   If you have created an MGS and an MDT on separate block devices, mount them both.

4. Create the OST. On the OSS node, run:

   ```
   mkfs.lustre --fsname=fsname --mgsnode=MGS_NID --ost --index=OST_index /dev/block
   ```

   When you create an OST, you are formatting a `ldiskfs` file system on a block storage device like you would with any local file system.

   You can have as many OSTs per OSS as the hardware or drivers allow. For more information about storage and memory requirements for a Lustre file system, see Chapter 5, *Determining Hardware Configuration Requirements and Formatting Options*.

   You can only configure one OST per block device. You should create an OST that uses the raw block device and does not use partitioning.

   You should specify the OST index number at format time in order to simplify translating the OST number in error messages or file striping to the OSS node and block device later on.

   If you are using block devices that are accessible from multiple OSS nodes, ensure that you mount the OSTs from only one OSS node at at time. It is strongly recommended that multiple-mount protection be enabled for such devices to prevent serious data corruption. For more information about multiple-mount protection, see Chapter 20, *Lustre File System Failover and Multiple-Mount Protection*.

Note

The Lustre software currently supports block devices up to 128 TB on Red Hat Enterprise Linux 5 and 6 (up to 8 TB on other distributions). If the device size is only slightly larger that 16 TB, it is recommended that you limit the file system size to 16 TB at format time. We recommend that you not place DOS partitions on top of RAID 5/6 block devices due to negative impacts on performance, but instead format the whole disk for the file system.

5. Mount the OST. On the OSS node where the OST was created, run:

   ```
   mount -t lustre /dev/block_device /mount_point
   ```

   ### Note

   To create additional OSTs, repeat Step 4 and Step 5, specifying the next higher OST index number.

6. Mount the Lustre file system on the client. On the client node, run:

   ```
   mount -t lustre MGS_node:/fsname /mount_point
   ```

   ### Note

   To create additional clients, repeat Step 6.

   ### Note

   If you have a problem mounting the file system, check the syslogs on the client and all the servers for errors and also check the network settings. A common issue with newly-installed systems is that `hosts.deny` or firewall rules may prevent connections on port 988.

7. Verify that the file system started and is working correctly. Do this by running `lfs df`, `dd` and `ls` commands on the client node.

8. *(Optional)* Run benchmarking tools to validate the performance of hardware and software layers in the cluster. Available tools include:

   - `obdfilter-survey` - Characterizes the storage performance of a Lustre file system. For details, see Section 25.3, “Testing OST Performance (`obdfilter-survey`)”.

   - `ost-survey` - Performs I/O against OSTs to detect anomalies between otherwise identical disk subsystems. For details, see Section 25.4, “Testing OST I/O Performance (`ost-survey`)”.

10.1.1. Simple Lustre Configuration Example

To see the steps to complete for a simple Lustre file system configuration, follow this example in which a combined MGS/MDT and two OSTs are created to form a file system called `temp`. Three block devices are used, one for the combined MGS/MDS node and one for each OSS node. Common parameters used in the example are listed below, along with individual node parameters.

Common Parameters	Value	Description
MGS node	`10.2.0.1@tcp0`	Node for the combined MGS/MDS

Common Parameters		Value	Description
	file system	`temp`	Name of the Lustre file system
	network type	`TCP/IP`	Network type used for Lustre file system `temp`

Node Parameters		Value	Description
MGS/MDS node			
	MGS/MDS node	`mdt0`	MDS in Lustre file system `temp`
	block device	`/dev/sdb`	Block device for the combined MGS/MDS node
	mount point	`/mnt/mdt`	Mount point for the `mdt0` block device (`/dev/sdb`) on the MGS/MDS node
First OSS node			
	OSS node	`oss0`	First OSS node in Lustre file system `temp`
	OST	`ost0`	First OST in Lustre file system `temp`
	block device	`/dev/sdc`	Block device for the first OSS node (`oss0`)
	mount point	`/mnt/ost0`	Mount point for the `ost0` block device (`/dev/sdc`) on the `oss1` node
Second OSS node			
	OSS node	`oss1`	Second OSS node in Lustre file system `temp`
	OST	`ost1`	Second OST in Lustre file system `temp`
	block device	`/dev/sdd`	Block device for the second OSS node (oss1)
	mount point	`/mnt/ost1`	Mount point for the `ost1` block device (`/dev/sdd`) on the `oss1` node
Client node			
	client node	`client1`	Client in Lustre file system `temp`
	mount point	`/lustre`	Mount point for Lustre file system `temp` on the `client1` node

Note

We recommend that you use 'dotted-quad' notation for IP addresses rather than host names to make it easier to read debug logs and debug configurations with multiple interfaces.

For this example, complete the steps below:

1. Create a combined MGS/MDT file system on the block device. On the MDS node, run:

```
[root@mds /]# mkfs.lustre --fsname=temp --mgs --mdt --index=0 /dev/sdb
```

This command generates this output:

```
    Permanent disk data:
Target:            temp-MDT0000
Index:             0
Lustre FS: temp
Mount type:        ldiskfs
Flags:             0x75
   (MDT MGS first_time update )
Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
Parameters: mdt.identity_upcall=/usr/sbin/l_getidentity

checking for existing Lustre data: not found
device size = 16MB
2 6 18
formatting backing filesystem ldiskfs on /dev/sdb
   target name             temp-MDTffff
   4k blocks               0
   options                 -i 4096 -I 512 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L temp-MDTffff  -i 4096 -I 512 -q -O
dir_index,uninit_groups -F /dev/sdb
Writing CONFIGS/mountdata
```

2. Mount the combined MGS/MDT file system on the block device. On the MDS node, run:

```
[root@mds /]# mount -t lustre /dev/sdb /mnt/mdt
```

This command generates this output:

```
Lustre: temp-MDT0000: new disk, initializing
Lustre: 3009:0:(lproc_mds.c:262:lprocfs_wr_identity_upcall()) temp-MDT0000:
group upcall set to /usr/sbin/l_getidentity
Lustre: temp-MDT0000.mdt: set parameter identity_upcall=/usr/sbin/l_getidentity
Lustre: Server temp-MDT0000 on device /dev/sdb has started
```

3. Create and mount `ost0`.

In this example, the OSTs (`ost0` and `ost1`) are being created on different OSS nodes (`oss0` and `oss1` respectively).

a. Create `ost0`. On `oss0` node, run:

```
[root@oss0 /]# mkfs.lustre --fsname=temp --mgsnode=10.2.0.1@tcp0 --ost --index
```

The command generates this output:

```
    Permanent disk data:
Target:            temp-OST0000
Index:             0
Lustre FS: temp
Mount type:        ldiskfs
Flags:             0x72
(OST first_time update)
Persistent mount opts: errors=remount-ro,extents,mballoc
Parameters: mgsnode=10.2.0.1@tcp

checking for existing Lustre data: not found
device size = 16MB
2 6 18
formatting backing filesystem ldiskfs on /dev/sdc
   target name             temp-OST0000
   4k blocks               0
   options                 -I 256 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L temp-OST0000  -I 256 -q -O
dir_index,uninit_groups -F /dev/sdc
Writing CONFIGS/mountdata
```

b. Mount ost0 on the OSS on which it was created. On `oss0` node, run:

```
root@oss0 /] mount -t lustre /dev/sdc /mnt/ost0
```

The command generates this output:

```
LDISKFS-fs: file extents enabled
LDISKFS-fs: mballoc enabled
Lustre: temp-OST0000: new disk, initializing
Lustre: Server temp-OST0000 on device /dev/sdb has started
```

Shortly afterwards, this output appears:

```
Lustre: temp-OST0000: received MDS connection from 10.2.0.1@tcp0
Lustre: MDS temp-MDT0000: temp-OST0000_UUID now active, resetting orphans
```

4. Create and mount `ost1`.

a. Create ost1. On `oss1` node, run:

```
[root@oss1 /]# mkfs.lustre --fsname=temp --mgsnode=10.2.0.1@tcp0 \
          --ost --index=1 /dev/sdd
```

The command generates this output:

```
    Permanent disk data:
Target:            temp-OST0001
Index:             1
Lustre FS: temp
Mount type:        ldiskfs
Flags:             0x72
(OST first_time update)
Persistent mount opts: errors=remount-ro,extents,mballoc
Parameters: mgsnode=10.2.0.1@tcp
```

```
checking for existing Lustre data: not found
device size = 16MB
2 6 18
formatting backing filesystem ldiskfs on /dev/sdd
   target name             temp-OST0001
   4k blocks               0
   options                 -I 256 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L temp-OST0001  -I 256 -q -O
dir_index,uninit_groups -F /dev/sdc
Writing CONFIGS/mountdata
```

b. Mount ost1 on the OSS on which it was created. On `oss1` node, run:

```
root@oss1 /] mount -t lustre /dev/sdd /mnt/ost1
```

The command generates this output:

```
LDISKFS-fs: file extents enabled
LDISKFS-fs: mballoc enabled
Lustre: temp-OST0001: new disk, initializing
Lustre: Server temp-OST0001 on device /dev/sdb has started
```

Shortly afterwards, this output appears:

```
Lustre: temp-OST0001: received MDS connection from 10.2.0.1@tcp0
Lustre: MDS temp-MDT0000: temp-OST0001_UUID now active, resetting orphans
```

5. Mount the Lustre file system on the client. On the client node, run:

```
root@client1 /] mount -t lustre 10.2.0.1@tcp0:/temp /lustre
```

This command generates this output:

```
Lustre: Client temp-client has started
```

6. Verify that the file system started and is working by running the `df`, `dd` and `ls` commands on the client node.

a. Run the `lfs df -h` command:

```
[root@client1 /] lfs df -h
```

The `lfs df -h` command lists space usage per OST and the MDT in human-readable format. This command generates output similar to this:

```
UUID               bytes      Used      Available   Use%    Mounted on
temp-MDT0000_UUID  8.0G      400.0M       7.6G        0%      /lustre[MDT:0]
temp-OST0000_UUID  800.0G    400.0M     799.6G        0%      /lustre[OST:0]
temp-OST0001_UUID  800.0G    400.0M     799.6G        0%      /lustre[OST:1]
filesystem summary:  1.6T    800.0M       1.6T        0%      /lustre
```

b. Run the `lfs df -ih` command.

```
[root@client1 /] lfs df -ih
```

The `lfs df -ih` command lists inode usage per OST and the MDT. This command generates output similar to this:

```
UUID               Inodes      IUsed       IFree   IUse%     Mounted on
temp-MDT0000_UUID   2.5M         32         2.5M      0%       /lustre[MDT:0]
temp-OST0000_UUID   5.5M         54         5.5M      0%       /lustre[OST:0]
temp-OST0001_UUID   5.5M         54         5.5M      0%       /lustre[OST:1]
filesystem summary: 2.5M         32         2.5M      0%       /lustre
```

c. Run the `dd` command:

```
[root@client1 /] cd /lustre
[root@client1 /lustre] dd if=/dev/zero of=/lustre/zero.dat bs=4M count=2
```

The `dd` command verifies write functionality by creating a file containing all zeros (`0`s). In this command, an 8 MB file is created. This command generates output similar to this:

```
2+0 records in
2+0 records out
8388608 bytes (8.4 MB) copied, 0.159628 seconds, 52.6 MB/s
```

d. Run the `ls` command:

```
[root@client1 /lustre] ls -lsah
```

The `ls -lsah` command lists files and directories in the current working directory. This command generates output similar to this:

```
total 8.0M
4.0K drwxr-xr-x  2 root root 4.0K Oct 16 15:27 .
8.0K drwxr-xr-x 25 root root 4.0K Oct 16 15:27 ..
8.0M -rw-r--r--  1 root root 8.0M Oct 16 15:27 zero.dat
```

Once the Lustre file system is configured, it is ready for use.

10.2. Additional Configuration Options

This section describes how to scale the Lustre file system or make configuration changes using the Lustre configuration utilities.

10.2.1. Scaling the Lustre File System

A Lustre file system can be scaled by adding OSTs or clients. For instructions on creating additional OSTs repeat Step 3 and Step 5 above. For mounting additional clients, repeat Step 6 for each client.

10.2.2. Changing Striping Defaults

The default settings for the file layout stripe pattern are shown in Table 10.1, “Default stripe pattern”.

Table 10.1. Default stripe pattern

File Layout Parameter	Default	Description

stripe_size	1 MB	Amount of data to write to one OST before moving to the next OST.
stripe_count	1	The number of OSTs to use for a single file.
start_ost	-1	The first OST where objects are created for each file. The default -1 allows the MDS to choose the starting index based on available space and load balancing. *It's strongly recommended not to change the default for this parameter to a value other than -1.*

Use the `lfs setstripe` command described in Chapter 18, *Managing File Layout (Striping) and Free Space* to change the file layout configuration.

10.2.3. Using the Lustre Configuration Utilities

If additional configuration is necessary, several configuration utilities are available:

- `mkfs.lustre` - Use to format a disk for a Lustre service.
- `tunefs.lustre` - Use to modify configuration information on a Lustre target disk.
- `lctl` - Use to directly control Lustre features via an `ioctl` interface, allowing various configuration, maintenance and debugging features to be accessed.
- `mount.lustre` - Use to start a Lustre client or target service.

For examples using these utilities, see the topic Chapter 37, *System Configuration Utilities*

The `lfs` utility is useful for configuring and querying a variety of options related to files. For more information, see Chapter 33, *User Utilities*.

Note

Some sample scripts are included in the directory where the Lustre software is installed. If you have installed the Lustre source code, the scripts are located in the `lustre/tests` sub-directory. These scripts enable quick setup of some simple standard Lustre configurations.

Chapter 11. Configuring Failover in a Lustre File System

This chapter describes how to configure failover in a Lustre file system. It includes:

- Section 11.1, “Setting Up a Failover Environment”
- Section 11.2, “Preparing a Lustre File System for Failover”
- Section 11.3, “Administering Failover in a Lustre File System”

For an overview of failover functionality in a Lustre file system, see Chapter 3, *Understanding Failover in a Lustre File System*.

11.1. Setting Up a Failover Environment

The Lustre software provides failover mechanisms only at the layer of the Lustre file system. No failover functionality is provided for system-level components such as failing hardware or applications, or even for the entire failure of a node, as would typically be provided in a complete failover solution. Failover functionality such as node monitoring, failure detection, and resource fencing must be provided by external HA software, such as PowerMan or the open source Corosync and Pacemaker packages provided by Linux operating system vendors. Corosync provides support for detecting failures, and Pacemaker provides the actions to take once a failure has been detected.

11.1.1. Selecting Power Equipment

Failover in a Lustre file system requires the use of a remote power control (RPC) mechanism, which comes in different configurations. For example, Lustre server nodes may be equipped with IPMI/BMC devices that allow remote power control. In the past, software or even “sneakerware” has been used, but these are not recommended. For recommended devices, refer to the list of supported RPC devices on the website for the PowerMan cluster power management utility:

http://code.google.com/p/powerman/wiki/SupportedDevs

11.1.2. Selecting Power Management Software

Lustre failover requires RPC and management capability to verify that a failed node is shut down before I/O is directed to the failover node. This avoids double-mounting the two nodes and the risk of unrecoverable data corruption. A variety of power management tools will work. Two packages that have been commonly used with the Lustre software are PowerMan and Linux-HA (aka. STONITH).

The PowerMan cluster power management utility is used to control RPC devices from a central location. PowerMan provides native support for several RPC varieties and Expect-like configuration simplifies the addition of new devices. The latest versions of PowerMan are available at:

http://code.google.com/p/powerman/

STONITH, or “Shoot The Other Node In The Head”, is a set of power management tools provided with the Linux-HA package prior to Red Hat Enterprise Linux 6. Linux-HA has native support for many power control devices, is extensible (uses Expect scripts to automate control), and provides the software to detect and respond to failures. With Red Hat Enterprise Linux 6, Linux-HA is being replaced in the open source

community by the combination of Corosync and Pacemaker. For Red Hat Enterprise Linux subscribers, cluster management using CMAN is available from Red Hat.

11.1.3. Selecting High-Availability (HA) Software

The Lustre file system must be set up with high-availability (HA) software to enable a complete Lustre failover solution. Except for PowerMan, the HA software packages mentioned above provide both power management and cluster management. For information about setting up failover with Pacemaker, see:

- Pacemaker Project website: http://clusterlabs.org/
- Article *Using Pacemaker with a Lustre File System*: https://wiki.hpdd.intel.com/display/PUB/Using+Pacemaker+with+a+Lustre+File+System

11.2. Preparing a Lustre File System for Failover

To prepare a Lustre file system to be configured and managed as an HA system by a third-party HA application, each storage target (MGT, MGS, OST) must be associated with a second node to create a failover pair. This configuration information is then communicated by the MGS to a client when the client mounts the file system.

The per-target configuration is relayed to the MGS at mount time. Some rules related to this are:

- When a target is initially mounted, the MGS reads the configuration information from the target (such as mgt vs. ost, failnode, fsname) to configure the target into a Lustre file system. If the MGS is reading the initial mount configuration, the mounting node becomes that target's “primary” node.
- When a target is subsequently mounted, the MGS reads the current configuration from the target and, as needed, will reconfigure the MGS database target information

When the target is formatted using the `mkfs.lustre`command, the failover service node(s) for the target are designated using the `--servicenode` option. In the example below, an OST with index `0` in the file system `testfs` is formatted with two service nodes designated to serve as a failover pair:

```
mkfs.lustre --reformat --ost --fsname testfs --mgsnode=192.168.10.1@o3ib \
              --index=0 --servicenode=192.168.10.7@o2ib \
              --servicenode=192.168.10.8@o2ib \
              /dev/sdb
```

More than two potential service nodes caan be designated for a target. The target can then be mounted on any of the designated service nodes.

When HA is configured on a storage target, the Lustre software enables multi-mount protection (MMP) on that storage target. MMP prevents multiple nodes from simultaneously mounting and thus corrupting the data on the target. For more about MMP, see Chapter 20, *Lustre File System Failover and Multiple-Mount Protection*.

If the MGT has been formatted with multiple service nodes designated, this information must be conveyed to the Lustre client in the mount command used to mount the file system. In the example below, NIDs for two MGSs that have been designated as service nodes for the MGT are specified in the mount command executed on the client:

```
mount -t lustre 10.10.120.1@tcp1:10.10.120.2@tcp1:/testfs /lustre/testfs
```

When a client mounts the file system, the MGS provides configuration information to the client for the MDT(s) and OST(s) in the file system along with the NIDs for all service nodes associated with each target and the service node on which the target is mounted. Later, when the client attempts to access data on a target, it will try the NID for each specified service node until it connects to the target.

Previous to Lustre software release 2.0, the `--failnode` option to `mkfs.lustre` was used to designate a failover service node for a primary server for a target. When the `--failnode` option is used, certain restrictions apply:

- The target must be initially mounted on the primary service node, not the failover node designated by the `--failnode` option.
- If the `tunefs.lustre –-writeconf` option is used to erase and regenerate the configuration log for the file system, a target cannot be initially mounted on a designated failnode.
- If a `--failnode` option is added to a target to designate a failover server for the target, the target must be re-mounted on the primary node before the `--failnode` option takes effect

11.3. Administering Failover in a Lustre File System

For additional information about administering failover features in a Lustre file system, see:

- Section 13.5, “Specifying Failout/Failover Mode for OSTs”
- Section 13.10, “Specifying NIDs and Failover”
- Section 14.11, “ Changing the Address of a Failover Node”
- Section 37.14, “ mkfs.lustre”

Part III. Administering Lustre

Part III provides information about tools and procedures to use to administer a Lustre file system. You will find information in this section about:

- Monitoring a Lustre File System
- Lustre Maintenance
- Managing Lustre Networking (LNET)
- Upgrading a Lustre File System
- Backing Up and Restoring a File System
- Managing File Layout (Striping) and Free Space
- Managing the File System and I/O
- Lustre File System Failover and Multiple-Mount Protection
- Configuring and Managing Quotas
- Hierarchical Storage Management (HSM)
- Managing Security in a Lustre File System

Tip

The starting point for administering a Lustre file system is to monitor all logs and console logs for system health:

- Monitor logs on all servers and all clients.

- Invest in tools that allow you to condense logs from multiple systems.

- Use the logging resources provided in the Linux distribution.

Chapter 12. Monitoring a Lustre File System

This chapter provides information on monitoring a Lustre file system and includes the following sections:

- Section 12.1, “ Lustre Changelogs”Lustre Changelogs
- Section 12.2, “ Lustre Jobstats”Lustre Jobstats
- Section 12.3, “ Lustre Monitoring Tool (LMT)”Lustre Monitoring Tool
- Section 12.4, “ `CollectL` ”CollectL
- Section 12.5, “ Other Monitoring Options”Other Monitoring Options

12.1. Lustre Changelogs

The changelogs feature records events that change the file system namespace or file metadata. Changes such as file creation, deletion, renaming, attribute changes, etc. are recorded with the target and parent file identifiers (FIDs), the name of the target, and a timestamp. These records can be used for a variety of purposes:

- Capture recent changes to feed into an archiving system.
- Use changelog entries to exactly replicate changes in a file system mirror.
- Set up "watch scripts" that take action on certain events or directories.
- Maintain a rough audit trail (file/directory changes with timestamps, but no user information).

Changelogs record types are:

Value	Description
MARK	Internal recordkeeping
CREAT	Regular file creation
MKDIR	Directory creation
HLINK	Hard link
SLINK	Soft link
MKNOD	Other file creation
UNLNK	Regular file removal
RMDIR	Directory removal
RNMFM	Rename, original
RNMTO	Rename, final
IOCTL	ioctl on file or directory
TRUNC	Regular file truncated
SATTR	Attribute change

Value	Description
XATTR	Extended attribute change
UNKNW	Unknown operation

FID-to-full-pathname and pathname-to-FID functions are also included to map target and parent FIDs into the file system namespace.

12.1.1. Working with Changelogs

Several commands are available to work with changelogs.

12.1.1.1. `lctl changelog_register`

Because changelog records take up space on the MDT, the system administration must register changelog users. The registrants specify which records they are "done with", and the system purges up to the greatest common record.

To register a new changelog user, run:

```
lctl --device /dev/mdt_device changelog_register
```

Changelog entries are not purged beyond a registered user's set point (see `lfs changelog_clear`).

12.1.1.2. `lfs changelog`

To display the metadata changes on an MDT (the changelog records), run:

```
lfs changelog fsname-MDTnumber [startrec [endrec]]
```

It is optional whether to specify the start and end records.

These are sample changelog records:

```
2 02MKDIR 4298396676 0x0 t=[0x200000405:0x15f9:0x0] p=[0x13:0x15e5a7a3:0x0]\
 pics
3 01CREAT 4298402264 0x0 t=[0x200000405:0x15fa:0x0] p=[0x200000405:0x15f9:0\
x0] chloe.jpg
4 06UNLNK 4298404466 0x0 t=[0x200000405:0x15fa:0x0] p=[0x200000405:0x15f9:0\
x0] chloe.jpg
5 07RMDIR 4298405394 0x0 t=[0x200000405:0x15f9:0x0] p=[0x13:0x15e5a7a3:0x0]\
 pics
```

12.1.1.3. `lfs changelog_clear`

To clear old changelog records for a specific user (records that the user no longer needs), run:

```
lfs changelog_clear mdt_name userid endrec
```

The `changelog_clear` command indicates that changelog records previous to *`endrec`* are no longer of interest to a particular user *`userid`*, potentially allowing the MDT to free up disk space. An *`endrec`* value of 0 indicates the current last record. To run `changelog_clear`, the changelog user must be registered on the MDT node using `lctl`.

When all changelog users are done with records < X, the records are deleted.

12.1.1.4. lctl changelog_deregister

To deregister (unregister) a changelog user, run:

```
lctl --device mdt_device changelog_deregister userid
```

`changelog_deregister cl1` effectively does a `changelog_clear cl1 0` as it deregisters.

12.1.2. Changelog Examples

This section provides examples of different changelog commands.

12.1.2.1. Registering a Changelog User

To register a new changelog user for a device (`lustre-MDT0000`):

```
# lctl --device lustre-MDT0000 changelog_register
lustre-MDT0000: Registered changelog userid 'cl1'
```

12.1.2.2. Displaying Changelog Records

To display changelog records on an MDT (`lustre-MDT0000`):

```
$ lfs changelog lustre-MDT0000
1 00MARK  19:08:20.890432813 2010.03.24 0x0 t=[0x10001:0x0:0x0] p=[0:0x0:0x\
0] mdd_obd-lustre-MDT0000-0
2 02MKDIR 19:10:21.509659173 2010.03.24 0x0 t=[0x200000420:0x3:0x0] p=[0x61\
b4:0xca2c7dde:0x0] mydir
3 14SATTR 19:10:27.329356533 2010.03.24 0x0 t=[0x200000420:0x3:0x0]
4 01CREAT 19:10:37.113847713 2010.03.24 0x0 t=[0x200000420:0x4:0x0] p=[0x20\
0000420:0x3:0x0] hosts
```

Changelog records include this information:

```
rec#
operation_type(numerical/text)
timestamp
datestamp
flags
t=target_FID
p=parent_FID
target_name
```

Displayed in this format:

```
rec# operation_type(numerical/text) timestamp datestamp flags t=target_FID \
p=parent_FID target_name
```

For example:

```
4 01CREAT 19:10:37.113847713 2010.03.24 0x0 t=[0x200000420:0x4:0x0] p=[0x20\
0000420:0x3:0x0] hosts
```

12.1.2.3. Clearing Changelog Records

To notify a device that a specific user (`cl1`) no longer needs records (up to and including 3):

```
$ lfs changelog_clear  lustre-MDT0000 cl1 3
```

To confirm that the `changelog_clear` operation was successful, run `lfs changelog`; only records after id-3 are listed:

```
$ lfs changelog lustre-MDT0000
4 01CREAT 19:10:37.113847713 2010.03.24 0x0 t=[0x200000420:0x4:0x0] p=[0x20\
0000420:0x3:0x0] hosts
```

12.1.2.4. Deregistering a Changelog User

To deregister a changelog user (`cl1`) for a specific device (`lustre-MDT0000`):

```
# lctl --device lustre-MDT0000 changelog_deregister cl1
lustre-MDT0000: Deregistered changelog user 'cl1'
```

The deregistration operation clears all changelog records for the specified user (`cl1`).

```
$ lfs changelog lustre-MDT0000
5 00MARK  19:13:40.858292517 2010.03.24 0x0 t=[0x40001:0x0:0x0] p=[0:0x0:0x\
0] mdd_obd-lustre-MDT0000-0
```

Note

MARK records typically indicate changelog recording status changes.

12.1.2.5. Displaying the Changelog Index and Registered Users

To display the current, maximum changelog index and registered changelog users for a specific device (`lustre-MDT0000`):

```
# lctl get_param  mdd.lustre-MDT0000.changelog_users
mdd.lustre-MDT0000.changelog_users=current index: 8
ID    index
cl2   8
```

12.1.2.6. Displaying the Changelog Mask

To show the current changelog mask on a specific device (`lustre-MDT0000`):

```
# lctl get_param  mdd.lustre-MDT0000.changelog_mask

mdd.lustre-MDT0000.changelog_mask=
MARK CREAT MKDIR HLINK SLINK MKNOD UNLNK RMDIR RNMFM RNMTO OPEN CLOSE IOCTL\
 TRUNC SATTR XATTR HSM
```

12.1.2.7. Setting the Changelog Mask

To set the current changelog mask on a specific device (`lustre-MDT0000`):

```
# lctl set_param mdd.lustre-MDT0000.changelog_mask=HLINK
mdd.lustre-MDT0000.changelog_mask=HLINK
$ lfs changelog_clear lustre-MDT0000 cl1 0
$ mkdir /mnt/lustre/mydir/foo
$ cp /etc/hosts /mnt/lustre/mydir/foo/file
$ ln /mnt/lustre/mydir/foo/file /mnt/lustre/mydir/myhardlink
```

Only item types that are in the mask show up in the changelog.

```
$ lfs changelog lustre-MDT0000
9 03HLINK 19:19:35.171867477 2010.03.24 0x0 t=[0x200000420:0x6:0x0] p=[0x20\
0000420:0x3:0x0] myhardlink
```

12.2. Lustre Jobstats

The Lustre jobstats feature is available starting in Lustre software release 2.3. It collects file system operation statistics for the jobs running on Lustre clients, and exposes them via procfs on the server. Job schedulers known to be able to work with jobstats include: SLURM, SGE, LSF, Loadleveler, PBS and Maui/MOAB.

Since jobstats is implemented in a scheduler-agnostic manner, it is likely that it will be able to work with other schedulers also.

12.2.1. Enable/Disable Jobstats

Jobstats are disabled by default. The current state of jobstats can be verified by checking `lctl get_param jobid_var` on a client:

```
$ lctl get_param jobid_var
jobid_var=disable
```

The Lustre jobstats code extracts the job identifier from an environment variable set by the scheduler when the job is started. To enable jobstats, specify the `jobid_var` to name the environment variable set by the scheduler. For example, SLURM sets the `SLURM_JOB_ID` environment variable with the unique job ID on each client. To permanently enable jobstats on the `testfs` file system:

```
$ lctl conf_param testfs.sys.jobid_var=SLURM_JOB_ID
```

The following table shows the environment variables which are set by various job schedulers. Set `jobid_var` to the value for your job scheduler to collect statistics on a per job basis.

Job Scheduler	**Environment Variable**
Simple Linux Utility for Resource Management (SLURM)	SLURM_JOB_ID
Sun Grid Engine (SGE)	JOB_ID
Load Sharing Facility (LSF)	LSB_JOBID
Loadleveler	LOADL_STEP_ID
Portable Batch Scheduler (PBS)/MAUI	PBS_JOBID
Cray Application Level Placement Scheduler (ALPS)	ALPS_APP_ID

There are two special values for `jobid_var`: `disable` and `procname_uid`. To disable jobstats, specify `jobid_var` as `disable`:

```
$ lctl conf_param testfs.sys.jobid_var=disable
```

To track job stats per process name and user ID (for debugging, or if no job scheduler is in use), specify `jobid_var` as `procname_uid`:

```
$ lctl conf_param testfs.sys.jobid_var=procname_uid
```

12.2.2. Check Job Stats

Metadata operation statistics are collected on MDTs. These statistics can be accessed for all file systems and all jobs on the MDT via the `lctl get_param mdt.*.job_stats`. For example, clients running with `jobid_var=procname_uid`:

```
$ lctl get_param mdt.*.job_stats
job_stats:
- job_id:          bash.0
  snapshot_time:   1352084992
  open:            { samples:           2, unit:  reqs }
  close:           { samples:           2, unit:  reqs }
  mknod:           { samples:           0, unit:  reqs }
  link:            { samples:           0, unit:  reqs }
  unlink:          { samples:           0, unit:  reqs }
  mkdir:           { samples:           0, unit:  reqs }
  rmdir:           { samples:           0, unit:  reqs }
  rename:          { samples:           0, unit:  reqs }
  getattr:         { samples:           3, unit:  reqs }
  setattr:         { samples:           0, unit:  reqs }
  getxattr:        { samples:           0, unit:  reqs }
  setxattr:        { samples:           0, unit:  reqs }
  statfs:          { samples:           0, unit:  reqs }
  sync:            { samples:           0, unit:  reqs }
  samedir_rename:  { samples:           0, unit:  reqs }
  crossdir_rename: { samples:           0, unit:  reqs }
- job_id:          dd.0
  snapshot_time:   1352085037
  open:            { samples:           1, unit:  reqs }
  close:           { samples:           1, unit:  reqs }
  mknod:           { samples:           0, unit:  reqs }
  link:            { samples:           0, unit:  reqs }
  unlink:          { samples:           0, unit:  reqs }
  mkdir:           { samples:           0, unit:  reqs }
  rmdir:           { samples:           0, unit:  reqs }
  rename:          { samples:           0, unit:  reqs }
  getattr:         { samples:           0, unit:  reqs }
  setattr:         { samples:           0, unit:  reqs }
  getxattr:        { samples:           0, unit:  reqs }
  setxattr:        { samples:           0, unit:  reqs }
  statfs:          { samples:           0, unit:  reqs }
  sync:            { samples:           2, unit:  reqs }
  samedir_rename:  { samples:           0, unit:  reqs }
  crossdir_rename: { samples:           0, unit:  reqs }
```

Data operation statistics are collected on OSTs. Data operations statistics can be accessed via `lctl get_param obdfilter.*.job_stats`, for example:

```
$ lctl get_param obdfilter.*.job_stats
```

```
job_stats:
- job_id:          bash.0
  snapshot_time:   1352085025
  read:            { samples:           0, unit: bytes, min:       0, max:       0
  write:           { samples:           1, unit: bytes, min:       4, max:       4
  setattr:         { samples:           0, unit:  reqs }
  punch:           { samples:           0, unit:  reqs }
  sync:            { samples:           0, unit:  reqs }
```

12.2.3. Clear Job Stats

Accumulated job statistics can be reset by writing proc file `job_stats`.

Clear statistics for all jobs on the local node:

```
$ lctl set_param obdfilter.*.job_stats=clear
```

Clear statistics for job 'dd.0' on lustre-MDT0000:

```
$ lctl set_param mdt.lustre-MDT0000.job_stats=clear
```

12.2.4. Configure Auto-cleanup Interval

By default, if a job is inactive for 600 seconds (10 minutes) statistics for this job will be dropped. This expiration value can be changed temporarily via:

```
$ lctl set_param *.*.job_cleanup_interval={max_age}
```

It can also be changed permanently, for example to 700 seconds via:

```
$ lctl conf_param testfs.mdt.job_cleanup_interval=700
```

The `job_cleanup_interval` can be set as 0 to disable the auto-cleanup. Note that if auto-cleanup of Jobstats is disabled, then all statistics will be kept in memory forever, which may eventually consume all memory on the servers. In this case, any monitoring tool should explicitly clear individual job statistics as they are processed, as shown above.

12.3. Lustre Monitoring Tool (LMT)

The Lustre Monitoring Tool (LMT) is a Python-based, distributed system that provides a `top`-like display of activity on server-side nodes (MDS, OSS and portals routers) on one or more Lustre file systems. It does not provide support for monitoring clients. For more information on LMT, including the setup procedure, see:

https://github.com/chaos/lmt/wiki [http://code.google.com/p/lmt/]

LMT questions can be directed to:

lmt-discuss@googlegroups.com [mailto:lmt-discuss@googlegroups.com]

12.4. `CollectL`

`CollectL` is another tool that can be used to monitor a Lustre file system. You can run `CollectL` on a Lustre system that has any combination of MDSs, OSTs and clients. The collected data can be written to

a file for continuous logging and played back at a later time. It can also be converted to a format suitable for plotting.

For more information about `CollectL`, see:

http://collectl.sourceforge.net

Lustre-specific documentation is also available. See:

http://collectl.sourceforge.net/Tutorial-Lustre.html

12.5. Other Monitoring Options

A variety of standard tools are available publicly including the following:

- `lltop` - Lustre load monitor with batch scheduler integration. https://github.com/jhammond/lltop
- `tacc_stats` - A job-oriented system monitor, analyzation, and visualization tool that probes Lustre interfaces and collects statistics. https://github.com/jhammond/tacc_stats
- `xltop` - A continuous Lustre monitor with batch scheduler integation. https://github.com/jhammond/xltop

Another option is to script a simple monitoring solution that looks at various reports from `ipconfig`, as well as the `procfs` files generated by the Lustre software.

Chapter 13. Lustre Operations

Once you have the Lustre file system up and running, you can use the procedures in this section to perform these basic Lustre administration tasks:

- Section 13.1, " Mounting by Label"
- Section 13.2, "Starting Lustre"
- Section 13.3, "Mounting a Server"
- Section 13.4, "Unmounting a Server"
- Section 13.5, "Specifying Failout/Failover Mode for OSTs"
- Section 13.6, "Handling Degraded OST RAID Arrays"
- Section 13.7, "Running Multiple Lustre File Systems"
- Section 13.8, "Creating a sub-directory on a given MDT"
- Section 13.9, "Setting and Retrieving Lustre Parameters"
- Section 13.10, "Specifying NIDs and Failover"
- Section 13.11, "Erasing a File System"
- Section 13.12, "Reclaiming Reserved Disk Space"
- Section 13.13, "Replacing an Existing OST or MDT"
- Section 13.14, "Identifying To Which Lustre File an OST Object Belongs"

13.1. Mounting by Label

The file system name is limited to 8 characters. We have encoded the file system and target information in the disk label, so you can mount by label. This allows system administrators to move disks around without worrying about issues such as SCSI disk reordering or getting the `/dev/device` wrong for a shared target. Soon, file system naming will be made as fail-safe as possible. Currently, Linux disk labels are limited to 16 characters. To identify the target within the file system, 8 characters are reserved, leaving 8 characters for the file system name:

fsname`-MDT0000 or `*fsname*`-OST0a19`

To mount by label, use this command:

```
mount -t lustre -L file_system_label /mount_point
```

This is an example of mount-by-label:

```
mds# mount -t lustre -L testfs-MDT0000 /mnt/mdt
```

Caution

Mount-by-label should NOT be used in a multi-path environment or when snapshots are being created of the device, since multiple block devices will have the same label.

Although the file system name is internally limited to 8 characters, you can mount the clients at any mount point, so file system users are not subjected to short names. Here is an example:

```
client# mount -t lustre mds0@tcp0:/short /dev/long_mountpoint_name
```

13.2. Starting Lustre

On the first start of a Lustre file system, the components must be started in the following order:

1. Mount the MGT.

   Note

   If a combined MGT/MDT is present, Lustre will correctly mount the MGT and MDT automatically.

2. Mount the MDT.

   Note

   Introduced in Lustre 2.4

   Mount all MDTs if multiple MDTs are present.

3. Mount the OST(s).

4. Mount the client(s).

13.3. Mounting a Server

Starting a Lustre server is straightforward and only involves the mount command. Lustre servers can be added to `/etc/fstab`:

```
mount -t lustre
```

The mount command generates output similar to this:

```
/dev/sda1 on /mnt/test/mdt type lustre (rw)
/dev/sda2 on /mnt/test/ost0 type lustre (rw)
192.168.0.21@tcp:/testfs on /mnt/testfs type lustre (rw)
```

In this example, the MDT, an OST (ost0) and file system (testfs) are mounted.

```
LABEL=testfs-MDT0000 /mnt/test/mdt lustre defaults,_netdev,noauto 0 0
LABEL=testfs-OST0000 /mnt/test/ost0 lustre defaults,_netdev,noauto 0 0
```

In general, it is wise to specify noauto and let your high-availability (HA) package manage when to mount the device. If you are not using failover, make sure that networking has been started before mounting a Lustre server. If you are running Red Hat Enterprise Linux, SUSE Linux Enterprise Server, Debian operating system (and perhaps others), use the `_netdev` flag to ensure that these disks are mounted after the network is up.

We are mounting by disk label here. The label of a device can be read with `e2label`. The label of a newly-formatted Lustre server may end in `FFFF` if the `--index` option is not specified to `mkfs.lustre`, meaning that it has yet to be assigned. The assignment takes place when the server is first started, and the

disk label is updated. It is recommended that the `--index` option always be used, which will also ensure that the label is set at format time.

Caution

Do not do this when the client and OSS are on the same node, as memory pressure between the client and OSS can lead to deadlocks.

Caution

Mount-by-label should NOT be used in a multi-path environment.

13.4. Unmounting a Server

To stop a Lustre server, use the `umount` *`/mount point`* command.

For example, to stop `ost0` on mount point `/mnt/test`, run:

```
$ umount /mnt/test
```

Gracefully stopping a server with the `umount` command preserves the state of the connected clients. The next time the server is started, it waits for clients to reconnect, and then goes through the recovery procedure.

If the force (`-f`) flag is used, then the server evicts all clients and stops WITHOUT recovery. Upon restart, the server does not wait for recovery. Any currently connected clients receive I/O errors until they reconnect.

Note

If you are using loopback devices, use the `-d` flag. This flag cleans up loop devices and can always be safely specified.

13.5. Specifying Failout/Failover Mode for OSTs

In a Lustre file system, an OST that has become unreachable because it fails, is taken off the network, or is unmounted can be handled in one of two ways:

- In `failout` mode, Lustre clients immediately receive errors (EIOs) after a timeout, instead of waiting for the OST to recover.
- In `failover` mode, Lustre clients wait for the OST to recover.

By default, the Lustre file system uses `failover` mode for OSTs. To specify `failout` mode instead, use the `--param="failover.mode=failout"` option as shown below (entered on one line):

```
oss# mkfs.lustre --fsname=fsname --mgsnode=mgs_NID --param=failover.mode=failout
      --ost --index=ost_index /dev/ost_block_device
```

In the example below, `failout` mode is specified for the OSTs on the MGS `mds0` in the file system `testfs` (entered on one line).

```
oss# mkfs.lustre --fsname=testfs --mgsnode=mds0 --param=failover.mode=failout
      --ost --index=3 /dev/sdb
```

Caution

Before running this command, unmount all OSTs that will be affected by a change in `failover` / `failout` mode.

Note

After initial file system configuration, use the `tunefs.lustre` utility to change the mode. For example, to set the `failout` mode, run:

```
$ tunefs.lustre --param failover.mode=failout /dev/ost_device
```

13.6. Handling Degraded OST RAID Arrays

Lustre includes functionality that notifies Lustre if an external RAID array has degraded performance (resulting in reduced overall file system performance), either because a disk has failed and not been replaced, or because a disk was replaced and is undergoing a rebuild. To avoid a global performance slowdown due to a degraded OST, the MDS can avoid the OST for new object allocation if it is notified of the degraded state.

A parameter for each OST, called `degraded`, specifies whether the OST is running in degraded mode or not.

To mark the OST as degraded, use:

```
lctl set_param obdfilter.{OST_name}.degraded=1
```

To mark that the OST is back in normal operation, use:

```
lctl set_param obdfilter.{OST_name}.degraded=0
```

To determine if OSTs are currently in degraded mode, use:

```
lctl get_param obdfilter.*.degraded
```

If the OST is remounted due to a reboot or other condition, the flag resets to `0`.

It is recommended that this be implemented by an automated script that monitors the status of individual RAID devices.

13.7. Running Multiple Lustre File Systems

Lustre supports multiple file systems provided the combination of `NID:fsname` is unique. Each file system must be allocated a unique name during creation with the `--fsname` parameter. Unique names for file systems are enforced if a single MGS is present. If multiple MGSs are present (for example if you have an MGS on every MDS) the administrator is responsible for ensuring file system names are unique. A single MGS and unique file system names provides a single point of administration and allows commands to be issued against the file system even if it is not mounted.

Lustre supports multiple file systems on a single MGS. With a single MGS fsnames are guaranteed to be unique. Lustre also allows multiple MGSs to co-exist. For example, multiple MGSs will be necessary if

multiple file systems on different Lustre software versions are to be concurrently available. With multiple MGSs additional care must be taken to ensure file system names are unique. Each file system should have a unique fsname among all systems that may interoperate in the future.

By default, the `mkfs.lustre` command creates a file system named `lustre`. To specify a different file system name (limited to 8 characters) at format time, use the `--fsname` option:

```
mkfs.lustre --fsname=file_system_name
```

Note

The MDT, OSTs and clients in the new file system must use the same file system name (prepended to the device name). For example, for a new file system named `foo`, the MDT and two OSTs would be named `foo-MDT0000`, `foo-OST0000`, and `foo-OST0001`.

To mount a client on the file system, run:

```
client# mount -t lustre mgsnode:/new_fsname /mount_point
```

For example, to mount a client on file system foo at mount point /mnt/foo, run:

```
client# mount -t lustre mgsnode:/foo /mnt/foo
```

Note

If a client(s) will be mounted on several file systems, add the following line to `/etc/xattr.conf` file to avoid problems when files are moved between the file systems: `lustre.* skip`

Note

To ensure that a new MDT is added to an existing MGS create the MDT by specifying: `--mdt --mgsnode=mgs_NID`.

A Lustre installation with two file systems (`foo` and `bar`) could look like this, where the MGS node is `mgsnode@tcp0` and the mount points are `/mnt/foo` and `/mnt/bar`.

```
mgsnode# mkfs.lustre --mgs /dev/sda
mdtfoonode# mkfs.lustre --fsname=foo --mgsnode=mgsnode@tcp0 --mdt --index=0 /dev/s
ossfoonode# mkfs.lustre --fsname=foo --mgsnode=mgsnode@tcp0 --ost --index=0 /dev/s
ossfoonode# mkfs.lustre --fsname=foo --mgsnode=mgsnode@tcp0 --ost --index=1 /dev/s
mdtbarnode# mkfs.lustre --fsname=bar --mgsnode=mgsnode@tcp0 --mdt --index=0 /dev/s
ossbarnode# mkfs.lustre --fsname=bar --mgsnode=mgsnode@tcp0 --ost --index=0 /dev/s
ossbarnode# mkfs.lustre --fsname=bar --mgsnode=mgsnode@tcp0 --ost --index=1 /dev/s
```

To mount a client on file system foo at mount point `/mnt/foo`, run:

```
client# mount -t lustre mgsnode@tcp0:/foo /mnt/foo
```

To mount a client on file system bar at mount point `/mnt/bar`, run:

```
client# mount -t lustre mgsnode@tcp0:/bar /mnt/bar
```

Introduced in Lustre 2.4

Lustre 2.4 enables individual sub-directories to be serviced by unique MDTs. An administrator can allocate a sub-directory to a given MDT using the command:

```
client# lfs mkdir –i mdt_index /mount_point/remote_dir
```

This command will allocate the sub-directory `remote_dir` onto the MDT of index `mdtindex`. For more information on adding additional MDTs and `mdtindex` see 2.

Warning

An administrator can allocate remote sub-directories to separate MDTs. Creating remote sub-directories in parent directories not hosted on MDT0 is not recommended. This is because the failure of the parent MDT will leave the namespace below it inaccessible. For this reason, by default it is only possible to create remote sub-directories off MDT0. To relax this restriction and enable remote sub-directories off any MDT, an administrator must issue the command `lctl set_param mdd.*.enable_remote_dir=1`.

13.9. Setting and Retrieving Lustre Parameters

Several options are available for setting parameters in Lustre:

- When creating a file system, use mkfs.lustre. See Section 13.9.1, “Setting Tunable Parameters with `mkfs.lustre`” below.
- When a server is stopped, use tunefs.lustre. See Section 13.9.2, “Setting Parameters with `tunefs.lustre`” below.
- When the file system is running, use lctl to set or retrieve Lustre parameters. See Section 13.9.3, “Setting Parameters with `lctl`” and Section 13.9.3.5, “Reporting Current Parameter Values” below.

13.9.1. Setting Tunable Parameters with `mkfs.lustre`

When the file system is first formatted, parameters can simply be added as a `--param` option to the `mkfs.lustre` command. For example:

```
mds# mkfs.lustre --mdt --param="sys.timeout=50" /dev/sda
```

For more details about creating a file system,see Chapter 10, *Configuring a Lustre File System*. For more details about `mkfs.lustre`, see Chapter 37, *System Configuration Utilities*.

13.9.2. Setting Parameters with `tunefs.lustre`

If a server (OSS or MDS) is stopped, parameters can be added to an existing file system using the `--param` option to the `tunefs.lustre` command. For example:

```
oss# tunefs.lustre --param=failover.node=192.168.0.13@tcp0 /dev/sda
```

With `tunefs.lustre`, parameters are *additive* -- new parameters are specified in addition to old parameters, they do not replace them. To erase all old `tunefs.lustre` parameters and just use newly-specified parameters, run:

```
mds# tunefs.lustre --erase-params --param=new_parameters
```

The tunefs.lustre command can be used to set any parameter settable in a /proc/fs/lustre file and that has its own OBD device, so it can be specified as *obdname*|*fsname*.*obdtype*.*proc_file_name*=*value*. For example:

```
mds# tunefs.lustre --param mdt.identity_upcall=NONE /dev/sda1
```

For more details about `tunefs.lustre`, see Chapter 37, *System Configuration Utilities*.

13.9.3. Setting Parameters with `lctl`

When the file system is running, the `lctl` command can be used to set parameters (temporary or permanent) and report current parameter values. Temporary parameters are active as long as the server or client is not shut down. Permanent parameters live through server and client reboots.

Note

The lctl list_param command enables users to list all parameters that can be set. See Section 13.9.3.4, “Listing Parameters”.

For more details about the `lctl` command, see the examples in the sections below and Chapter 37, *System Configuration Utilities*.

13.9.3.1. Setting Temporary Parameters

Use `lctl set_param` to set temporary parameters on the node where it is run. These parameters map to items in `/proc/{fs,sys}/{lnet,lustre}`. The `lctl set_param` command uses this syntax:

```
lctl set_param [-n] obdtype.obdname.proc_file_name=value
```

For example:

```
# lctl set_param osc.*.max_dirty_mb=1024
osc.myth-OST0000-osc.max_dirty_mb=32
osc.myth-OST0001-osc.max_dirty_mb=32
osc.myth-OST0002-osc.max_dirty_mb=32
osc.myth-OST0003-osc.max_dirty_mb=32
osc.myth-OST0004-osc.max_dirty_mb=32
```

13.9.3.2. Setting Permanent Parameters

Use the `lctl conf_param` command to set permanent parameters. In general, the `lctl conf_param` command can be used to specify any parameter settable in a `/proc/fs/lustre` file, with its own OBD device. The `lctl conf_param` command uses this syntax (same as the `mkfs.lustre` and `tunefs.lustre` commands):

```
obdname|fsname.obdtype.proc_file_name=value)
```

Here are a few examples of `lctl conf_param` commands:

```
mgs# lctl conf_param testfs-MDT0000.sys.timeout=40
$ lctl conf_param testfs-MDT0000.mdt.identity_upcall=NONE
$ lctl conf_param testfs.llite.max_read_ahead_mb=16
$ lctl conf_param testfs-MDT0000.lov.stripesize=2M
$ lctl conf_param testfs-OST0000.osc.max_dirty_mb=29.15
$ lctl conf_param testfs-OST0000.ost.client_cache_seconds=15
$ lctl conf_param testfs.sys.timeout=40
```

Caution

Parameters specified with the `lctl conf_param` command are set permanently in the file system's configuration file on the MGS.

Introduced in Lustre 2.5

Use the `lctl set_param -P` to set parameters permanently. This command must be issued on the MGS. The given parameter is set on every host using `lctl` upcall. Parameters map to items in `/proc/{fs,sys}/{lnet,lustre}`. The `lctl set_param` command uses this syntax:

```
lctl set_param -P obdtype.obdname.proc_file_name=value
```

For example:

```
# lctl set_param -P osc.*.max_dirty_mb=1024
osc.myth-OST0000-osc.max_dirty_mb=32
osc.myth-OST0001-osc.max_dirty_mb=32
osc.myth-OST0002-osc.max_dirty_mb=32
osc.myth-OST0003-osc.max_dirty_mb=32
osc.myth-OST0004-osc.max_dirty_mb=32
```

Use `-d` (only with -P) option to delete permanent parameter. Syntax:

```
lctl set_param -P -dobdtype.obdname.proc_file_name
```

For example:

```
# lctl set_param -P -d osc.*.max_dirty_mb
```

13.9.3.4. Listing Parameters

To list Lustre or LNET parameters that are available to set, use the `lctl list_param` command. For example:

```
lctl list_param [-FR] obdtype.obdname
```

The following arguments are available for the `lctl list_param` command.

`-F` Add '/', '@' or '=' for directories, symlinks and writeable files, respectively

`-R` Recursively lists all parameters under the specified path

For example:

```
oss# lctl list_param obdfilter.lustre-OST0000
```

13.9.3.5. Reporting Current Parameter Values

To report current Lustre parameter values, use the `lctl get_param` command with this syntax:

```
lctl get_param [-n] obdtype.obdname.proc_file_name
```

This example reports data on RPC service times.

```
oss# lctl get_param -n ost.*.ost_io.timeouts
service : cur 1 worst 30 (at 1257150393, 85d23h58m54s ago) 1 1 1 1
```

This example reports the amount of space this client has reserved for writeback cache with each OST:

```
client# lctl get_param osc.*.cur_grant_bytes
osc.myth-OST0000-osc-ffff8800376bdc00.cur_grant_bytes=2097152
osc.myth-OST0001-osc-ffff8800376bdc00.cur_grant_bytes=33890304
osc.myth-OST0002-osc-ffff8800376bdc00.cur_grant_bytes=35418112
osc.myth-OST0003-osc-ffff8800376bdc00.cur_grant_bytes=2097152
osc.myth-OST0004-osc-ffff8800376bdc00.cur_grant_bytes=33808384
```

13.10. Specifying NIDs and Failover

If a node has multiple network interfaces, it may have multiple NIDs, which must all be identified so other nodes can choose the NID that is appropriate for their network interfaces. Typically, NIDs are specified in a list delimited by commas (`,`). However, when failover nodes are specified, the NIDs are delimited by a colon (`:`) or by repeating a keyword such as `--mgsnode=` or `--servicenode=`).

To display the NIDs of all servers in networks configured to work with the Lustre file system, run (while LNET is running):

```
lctl list_nids
```

In the example below, `mds0` and `mds1` are configured as a combined MGS/MDT failover pair and `oss0` and `oss1` are configured as an OST failover pair. The Ethernet address for `mds0` is 192.168.10.1, and for `mds1` is 192.168.10.2. The Ethernet addresses for `oss0` and `oss1` are 192.168.10.20 and 192.168.10.21 respectively.

```
mds0# mkfs.lustre --fsname=testfs --mdt --mgs \
        --servicenode=192.168.10.2@tcp0 \
        -–servicenode=192.168.10.1@tcp0 /dev/sda1
mds0# mount -t lustre /dev/sda1 /mnt/test/mdt
oss0# mkfs.lustre --fsname=testfs --servicenode=192.168.10.20@tcp0 \
        --servicenode=192.168.10.21 --ost --index=0 \
        --mgsnode=192.168.10.1@tcp0 --mgsnode=192.168.10.2@tcp0 \
        /dev/sdb
oss0# mount -t lustre /dev/sdb /mnt/test/ost0
client# mount -t lustre 192.168.10.1@tcp0:192.168.10.2@tcp0:/testfs \
        /mnt/testfs
mds0# umount /mnt/mdt
mds1# mount -t lustre /dev/sda1 /mnt/test/mdt
mds1# cat /proc/fs/lustre/mds/testfs-MDT0000/recovery_status
```

Where multiple NIDs are specified separated by commas (for example, `10.67.73.200@tcp,192.168.10.1@tcp`), the two NIDs refer to the same host, and the Lustre software chooses the *best* one for communication. When a pair of NIDs is separated by a colon (for example, `10.67.73.200@tcp:10.67.73.201@tcp`), the two NIDs refer to two different hosts and are treated as a failover pair (the Lustre software tries the first one, and if that fails, it tries the second one.)

Two options to `mkfs.lustre` can be used to specify failover nodes. Introduced in Lustre software release 2.0, the `--servicenode` option is used to specify all service NIDs, including those for primary nodes and failover nodes. When the `--servicenode`option is used, the first service node to load the target device becomes the primary service node, while nodes corresponding to the other specified NIDs become failover locations for the target device. An older option, `--failnode`, specifies just the NIDS of failover nodes. For more information about the `--servicenode` and `--failnode` options, see Chapter 11, *Configuring Failover in a Lustre File System.*

13.11. Erasing a File System

If you want to erase a file system and permanently delete all the data in the file system, run this command on your targets:

```
$ "mkfs.lustre --reformat"
```

If you are using a separate MGS and want to keep other file systems defined on that MGS, then set the `writeconf` flag on the MDT for that file system. The `writeconf` flag causes the configuration logs to be erased; they are regenerated the next time the servers start.

To set the `writeconf` flag on the MDT:

1. Unmount all clients/servers using this file system, run:

   ```
   $ umount /mnt/lustre
   ```

2. Permanently erase the file system and, presumably, replace it with another file system, run:

   ```
   $ mkfs.lustre --reformat --fsname spfs --mgs --mdt --index=0 /dev/{mdsdev}
   ```

3. If you have a separate MGS (that you do not want to reformat), then add the `--writeconf` flag to `mkfs.lustre` on the MDT, run:

   ```
   $ mkfs.lustre --reformat --writeconf --fsname spfs --mgsnode=mgs_nid --mdt --ind
   ```

   Note

   If you have a combined MGS/MDT, reformatting the MDT reformats the MGS as well, causing all configuration information to be lost; you can start building your new file system. Nothing needs to be done with old disks that will not be part of the new file system, just do not mount them.

13.12. Reclaiming Reserved Disk Space

All current Lustre installations run the ldiskfs file system internally on service nodes. By default, ldiskfs reserves 5% of the disk space to avoid file system fragmentation. In order to reclaim this space, run the following command on your OSS for each OST in the file system:

```
tune2fs [-m reserved_blocks_percent] /dev/{ostdev}
```

You do not need to shut down Lustre before running this command or restart it afterwards.

Warning

Reducing the space reservation can cause severe performance degradation as the OST file system becomes more than 95% full, due to difficulty in locating large areas of contiguous free space. This performance degradation may persist even if the space usage drops below 95% again. It is recommended NOT to reduce the reserved disk space below 5%.

13.13. Replacing an Existing OST or MDT

To copy the contents of an existing OST to a new OST (or an old MDT to a new MDT), follow the process for either OST/MDT backups in Section 17.2, “Backing Up and Restoring an MDS or OST (Device Level)”

or Section 17.3, “Making a File-Level Backup of an OST or MDT File System”. For more information on removing a MDT, see Section 14.8.1, “Removing a MDT from the File System”.

13.14. Identifying To Which Lustre File an OST Object Belongs

Use this procedure to identify the file containing a given object on a given OST.

1. On the OST (as root), run `debugfs` to display the file identifier (`FID`) of the file associated with the object.

   For example, if the object is `34976` on `/dev/lustre/ost_test2`, the debug command is:

   ```
   # debugfs -c -R "stat /O/0/d$((34976 % 32))/34976" /dev/lustre/ost_test2
   ```

   The command output is:

   ```
   debugfs 1.42.3.wc3 (15-Aug-2012)
   /dev/lustre/ost_test2: catastrophic mode - not reading inode or group bitmaps
   Inode: 352365   Type: regular    Mode:  0666   Flags: 0x80000
   Generation: 2393149953    Version: 0x0000002a:00005f81
   User:  1000   Group:  1000   Size: 260096
   File ACL: 0    Directory ACL: 0
   Links: 1   Blockcount: 512
   Fragment:  Address: 0    Number: 0    Size: 0
   ctime: 0x4a216b48:00000000 -- Sat May 30 13:22:16 2009
   atime: 0x4a216b48:00000000 -- Sat May 30 13:22:16 2009
   mtime: 0x4a216b48:00000000 -- Sat May 30 13:22:16 2009
   crtime: 0x4a216b3c:975870dc -- Sat May 30 13:22:04 2009
   Size of extra inode fields: 24
   Extended attributes stored in inode body:
     fid = "b9 da 24 00 00 00 00 00 6a fa 0d 3f 01 00 00 00 eb 5b 0b 00 00 00 0000
     fid: objid=34976 seq=0 parent=[0x24dab9:0x3f0dfa6a:0x0] stripe=1
   EXTENTS:
   (0-64):4620544-4620607
   ```

2. For Lustre software release 2.x file systems, the parent FID will be of the form [0x200000400:0x122:0x0] and can be resolved directly using the `lfs fid2path [0x200000404:0x122:0x0] /mnt/lustre` command on any Lustre client, and the process is complete.

3. In this example the parent inode FID is an upgraded 1.x inode (due to the first part of the FID being below 0x200000400), the MDT inode number is `0x24dab9` and generation `0x3f0dfa6a` and the pathname needs to be resolved using `debugfs`.

4. On the MDS (as root), use `debugfs` to find the file associated with the inode:

   ```
   # debugfs -c -R "ncheck 0x24dab9" /dev/lustre/mdt_test
   ```

   Here is the command output:

   ```
   debugfs 1.42.3.wc2 (15-Aug-2012)
   /dev/lustre/mdt_test: catastrophic mode - not reading inode or group bitmap\
   s
   ```

```
Inode      Pathname
2415289    /ROOT/brian-laptop-guest/clients/client11/~dmtmp/PWRPNT/ZD16.BMP
```

The command lists the inode and pathname associated with the object.

Note

`Debugfs`' "ncheck" is a brute-force search that may take a long time to complete.

Note

To find the Lustre file from a disk LBA, follow the steps listed in the document at this URL: http://smartmontools.sourceforge.net/badblockhowto.html. Then, follow the steps above to resolve the Lustre filename.

Chapter 14. Lustre Maintenance

Once you have the Lustre file system up and running, you can use the procedures in this section to perform these basic Lustre maintenance tasks:

- Section 14.1, " Working with Inactive OSTs"
- Section 14.2, " Finding Nodes in the Lustre File System"
- Section 14.3, " Mounting a Server Without Lustre Service"
- Section 14.4, " Regenerating Lustre Configuration Logs"
- Section 14.5, " Changing a Server NID"
- Section 14.6, "Adding a New MDT to a Lustre File System"
- Section 14.7, " Adding a New OST to a Lustre File System"
- Section 14.8, " Removing and Restoring OSTs"
- Section 14.8.1, "Removing a MDT from the File System"
- Section 14.8.2, " Working with Inactive MDTs"\
- Section 14.8.3, " Removing an OST from the File System"
- Section 14.8.4, " Backing Up OST Configuration Files"
- Section 14.8.5, " Restoring OST Configuration Files"
- Section 14.8.6, " Returning a Deactivated OST to Service"
- Section 14.9, " Aborting Recovery"
- Section 14.10, " Determining Which Machine is Serving an OST "
- Section 14.11, " Changing the Address of a Failover Node"
- Section 14.12, " Separate a combined MGS/MDT"

14.1. Working with Inactive OSTs

To mount a client or an MDT with one or more inactive OSTs, run commands similar to this:

```
client# mount -o exclude=testfs-OST0000 -t lustre \
           uml1:/testfs /mnt/testfs
            client# cat /proc/fs/lustre/lov/testfs-clilov-*/target_obd
```

To activate an inactive OST on a live client or MDT, use the `lctl activate` command on the OSC device. For example:

```
lctl --device 7 activate
```

Note

A colon-separated list can also be specified. For example, `exclude=testfs-OST0000:testfs-OST0001`.

14.2. Finding Nodes in the Lustre File System

There may be situations in which you need to find all nodes in your Lustre file system or get the names of all OSTs.

To get a list of all Lustre nodes, run this command on the MGS:

```
# cat /proc/fs/lustre/mgs/MGS/live/*
```

Note

This command must be run on the MGS.

In this example, file system `testfs` has three nodes, `testfs-MDT0000`, `testfs-OST0000`, and `testfs-OST0001`.

```
cfs21:/tmp# cat /proc/fs/lustre/mgs/MGS/live/*
               fsname: testfs
               flags: 0x0     gen: 26
               testfs-MDT0000
               testfs-OST0000
               testfs-OST0001
```

To get the names of all OSTs, run this command on the MDS:

```
# cat /proc/fs/lustre/lov/fsname-mdtlov/target_obd
```

Note

This command must be run on the MDS.

In this example, there are two OSTs, testfs-OST0000 and testfs-OST0001, which are both active.

```
cfs21:/tmp# cat /proc/fs/lustre/lov/testfs-mdtlov/target_obd
0: testfs-OST0000_UUID ACTIVE
1: testfs-OST0001_UUID ACTIVE
```

14.3. Mounting a Server Without Lustre Service

If you are using a combined MGS/MDT, but you only want to start the MGS and not the MDT, run this command:

```
mount -t lustre /dev/mdt_partition -o nosvc /mount_point
```

The *mdt_partition* variable is the combined MGS/MDT block device.

In this example, the combined MGS/MDT is `testfs-MDT0000` and the mount point is `/mnt/test/mdt`.

```
$ mount -t lustre -L testfs-MDT0000 -o nosvc /mnt/test/mdt
```

14.4. Regenerating Lustre Configuration Logs

If the Lustre file system configuration logs are in a state where the file system cannot be started, use the `writeconf` command to erase them. After the `writeconf` command is run and the servers restart, the configuration logs are re-generated and stored on the MGS (as in a new file system).

You should only use the `writeconf` command if:

- The configuration logs are in a state where the file system cannot start
- A server NID is being changed

The `writeconf` command is destructive to some configuration items (i.e., OST pools information and items set via `conf_param`), and should be used with caution. To avoid problems:

- Shut down the file system before running the `writeconf` command
- Run the `writeconf` command on all servers (MDT first, then OSTs)
- Start the file system in this order:
  - MGS (or the combined MGS/MDT)
  - MDT
  - OSTs
  - Lustre clients

Caution

The OST pools feature enables a group of OSTs to be named for file striping purposes. If you use OST pools, be aware that running the `writeconf` command erases **all** pools information (as well as any other parameters set via `lctl conf_param`). We recommend that the pools definitions (and `conf_param` settings) be executed via a script, so they can be reproduced easily after a `writeconf` is performed.

To regenerate Lustre file system configuration logs:

1. Shut down the file system in this order.
   a. Unmount the clients.
   b. Unmount the MDT.
   c. Unmount all OSTs.
2. Make sure the the MDT and OST devices are available.
3. Run the `writeconf` command on all servers.

Run writeconf on the MDT first, and then the OSTs.

a. On the MDT, run:

```
mdt# tunefs.lustre --writeconf /dev/mdt_device
```

b. On each OST, run:

```
ost# tunefs.lustre --writeconf /dev/ost_device
```

4. Restart the file system in this order.

a. Mount the MGS (or the combined MGS/MDT).

b. Mount the MDT.

c. Mount the OSTs.

d. Mount the clients.

After the `writeconf` command is run, the configuration logs are re-generated as servers restart.

14.5. Changing a Server NID

In Lustre software release 2.3 or earlier, the `tunefs.lustre --writeconf` command is used to rewrite all of the configuration files.

Introduced in Lustre 2.4

If you need to change the NID on the MDT or OST, a new `replace_nids` command was added in Lustre software release 2.4 to simplify this process. The `replace_nids` command differs from `tunefs.lustre --writeconf` in that it does not erase the entire configuration log, precluding the need the need to execute the `writeconf` command on all servers and re-specify all permanent parameter settings. However, the `writeconf` command can still be used if desired.

Change a server NID in these situations:

- New server hardware is added to the file system, and the MDS or an OSS is being moved to the new machine.
- New network card is installed in the server.
- You want to reassign IP addresses.

To change a server NID:

1. Update the LNET configuration in the `/etc/modprobe.conf` file so the list of server NIDs is correct. Use `lctl list_nids` to view the list of server NIDS.

   The `lctl list_nids` command indicates which network(s) are configured to work with the Lustre file system.

2. Shut down the file system in this order:

a. Unmount the clients.

b. Unmount the MDT.

c. Unmount all OSTs.

3. If the MGS and MDS share a partition, start the MGS only:

   ```
   mount -t lustre MDT partition -o nosvc mount_point
   ```

4. Run the `replace_nids` command on the MGS:

   ```
   lctl replace_nids devicename nid1[,nid2,nid3 ...]
   ```

   where *devicename* is the Lustre target name, e.g. `testfs-OST0013`

5. If the MGS and MDS share a partition, stop the MGS:

   ```
   umount mount_point
   ```

Note

The `replace_nids` command also cleans all old, invalidated records out of the configuration log, while preserving all other current settings.

Note

The previous configuration log is backed up on the MGS disk with the suffix `'.bak'`.

Introduced in Lustre 2.4

Additional MDTs can be added to serve one or more remote sub-directories within the file system. It is possible to have multiple remote sub-directories reference the same MDT. However, the root directory will always be located on MDT0. To add a new MDT into the file system:

1. Discover the maximum MDT index. Each MDTs must have unique index.

   ```
   client$ lctl dl | grep mdc
   36 UP mdc testfs-MDT0000-mdc-ffff88004edf3c00 4c8be054-144f-9359-b063-8477566eb8
   37 UP mdc testfs-MDT0001-mdc-ffff88004edf3c00 4c8be054-144f-9359-b063-8477566eb8
   38 UP mdc testfs-MDT0002-mdc-ffff88004edf3c00 4c8be054-144f-9359-b063-8477566eb8
   39 UP mdc testfs-MDT0003-mdc-ffff88004edf3c00 4c8be054-144f-9359-b063-8477566eb8
   ```

2. Add the new block device as a new MDT at the next available index. In this example, the next available index is 4.

   ```
   mds# mkfs.lustre --reformat --fsname=filesystem_name --mdt --mgsnode=mgsnode --i
   ```

3. Mount the MDTs.

   ```
   mds# mount –t lustre /dev/mdt4_blockdevice /mnt/mdt4
   ```

14.7. Adding a New OST to a Lustre File System

To add an OST to existing Lustre file system:

1. Add a new OST by passing on the following commands, run:

```
oss# mkfs.lustre --fsname=spfs --mgsnode=mds16@tcp0 --ost --index=12 /dev/sda
oss# mkdir -p /mnt/test/ost12
oss# mount -t lustre /dev/sda /mnt/test/ost12
```

2. Migrate the data (possibly).

   The file system is quite unbalanced when new empty OSTs are added. New file creations are automatically balanced. If this is a scratch file system or files are pruned at a regular interval, then no further work may be needed.

   New files being created will preferentially be placed on the empty OST. As old files are deleted, they will release space on the old OST.

   Files existing prior to the expansion can optionally be rebalanced with an in-place copy, which can be done with a simple script. The basic method is to copy existing files to a temporary file, then move the temp file over the old one. This should not be attempted with files which are currently being written to by users or applications. This operation redistributes the stripes over the entire set of OSTs.

   For example, to rebalance all files within `/mnt/lustre/dir`, enter:

```
client# lfs_migrate /mnt/lustre/file
```

   To migrate files within the `/test` file system on `OST0004` that are larger than 4GB in size, enter:

```
client# lfs find /test -obd test-OST0004 -size +4G | lfs_migrate -y
```

   See Section 33.2, “ `lfs_migrate` ” for more details.

14.8. Removing and Restoring OSTs

OSTs can be removed from and restored to a Lustre file system. Removing a OST means the OST is *deactivated* in the file system, not permanently removed.

Note

A removed OST still appears in the file system; do not create a new OST with the same name.

You may want to remove (deactivate) an OST and prevent new files from being written to it in several situations:

- Hard drive has failed and a RAID resync/rebuild is underway
- OST is nearing its space capacity
- OST storage has failed permanently

Introduced in Lustre 2.4

If the MDT is permanently inaccessible, `lfs rmdir {directory}` can be used to delete the directory entry. A normal `rmdir` will report an IO error due to the remote MDT being inactive. After the remote directory has been removed, the administrator should mark the MDT as permanently inactive with:

```
lctl conf_param {MDT name}.mdc.active=0
```

A user can identify the location of a remote sub-directory using the `lfs` utility. For example:

```
client$ lfs getstripe -M /mnt/lustre/remote_dir1
1
client$ mkdir /mnt/lustre/local_dir0
client$ lfs getstripe -M /mnt/lustre/local_dir0
0
```

The `getstripe -M` parameters return the index of the MDT that is serving the given directory.

Introduced in Lustre 2.4

Files located on or below an inactive MDT are inaccessible until the MDT is activated again. Clients accessing an inactive MDT will receive an EIO error.

14.8.3. Removing an OST from the File System

When removing an OST, remember that the MDT does not communicate directly with OSTs. Rather, each OST has a corresponding OSC which communicates with the MDT. It is necessary to determine the device number of the OSC that corresponds to the OST. Then, you use this device number to deactivate the OSC on the MDT.

To remove an OST from the file system:

1. For the OST to be removed, determine the device number of the corresponding OSC on the MDT.

   a. List all OSCs on the node, along with their device numbers. Run:

      ```
      lctl dl | grep osc
      ```

      For example: `lctl dl | grep`

      ```
      11 UP osc testfs-OST-0000-osc-cac94211 4ea5b30f-6a8e-55a0-7519-2f20318ebdb4 5
      12 UP osc testfs-OST-0001-osc-cac94211 4ea5b30f-6a8e-55a0-7519-2f20318ebdb4 5
      13 IN osc testfs-OST-0000-osc testfs-MDT0000-mdtlov_UUID 5
      14 UP osc testfs-OST-0001-osc testfs-MDT0000-mdtlov_UUID 5
      ```

   b. Determine the device number of the OSC that corresponds to the OST to be removed.

2. Temporarily deactivate the OSC on the MDT. On the MDT, run:

   ```
   mds# lctl --device lustre_devno deactivate
   ```

   For example, based on the command output in Step 1, to deactivate device 13 (the MDT's OSC for `OST-0000`), the command would be:

   ```
   mds# lctl --device 13 deactivate
   ```

   This marks the OST as inactive on the MDS, so no new objects are assigned to the OST. This does not prevent use of existing objects for reads or writes.

Note

Do not deactivate the OST on the clients. Do so causes errors (EIOs), and the copy out to fail.

Caution

Do not use `lctl conf_param` to deactivate the OST. It permanently sets a parameter in the file system configuration.

3. Discover all files that have objects residing on the deactivated OST.

   Depending on whether the deactivated OST is available or not, the data from that OST may be migrated to other OSTs, or may need to be restored from backup.

   a. If the OST is still online and available, find all files with objects on the deactivated OST, and copy them to other OSTs in the file system to:

   ```
   client# lfs find --obd ost_name /mount/point | lfs_migrate -y
   ```

   b. If the OST is no longer available, delete the files on that OST and restore them from backup:

   ```
   client# lfs find --obd ost_uuid -print0 /mount/point | \
             tee /tmp/files_to_restore | xargs -0 -n 1 unlink
   ```

   The list of files that need to be restored from backup is stored in `/tmp/files_to_restore`. Restoring these files is beyond the scope of this document.

4. Deactivate the OST.

   a. To temporarily disable the deactivated OST, enter:

   ```
   [client]# lctl set_param osc.fsname-OSTnumber-*.active=0
   ```

   If there is expected to be a replacement OST in some short time (a few days), the OST can temporarily be deactivated on the clients:

   ### Note

   This setting is only temporary and will be reset if the clients or MDS are rebooted. It needs to be run on all clients.

   If there is not expected to be a replacement for this OST in the near future, permanently deactivate the OST on all clients and the MDS:

   ```
   [mgs]# lctl conf_param ost_name.osc.active=0
   ```

   ### Note

   A removed OST still appears in the file system; do not create a new OST with the same name.

14.8.4. Backing Up OST Configuration Files

If the OST device is still accessible, then the Lustre configuration files on the OST should be backed up and saved for future use in order to avoid difficulties when a replacement OST is returned to service. These

files rarely change, so they can and should be backed up while the OST is functional and accessible. If the deactivated OST is still available to mount (i.e. has not permanently failed or is unmountable due to severe corruption), an effort should be made to preserve these files.

1. Mount the OST file system.

```
oss# mkdir -p /mnt/ost
[oss]# mount -t ldiskfs /dev/ost_device /mnt/ost
```

2. Back up the OST configuration files.

```
oss# tar cvf ost_name.tar -C /mnt/ost last_rcvd \
          CONFIGS/ O/0/LAST_ID
```

3. Unmount the OST file system.

```
oss# umount /mnt/ost
```

14.8.5. Restoring OST Configuration Files

If the original OST is still available, it is best to follow the OST backup and restore procedure given in either Section 17.2, “Backing Up and Restoring an MDS or OST (Device Level)”, or Section 17.3, “Making a File-Level Backup of an OST or MDT File System” and Section 17.4, “Restoring a File-Level Backup”.

To replace an OST that was removed from service due to corruption or hardware failure, the file system needs to be formatted using `mkfs.lustre`, and the Lustre file system configuration should be restored, if available.

If the OST configuration files were not backed up, due to the OST file system being completely inaccessible, it is still possible to replace the failed OST with a new one at the same OST index.

1. Format the OST file system.

```
oss# mkfs.lustre --ost --index=old_ost_index other_options \
          /dev/new_ost_dev
```

2. Mount the OST file system.

```
oss# mkdir /mnt/ost
oss# mount -t ldiskfs /dev/new_ost_dev /mnt/ost
```

3. Restore the OST configuration files, if available.

```
oss# tar xvf ost_name.tar -C /mnt/ost
```

4. Recreate the OST configuration files, if unavailable.

   Follow the procedure in Section 27.3.4, “Fixing a Bad LAST_ID on an OST” to recreate the LAST_ID file for this OST index. The `last_rcvd` file will be recreated when the OST is first mounted using the default parameters, which are normally correct for all file systems. The `CONFIGS/mountdata` file is created by `mkfs.lustre` at format time, but has flags set that request it to register itself with the MGS. It is possible to copy these flags from another working OST (which should be the same):

```
oss1# debugfs -c -R "dump CONFIGS/mountdata /tmp/ldd" /dev/other_osdev
oss1# scp /tmp/ldd oss0:/tmp/ldd
```

```
oss0# dd if=/tmp/ldd of=/mnt/ost/CONFIGS/mountdata bs=4 count=1 seek=5 skip=5 co
```

5. Unmount the OST file system.

```
oss# umount /mnt/ost
```

14.8.6. Returning a Deactivated OST to Service

If the OST was permanently deactivated, it needs to be reactivated in the MGS configuration.

```
mgs# lctl conf_param ost_name.osc.active=1
```

If the OST was temporarily deactivated, it needs to be reactivated on the MDS and clients.

```
mds# lctl --device lustre_devno activate
                    client# lctl set_param osc.fsname-OSTnumber-*.active=0
```

14.9. Aborting Recovery

You can abort recovery with either the `lctl` utility or by mounting the target with the `abort_recov` option (`mount -o abort_recov`). When starting a target, run:

```
mds# mount -t lustre -L mdt_name -o abort_recov /mount_point
```

Note

The recovery process is blocked until all OSTs are available.

14.10. Determining Which Machine is Serving an OST

In the course of administering a Lustre file system, you may need to determine which machine is serving a specific OST. It is not as simple as identifying the machine's IP address, as IP is only one of several networking protocols that the Lustre software uses and, as such, LNET does not use IP addresses as node identifiers, but NIDs instead. To identify the NID that is serving a specific OST, run one of the following commands on a client (you do not need to be a root user):

```
client$ lctl get_param osc.fsname-OSTnumber*.ost_conn_uuid
```

For example:

```
client$ lctl get_param osc.*-OST0000*.ost_conn_uuid
osc.testfs-OST0000-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
```

- OR -

```
client$ lctl get_param osc.*.ost_conn_uuid
osc.testfs-OST0000-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.testfs-OST0001-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.testfs-OST0002-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.testfs-OST0003-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.testfs-OST0004-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
```

14.11. Changing the Address of a Failover Node

To change the address of a failover node (e.g, to use node X instead of node Y), run this command on the OSS/OST partition (depending on which option was used to originally identify the NID):

```
oss# tunefs.lustre --erase-params --servicenode=NID /dev/ost_device
```

or

```
oss# tunefs.lustre --erase-params --failnode=NID /dev/ost_device
```

For more information about the `--servicenode` and `--failnode` options, see Chapter 11, *Configuring Failover in a Lustre File System*.

14.12. Separate a combined MGS/MDT

These instructions assume the MGS node will be the same as the MDS node. For instructions on how to move MGS to a different node, see Section 14.5, " Changing a Server NID".

These instructions are for doing the split without shutting down other servers and clients.

1. Stop the MDS.

   Unmount the MDT

   ```
   umount -f /dev/mdt_device
   ```

2. Create the MGS.

   ```
   mds# mkfs.lustre --mgs --device-size=size /dev/mgs_device
   ```

3. Copy the configuration data from MDT disk to the new MGS disk.

   ```
   mds# mount -t ldiskfs -o ro /dev/mdt_device /mdt_mount_point
   mds# mount -t ldiskfs -o rw /dev/mgs_device /mgs_mount_point
   mds# cp -r /mdt_mount_point/CONFIGS/filesystem_name-* /mgs_mount_point/CONFIGS/.
   mds# umount /mgs_mount_point
   mds# umount /mdt_mount_point
   ```

   See Section 14.4, " Regenerating Lustre Configuration Logs" for alternative method.

4. Start the MGS.

   ```
   mgs# mount -t lustre /dev/mgs_device /mgs_mount_point
   ```

   Check to make sure it knows about all your file system

   ```
   cat /proc/fs/lustre/mgs/MGS/filesystems
   ```

5. Remove the MGS option from the MDT, and set the new MGS nid.

```
mds# tunefs.lustre --nomgs --mgsnode=new_mgs_nid /dev/mdt-device
```

6. Start the MDT.

   ```
   mds# mount -t lustre /dev/mdt_device /mdt_mount_point
   ```

   Check to make sure the MGS configuration look right

   ```
   mds# cat /proc/fs/lustre/mgs/MGS/live/filesystem_name
   ```

Chapter 15. Managing Lustre Networking (LNET)

This chapter describes some tools for managing Lustre networking (LNET) and includes the following sections:

- Section 15.1, " Updating the Health Status of a Peer or Router"
- Section 15.2, "Starting and Stopping LNET"
- Section 15.3, "Multi-Rail Configurations with LNET"
- Section 15.4, "Load Balancing with an InfiniBand* Network"
- Section 15.5, "Dynamically Configuring LNET Routes"

15.1. Updating the Health Status of a Peer or Router

There are two mechanisms to update the health status of a peer or a router:

- LNET can actively check health status of all routers and mark them as dead or alive automatically. By default, this is off. To enable it set `auto_down` and if desired `check_routers_before_use`. This initial check may cause a pause equal to `router_ping_timeout` at system startup, if there are dead routers in the system.
- When there is a communication error, all LNDs notify LNET that the peer (not necessarily a router) is down. This mechanism is always on, and there is no parameter to turn it off. However, if you set the LNET module parameter `auto_down` to `0`, LNET ignores all such peer-down notifications.

Several key differences in both mechanisms:

- The router pinger only checks routers for their health, while LNDs notices all dead peers, regardless of whether they are a router or not.
- The router pinger actively checks the router health by sending pings, but LNDs only notice a dead peer when there is network traffic going on.
- The router pinger can bring a router from alive to dead or vice versa, but LNDs can only bring a peer down.

15.2. Starting and Stopping LNET

The Lustre software automatically starts and stops LNET, but it can also be manually started in a standalone manner. This is particularly useful to verify that your networking setup is working correctly before you attempt to start the Lustre file system.

15.2.1. Starting LNET

To start LNET, run:

```
$ modprobe lnet
$ lctl network up
```

To see the list of local NIDs, run:

```
$ lctl list_nids
```

This command tells you the network(s) configured to work with the Lustre file system.

If the networks are not correctly setup, see the `modules.conf` "`networks=`" line and make sure the network layer modules are correctly installed and configured.

To get the best remote NID, run:

```
$ lctl which_nid NIDs
```

where *`NIDs`* is the list of available NIDs.

This command takes the "best" NID from a list of the NIDs of a remote host. The "best" NID is the one that the local node uses when trying to communicate with the remote node.

15.2.1.1. Starting Clients

To start a TCP client, run:

```
mount -t lustre mdsnode:/mdsA/client /mnt/lustre/
```

To start an Elan client, run:

```
mount -t lustre 2@elan0:/mdsA/client /mnt/lustre
```

15.2.2. Stopping LNET

Before the LNET modules can be removed, LNET references must be removed. In general, these references are removed automatically when the Lustre file system is shut down, but for standalone routers, an explicit step is needed to stop LNET. Run:

```
lctl network unconfigure
```

> **Note**
>
> Attempting to remove Lustre modules prior to stopping the network may result in a crash or an LNET hang. If this occurs, the node must be rebooted (in most cases). Make sure that the Lustre network and Lustre file system are stopped prior to unloading the modules. Be extremely careful using `rmmod -f`.

To unconfigure the LNET network, run:

```
modprobe -r lnd_and_lnet_modules
```

> **Note**
>
> To remove all Lustre modules, run:
>
> ```
> $ lustre_rmmod
> ```

15.3. Multi-Rail Configurations with LNET

To aggregate bandwidth across both rails of a dual-rail IB cluster (o2iblnd) [1] using LNET, consider these points:

- LNET can work with multiple rails, however, it does not load balance across them. The actual rail used for any communication is determined by the peer NID.
- Multi-rail LNET configurations do not provide an additional level of network fault tolerance. The configurations described below are for bandwidth aggregation only.
- A Lustre node always uses the same local NID to communicate with a given peer NID. The criteria used to determine the local NID are:
  - Introduced in Lustre 2.5

    Lowest route priority number (lower number, higher priority).
  - Fewest hops (to minimize routing), and
  - Appears first in the "`networks`" or "`ip2nets`" LNET configuration strings

15.4. Load Balancing with an InfiniBand* Network

A Lustre file system contains OSSs with two InfiniBand HCAs. Lustre clients have only one InfiniBand HCA using OFED-based Infiniband "o2ib" drivers. Load balancing between the HCAs on the OSS is accomplished through LNET.

15.4.1. Setting Up `lustre.conf` for Load Balancing

To configure LNET for load balancing on clients and servers:

1. Set the `lustre.conf` options.

   Depending on your configuration, set `lustre.conf` options as follows:

   - Dual HCA OSS server

   ```
   options lnet networks="o2ib0(ib0),o2ib1(ib1) 192.168.10.1.[101-102]
   ```

   - Client with the odd IP address

   ```
   options lnet networks=o2ib0(ib0) 192.168.10.[103-253/2]
   ```

   - Client with the even IP address

   ```
   options lnet networks=o2ib1(ib0) 192.168.10.[102-254/2]
   ```

2. Run the modprobe lnet command and create a combined MGS/MDT file system.

   The following commands create an MGS/MDT or OST file system and mount the targets on the servers.

[1] Multi-rail configurations are only supported by o2iblnd; other IB LNDs do not support multiple interfaces.

```
modprobe lnet
# mkfs.lustre --fsname lustre --mgs --mdt /dev/mdt_device
# mkdir -p /mount_point
# mount -t lustre /dev/mdt_device /mount_point
```

For example:

```
modprobe lnet
mds# mkfs.lustre --fsname lustre --mdt --mgs /dev/sda
mds# mkdir -p /mnt/test/mdt
mds# mount -t lustre /dev/sda /mnt/test/mdt
mds# mount -t lustre mgs@o2ib0:/lustre /mnt/mdt
oss# mkfs.lustre --fsname lustre --mgsnode=mds@o2ib0 --ost --index=0 /dev/sda
oss# mkdir -p /mnt/test/mdt
oss# mount -t lustre /dev/sda /mnt/test/ost
oss# mount -t lustre mgs@o2ib0:/lustre /mnt/ost0
```

3. Mount the clients.

```
client# mount -t lustre mgs_node:/fsname /mount_point
```

This example shows an IB client being mounted.

```
client# mount -t lustre
192.168.10.101@o2ib0,192.168.10.102@o2ib1:/mds/client /mnt/lustre
```

As an example, consider a two-rail IB cluster running the OFED stack with these IPoIB address assignments.

```
             ib0                              ib1
Servers             192.168.0.*                      192.168.1.*
Clients             192.168.[2-127].*                192.168.[128-253].*
```

You could create these configurations:

- A cluster with more clients than servers. The fact that an individual client cannot get two rails of bandwidth is unimportant because the servers are typically the actual bottleneck.

```
ip2nets="o2ib0(ib0),    o2ib1(ib1)      192.168.[0-1].*                     \
                                            #all servers;\
                  o2ib0(ib0)      192.168.[2-253].[0-252/2]       #even cl\
ients;\
                  o2ib1(ib1)      192.168.[2-253].[1-253/2]       #odd cli\
ents"
```

This configuration gives every server two NIDs, one on each network, and statically load-balances clients between the rails.

- A single client that must get two rails of bandwidth, and it does not matter if the maximum aggregate bandwidth is only (# servers) * (1 rail).

```
ip2nets="       o2ib0(ib0)                          192.168.[0-1].[0-252/2]     \
                                                  #even servers;\
           o2ib1(ib1)                           192.168.[0-1].[1-253/2]         \
                                              #odd servers;\
```

```
          o2ib0(ib0),o2ib1(ib1)           192.168.[2-253].*               \
                                        #clients"
```

This configuration gives every server a single NID on one rail or the other. Clients have a NID on both rails.

- All clients and all servers must get two rails of bandwidth.

```
ip2nets=â€   o2ib0(ib0),o2ib2(ib1)           192.168.[0-1].[0-252/2]       \
  #even servers;\
          o2ib1(ib0),o2ib3(ib1)           192.168.[0-1].[1-253/2]         \
#odd servers;\
          o2ib0(ib0),o2ib3(ib1)           192.168.[2-253].[0-252/2)       \
#even clients;\
          o2ib1(ib0),o2ib2(ib1)           192.168.[2-253].[1-253/2)       \
#odd clients"
```

This configuration includes two additional proxy o2ib networks to work around the simplistic NID selection algorithm in the Lustre software. It connects "even" clients to "even" servers with `o2ib0` on `rail0`, and "odd" servers with `o2ib3` on `rail1`. Similarly, it connects "odd" clients to "odd" servers with `o2ib1` on `rail0`, and "even" servers with `o2ib2` on `rail1`.

Introduced in Lustre 2.4

Two scripts are provided: `lustre/scripts/lustre_routes_config` and `lustre/scripts/lustre_routes_conversion`.

`lustre_routes_config` sets or cleans up LNET routes from the specified config file. `/etc/sysconfig/lustre_routes.conf` file can be used to automatically configure routes on LNET startup.

`lustre_routes_conversion` converts a legacy routes configuration file to the new syntax, which is parsed by `lustre_routes_config`.

15.5.1. `lustre_routes_config`

`lustre_routes_config` usage is as follows

```
lustre_routes_config [--setup|--cleanup|--dry-run|--verbose] config_file
         --setup: configure routes listed in config_file
         --cleanup: unconfigure routes listed in config_file
         --dry-run: echo commands to be run, but do not execute them
         --verbose: echo commands before they are executed
```

The format of the file which is passed into the script is as follows:

```
network: { gateway: gateway@exit_network [hop: hop] [priority: priority] }
```

An LNET router is identified when its local NID appears within the list of routes. However, this can not be achieved by the use of this script, since the script only adds extra routes after the router is identified. To ensure that a router is identified correctly, make sure to add its local NID in the routes parameter in the modprobe lustre configuration file. See Section 36.1, “ Introduction”.

15.5.2. `lustre_routes_conversion`

`lustre_routes_conversion` usage is as follows:

```
lustre_routes_conversion legacy_file new_file
```

`lustre_routes_conversion` takes as a first parameter a file with routes configured as follows:

```
network [hop] gateway@exit network[:priority];
```

The script then converts each routes entry in the provided file to:

```
network: { gateway: gateway@exit network [hop: hop] [priority: priority] }
```

and appends each converted entry to the output file passed in as the second parameter to the script.

15.5.3. Route Configuration Examples

Below is an example of a legacy LNET route configuration. A legacy configuration file can have multiple entries.

```
tcp1 10.1.1.2@tcp0:1;
tcp2 10.1.1.3@tcp0:2;
tcp3 10.1.1.4@tcp0;
```

Below is an example of the converted LNET route configuration. The following would be the result of the `lustre_routes_conversion` script, when run on the above legacy entries.

```
tcp1: { gateway: 10.1.1.2@tcp0 priority: 1 }
tcp2: { gateway: 10.1.1.2@tcp0 priority: 2 }
tcp1: { gateway: 10.1.1.4@tcp0 }
```

Chapter 16. Upgrading a Lustre File System

This chapter describes interoperability between Lustre software releases. It also provides procedures for upgrading from Lustre software release 1.8 to Lustre softeware release 2.x , from a Lustre software release 2.x to a more recent Lustre software release 2.x (major release upgrade), and from a a Lustre software release 2.x.y to a more recent Lustre software release 2.x.y (minor release upgrade). It includes the following sections:

- Section 16.1, “Release Interoperability and Upgrade Requirements”
- Section 16.2, “Upgrading to Lustre Software Release 2.x (Major Release)”
- Section 16.3, “Upgrading to Lustre Software Release 2.x.y (Minor Release)”

16.1. Release Interoperability and Upgrade Requirements

Lustre software release 2.x (major) upgrade:

- All servers must be upgraded at the same time, while some or all clients may be upgraded.
- All servers must be be upgraded to a Linux kernel supported by the Lustre software. See the Linux Test Matrix at Table 8.1, “Lustre Test Matrix” for a list of tested Lustre distributions.
- Clients to be upgraded to the Lustre software release 2.4 or higher must be running a compatible Linux distribution. See the Linux Test Matrix at Table 8.1, “Lustre Test Matrix” for a list of tested Linux distributions.

Lustre software release 2.x.y release (minor) upgrade:

- All servers must be upgraded at the same time, while some or all clients may be upgraded.
- Rolling upgrades are supported for minor releases allowing individual servers and clients to be upgraded without stopping the Lustre file system.

16.2. Upgrading to Lustre Software Release 2.x (Major Release)

The procedure for upgrading from a Lustre software release 2.x to a more recent 2.x release of the Lustre software is described in this section.

Note

This procedure can also be used to upgrade Lustre software release 1.8.6-wc1 or later to any Lustre software release 2.x. To upgrade other versions of Lustre software release 1.8.x, contact your support provider.

Note

Introduced in Lustre 2.2

In Lustre software release 2.2, a feature has been added that allows striping across up to 2000 OSTs. By default, this "wide striping" feature is disabled. It is activated by setting the `large_xattr` or `ea_inode` option on the MDT using either `mkfs.lustre` or `tune2fs`. For example after upgrading an existing file system to Lustre software release 2.2 or later, wide striping can be enabled by running the following command on the MDT device before mounting it:

```
tune2fs -O large_xattr
```

Once the wide striping feature is enabled and in use on the MDT, it is not possible to directly downgrade the MDT file system to an earlier version of the Lustre software that does not support wide striping. To disable wide striping:

1. Delete all wide-striped files.

   OR

   Use `lfs_migrate` with the option `-c` *`stripe_count`* (set *`stripe_count`* to 160) to move the files to another location.

2. Unmount the MDT.

3. Run the following command to turn off the `large_xattr` option:

   ```
   tune2fs -O ^large_xattr
   ```

Using either `mkfs.lustre` or `tune2fs` with `large_xattr` or `ea_inode` option reseults in `ea_inode` in the file system feature list.

Introduced in Lustre 2.3

To generate a list of all files with more than 160 stripes use `lfs find` with the `--stripe-count` option:

```
lfs find ${mountpoint} --stripe-count=+160
```

Introduced in Lustre 2.4

In Lustre software release 2.4, a new feature allows using multiple MDTs, which can each serve one or more remote sub-directories in the file system. The `root` directory is always located on MDT0.

Note that clients running a release prior to the Lustre software release 2.4 can only see the namespace hosted by MDT0 and will return an IO error if an attempt is made to access a directory on another MDT.

To upgrade a Lustre software release 2.x to a more recent major release, complete these steps:

1. Create a complete, restorable file system backup.

Caution

Before installing the Lustre software, back up ALL data. The Lustre software contains kernel modifications that interact with storage devices and may introduce security issues and data loss if not installed, configured, or administered properly. If a full backup of the file system is

not practical, a device-level backup of the MDT file system is recommended. See Chapter 17, *Backing Up and Restoring a File System* for a procedure.

2. Shut down the file system by unmounting all clients and servers in the order shown below (unmounting a block device causes the Lustre software to be shut down on that node):

   a. Unmount the clients. On each client node, run:

   ```
   umount -a -t lustre
   ```

   b. Unmount the MDT. On the MDS node, run:

   ```
   umount -a -t lustre
   ```

   c. Unmount all the OSTs. On each OSS node, run:

   ```
   umount -a -t lustre
   ```

3. Upgrade the Linux operating system on all servers to a compatible (tested) Linux distribution and reboot. See the Linux Test Matrix at Table 8.1, “Lustre Test Matrix”.

4. Upgrade the Linux operating system on all clients to Red Hat Enterprise Linux 6 or other compatible (tested) distribution and reboot. See the Linux Test Matrix at Table 8.1, “Lustre Test Matrix”.

5. Download the Lustre server RPMs for your platform from the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository. See Table 8.2, “Packages Installed on Lustre Servers” for a list of required packages.

6. Install the Lustre server packages on all Lustre servers (MGS, MDSs, and OSSs).

   a. Log onto a Lustre server as the `root` user

   b. Use the `yum` command to install the packages:

   ```
   # yum --nogpgcheck install pkg1.rpm pkg2.rpm ...
   ```

   c. Verify the packages are installed correctly:

   ```
   rpm -qa|egrep "lustre|wc"
   ```

   d. Repeat these steps on each Lustre server.

7. Download the Lustre client RPMs for your platform from the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository. See Table 8.3, “Packages Installed on Lustre Clients” for a list of required packages.

   ### Note

   The version of the kernel running on a Lustre client must be the same as the version of the `lustre-client-modules-`*`ver`* package being installed. If not, a compatible kernel must be installed on the client before the Lustre client packages are installed.

8. Install the Lustre client packages on each of the Lustre clients to be upgraded.

   a. Log onto a Lustre client as the `root` user.

   b. Use the `yum` command to install the packages:

```
# yum --nogpgcheck install pkg1.rpm pkg2.rpm ...
```

c. Verify the packages were installed correctly:

```
# rpm -qa|egrep "lustre|kernel"
```

d. Repeat these steps on each Lustre client.

9. (Optional) For upgrades to Lustre software release 2.2 or higher, to enable wide striping on an existing MDT, run the following command on the MDT :

```
mdt# tune2fs -O large_xattr device
```

For more information about wide striping, see Section 18.6, “Lustre Striping Internals”.

10. (Optional) For upgrades to Lustre software release 2.4 or higher, to format an additional MDT, complete these steps:

a. Determine the index used for the first MDT (each MDT must have unique index). Enter:

```
client$ lctl dl | grep mdc
36 UP mdc lustre-MDT0000-mdc-ffff88004edf3c00
      4c8be054-144f-9359-b063-8477566eb84e 5
```

In this example, the next available index is 1.

b. Add the new block device as a new MDT at the next available index by entering (on one line):

```
mds# mkfs.lustre --reformat --fsname=filesystem_name --mdt \
    --mgsnode=mgsnode --index 1 /dev/mdt1_device
```

11. (Optional) If you are upgrading to Lustre software release 2.3 or higher from Lustre software release 2.2 or earlier and want to enable the quota feature, complete these steps:

a. Before setting up the file system, enter on both the MDS and OSTs:

```
tunefs.lustre --quota
```

b. When setting up the file system, enter:

```
conf_param $FSNAME.quota.mdt=$QUOTA_TYPE
conf_param $FSNAME.quota.ost=$QUOTA_TYPE
```

12. (Optional) If you are upgrading from Lustre software release 1.8, you must manually enable the FID-in-dirent feature. On the MDS, enter:

```
tune2fs –O dirdata /dev/mdtdev
```

Warning

This step is not reversible. Do not complete this step until you are sure you will not be downgrading the Lustre software.

This step only enables FID-in-dirent for newly created files. If you are upgrading to Lustre software release 2.4, you can use LFSCK 1.5 to enable FID-in-dirent for existing files. For more information

about FID-in-dirent and related functionalities in LFSCK 1.5, see Section 1.3, “ Lustre File System Storage and I/O”.

13. Start the Lustre file system by starting the components in the order shown in the following steps:

    a. Mount the MGT. On the MGS, run

    ```
    mgs# mount -a -t lustre
    ```

    b. Mount the MDT(s). On each MDT, run:

    ```
    mds# mount -a -t lustre
    ```

    c. Mount all the OSTs. On each OSS node, run:

    ```
    oss# mount -a -t lustre
    ```

    Note

    This command assumes that all the OSTs are listed in the `/etc/fstab` file. OSTs that are not listed in the `/etc/fstab` file, must be mounted individually by running the mount command:

    ```
    mount -t lustre /dev/block_device /mount_point
    ```

    d. Mount the file system on the clients. On each client node, run:

    ```
    client# mount -a -t lustre
    ```

Note

The mounting order described in the steps above must be followed for the intial mount and registration of a Lustre file system after an upgrade. For a normal start of a Lustre file system, the mounting order is MGT, OSTs, MDT(s), clients.

If you have a problem upgrading a Lustre file system, see Section 27.2, “ Reporting a Lustre File System Bug” for some ways to get help.

16.3. Upgrading to Lustre Software Release 2.x.y (Minor Release)

Rolling upgrades are supported for upgrading from any Lustre software release 2.x.y to a more recent Lustre software release 2.X.y. This allows the Lustre file system to continue to run while individual servers (or their failover partners) and clients are upgraded one at a time. The procedure for upgrading a Lustre software release 2.x.y to a more recent minor release is described in this section.

To upgrade Lustre software release 2.x.y to a more recent minor release, complete these steps:

1. Create a complete, restorable file system backup.

   Caution

   Before installing the Lustre software, back up ALL data. The Lustre software contains kernel modifications that interact with storage devices and may introduce security issues and data

loss if not installed, configured, or administered properly. If a full backup of the file system is not practical, a device-level backup of the MDT file system is recommended. See Chapter 17, *Backing Up and Restoring a File System* for a procedure.

2. Download the Lustre server RPMs for your platform from the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository. See Table 8.2, “Packages Installed on Lustre Servers” for a list of required packages.

3. For a rolling upgrade, complete any procedures required to keep the Lustre file system running while the server to be upgraded is offline, such as failing over a primary server to its secondary partner.

4. Unmount the Lustre server to be upgraded (MGS, MDS, or OSS)

5. Install the Lustre server packages on the Lustre server.

   a. Log onto the Lustre server as the `root` user

   b. Use the `yum` command to install the packages:

   ```
   # yum --nogpgcheck install pkg1.rpm pkg2.rpm ...
   ```

   c. Verify the packages are installed correctly:

   ```
   rpm -qa|egrep "lustre|wc"
   ```

   d. Mount the Lustre server to restart the Lustre software on the server:

   ```
   server# mount -a -t lustre
   ```

   e. Repeat these steps on each Lustre server.

6. Download the Lustre client RPMs for your platform from the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository. See Table 8.3, “Packages Installed on Lustre Clients” for a list of required packages.

7. Install the Lustre client packages on each of the Lustre clients to be upgraded.

   a. Log onto a Lustre client as the `root` user.

   b. Use the `yum` command to install the packages:

   ```
   # yum --nogpgcheck install pkg1.rpm pkg2.rpm ...
   ```

   c. Verify the packages were installed correctly:

   ```
   # rpm -qa|egrep "lustre|kernel"
   ```

   d. Mount the Lustre client to restart the Lustre software on the client:

   ```
   client# mount -a -t lustre
   ```

   e. Repeat these steps on each Lustre client.

If you have a problem upgrading a Lustre file system, see Section 27.2, “ Reporting a Lustre File System Bug” for some suggestions for how to get help.

Chapter 17. Backing Up and Restoring a File System

This chapter describes how to backup and restore at the file system-level, device-level and file-level in a Lustre file system. Each backup approach is described in the the following sections:

- Section 17.1, " Backing up a File System"
- Section 17.2, "Backing Up and Restoring an MDS or OST (Device Level)"
- Section 17.3, "Making a File-Level Backup of an OST or MDT File System"
- Section 17.4, "Restoring a File-Level Backup"
- Section 17.5, "Using LVM Snapshots with the Lustre File System"

17.1. Backing up a File System

Backing up a complete file system gives you full control over the files to back up, and allows restoration of individual files as needed. File system-level backups are also the easiest to integrate into existing backup solutions.

File system backups are performed from a Lustre client (or many clients working parallel in different directories) rather than on individual server nodes; this is no different than backing up any other file system.

However, due to the large size of most Lustre file systems, it is not always possible to get a complete backup. We recommend that you back up subsets of a file system. This includes subdirectories of the entire file system, filesets for a single user, files incremented by date, and so on.

Note

In order to allow the file system namespace to scale for future applications, Lustre software release 2.x internally uses a 128-bit file identifier for all files. To interface with user applications, the Lustre software presents 64-bit inode numbers for the `stat()`, `fstat()`, and `readdir()` system calls on 64-bit applications, and 32-bit inode numbers to 32-bit applications.

Some 32-bit applications accessing Lustre file systems (on both 32-bit and 64-bit CPUs) may experience problems with the `stat()`, `fstat()` or `readdir()` system calls under certain circumstances, though the Lustre client should return 32-bit inode numbers to these applications.

In particular, if the Lustre file system is exported from a 64-bit client via NFS to a 32-bit client, the Linux NFS server will export 64-bit inode numbers to applications running on the NFS client. If the 32-bit applications are not compiled with Large File Support (LFS), then they return `EOVERFLOW` errors when accessing the Lustre files. To avoid this problem, Linux NFS clients can use the kernel command-line option "`nfs.enable_ino64=0`" in order to force the NFS client to export 32-bit inode numbers to the client.

Workaround: We very strongly recommend that backups using `tar(1)` and other utilities that depend on the inode number to uniquely identify an inode to be run on 64-bit clients. The 128-bit Lustre file identifiers cannot be uniquely mapped to a 32-bit inode number, and as a result these utilities may operate incorrectly on 32-bit clients.

17.1.1. Lustre_rsync

The lustre_rsync feature keeps the entire file system in sync on a backup by replicating the file system's changes to a second file system (the second file system need not be a Lustre file system, but it must be sufficiently large). lustre_rsync uses Lustre changelogs to efficiently synchronize the file systems without having to scan (directory walk) the Lustre file system. This efficiency is critically important for large file systems, and distinguishes the Lustre lustre_rsync feature from other replication/backup solutions.

17.1.1.1. Using Lustre_rsync

The lustre_rsync feature works by periodically running lustre_rsync, a userspace program used to synchronize changes in the Lustre file system onto the target file system. The lustre_rsync utility keeps a status file, which enables it to be safely interrupted and restarted without losing synchronization between the file systems.

The first time that lustre_rsync is run, the user must specify a set of parameters for the program to use. These parameters are described in the following table and in Section 37.13, " lustre_rsync". On subsequent runs, these parameters are stored in the the status file, and only the name of the status file needs to be passed to lustre_rsync.

Before using lustre_rsync:

- Register the changelog user. For details, see the Chapter 37, *System Configuration Utilities* (changelog_register) parameter in the Chapter 37, *System Configuration Utilities* (lctl).

- AND -

- Verify that the Lustre file system (source) and the replica file system (target) are identical *before* registering the changelog user. If the file systems are discrepant, use a utility, e.g. regular rsync (not lustre_rsync), to make them identical.

The lustre_rsync utility uses the following parameters:

Parameter	Description
--source=*src*	The path to the root of the Lustre file system (source) which will be synchronized. This is a mandatory option if a valid status log created during a previous synchronization operation (--statuslog) is not specified.
--target=*tgt*	The path to the root where the source file system will be synchronized (target). This is a mandatory option if the status log created during a previous synchronization operation (--statuslog) is not specified. This option can be repeated if multiple synchronization targets are desired.
--mdt=*mdt*	The metadata device to be synchronized. A changelog user must be registered for this device. This is a mandatory option if a valid status log created during a previous synchronization operation (--statuslog) is not specified.
--user=*userid*	The changelog user ID for the specified MDT. To use lustre_rsync, the changelog user must be registered. For details, see the changelog_register parameter in Chapter 37, *System Configuration Utilities* (lctl). This is a mandatory option if a valid status log created during a previous synchronization operation (--statuslog) is not specified.
--statuslog=*log*	A log file to which synchronization status is saved. When the lustre_rsync utility starts, if the status log from a previous synchronization operation is

Parameter	Description
	specified, then the state is read from the log and otherwise mandatory `--source`, `--target` and `--mdt` options can be skipped. Specifying the `--source`, `--target` and/or `--mdt` options, in addition to the `--statuslog` option, causes the specified parameters in the status log to be overridden. Command line options take precedence over options in the status log.
`--xattr` *yes|no*	Specifies whether extended attributes (`xattrs`) are synchronized or not. The default is to synchronize extended attributes. **Note** Disabling xattrs causes Lustre striping information not to be synchronized.
`--verbose`	Produces verbose output.
`--dry-run`	Shows the output of `lustre_rsync` commands (`copy`, `mkdir`, etc.) on the target file system without actually executing them.
`--abort-on-err`	Stops processing the `lustre_rsync` operation if an error occurs. The default is to continue the operation.

17.1.1.2. `lustre_rsync` Examples

Sample `lustre_rsync` commands are listed below.

Register a changelog user for an MDT (e.g. `testfs-MDT0000`).

```
# lctl --device testfs-MDT0000 changelog_register testfs-MDT0000
Registered changelog userid 'cl1'
```

Synchronize a Lustre file system (`/mnt/lustre`) to a target file system (`/mnt/target`).

```
$ lustre_rsync --source=/mnt/lustre --target=/mnt/target \
           --mdt=testfs-MDT0000 --user=cl1 --statuslog sync.log  --verbose
Lustre filesystem: testfs
MDT device: testfs-MDT0000
Source: /mnt/lustre
Target: /mnt/target
Statuslog: sync.log
Changelog registration: cl1
Starting changelog record: 0
Errors: 0
lustre_rsync took 1 seconds
Changelog records consumed: 22
```

After the file system undergoes changes, synchronize the changes onto the target file system. Only the `statuslog` name needs to be specified, as it has all the parameters passed earlier.

```
$ lustre_rsync --statuslog sync.log --verbose
Replicating Lustre filesystem: testfs
MDT device: testfs-MDT0000
Source: /mnt/lustre
Target: /mnt/target
Statuslog: sync.log
```

```
Changelog registration: cl1
Starting changelog record: 22
Errors: 0
lustre_rsync took 2 seconds
Changelog records consumed: 42
```

To synchronize a Lustre file system (`/mnt/lustre`) to two target file systems (`/mnt/target1` and `/mnt/target2`).

```
$ lustre_rsync --source=/mnt/lustre --target=/mnt/target1 \
           --target=/mnt/target2 --mdt=testfs-MDT0000 --user=cl1  \
           --statuslog sync.log
```

17.2. Backing Up and Restoring an MDS or OST (Device Level)

In some cases, it is useful to do a full device-level backup of an individual device (MDT or OST), before replacing hardware, performing maintenance, etc. Doing full device-level backups ensures that all of the data and configuration files is preserved in the original state and is the easiest method of doing a backup. For the MDT file system, it may also be the fastest way to perform the backup and restore, since it can do large streaming read and write operations at the maximum bandwidth of the underlying devices.

Note

Keeping an updated full backup of the MDT is especially important because a permanent failure of the MDT file system renders the much larger amount of data in all the OSTs largely inaccessible and unusable.

Introduced in Lustre 2.3

In Lustre software release 2.0 through 2.2, the only successful way to backup and restore an MDT is to do a device-level backup as is described in this section. File-level restore of an MDT is not possible before Lustre software release 2.3, as the Object Index (OI) file cannot be rebuilt after restore without the OI Scrub functionality. **Since Lustre software release 2.3**, Object Index files are automatically rebuilt at first mount after a restore is detected (see LU-957 [http://jira.hpdd.intel.com/browse/LU-957]), and file-level backup is supported (see Section 17.3, “Making a File-Level Backup of an OST or MDT File System”).

If hardware replacement is the reason for the backup or if a spare storage device is available, it is possible to do a raw copy of the MDT or OST from one block device to the other, as long as the new device is at least as large as the original device. To do this, run:

```
dd if=/dev/{original} of=/dev/{newdev} bs=1M
```

If hardware errors cause read problems on the original device, use the command below to allow as much data as possible to be read from the original device while skipping sections of the disk with errors:

```
dd if=/dev/{original} of=/dev/{newdev} bs=4k conv=sync,noerror /
      count={original size in 4kB blocks}
```

Even in the face of hardware errors, the `ldiskfs` file system is very robust and it may be possible to recover the file system data after running `e2fsck -fy /dev/{newdev}` on the new device, along with `ll_recover_lost_found_objs` for OST devices.

17.3. Making a File-Level Backup of an OST or MDT File System

This procedure provides an alternative to backup or migrate the data of an OST or MDT at the file level. At the file-level, unused space is omitted from the backed up and the process may be completed quicker with smaller total backup size. Backing up a single OST device is not necessarily the best way to perform backups of the Lustre file system, since the files stored in the backup are not usable without metadata stored on the MDT and additional file stripes that may be on other OSTs. However, it is the preferred method for migration of OST devices, especially when it is desirable to reformat the underlying file system with different configuration options or to reduce fragmentation.

Note

Prior to Lustre software release 2.3, the only successful way to perform an MDT backup and restore is to do a device-level backup as is described in Section 17.2, “Backing Up and Restoring an MDS or OST (Device Level)”. The ability to do MDT file-level backups is not available for Lustre software release 2.0 through 2.2, because restoration of the Object Index (OI) file does not return the MDT to a functioning state. **Since Lustre software release 2.3**, Object Index files are automatically rebuilt at first mount after a restore is detected (see LU-957 [http://jira.hpdd.intel.com/browse/LU-957]), so file-level MDT restore is supported.

For Lustre software release 2.3 and newer with MDT file-level backup support, substitute `mdt` for `ost` in the instructions below.

1. **Make a mountpoint for the file system.**

   ```
   [oss]# mkdir -p /mnt/ost
   ```

2. **Mount the file system.**

   ```
   [oss]# mount -t ldiskfs /dev/{ostdev} /mnt/ost
   ```

3. **Change to the mountpoint being backed up.**

   ```
   [oss]# cd /mnt/ost
   ```

4. **Back up the extended attributes.**

   ```
   [oss]# getfattr -R -d -m '.*' -e hex -P . > ea-$(date +%Y%m%d).bak
   ```

   ### Note

   If the `tar(1)` command supports the `--xattr` option, the `getfattr` step may be unnecessary as long as tar does a backup of the `trusted.*` attributes. However, completing this step is not harmful and can serve as an added safety measure.

   ### Note

   In most distributions, the `getfattr` command is part of the `attr` package. If the `getfattr` command returns errors like `Operation not supported`, then the kernel does not correctly support EAs. Stop and use a different backup method.

5. **Verify that the `ea-$date.bak` file has properly backed up the EA data on the OST.**

Without this attribute data, the restore process may be missing extra data that can be very useful in case of later file system corruption. Look at this file with more or a text editor. Each object file should have a corresponding item similar to this:

```
[oss]# file: O/0/d0/100992
trusted.fid= \
0x0d822200000000004a8a73e500000000808a0100000000000000000000000000
```

6. **Back up all file system data.**

   ```
   [oss]# tar czvf {backup file}.tgz [--xattrs] --sparse .
   ```

   ### Note

   The tar `--sparse` option is vital for backing up an MDT. In order to have `--sparse` behave correctly, and complete the backup of and MDT in finite time, the version of tar must be specified. Correctly functioning versions of tar include the Lustre software enhanced version of tar at https://wiki.hpdd.intel.com/display/PUB/Lustre+Tools#LustreTools-lustre-tar, the tar from a Red Hat Enterprise Linux distribution (version 6.3 or more recent) and the GNU tar version 1.25 or more recent.

   ### Warning

   The tar `--xattrs` option is only available in GNU tar distributions from Red Hat or Intel.

7. **Change directory out of the file system.**

   ```
   [oss]# cd -
   ```

8. **Unmount the file system.**

   ```
   [oss]# umount /mnt/ost
   ```

   ### Note

   When restoring an OST backup on a different node as part of an OST migration, you also have to change server NIDs and use the `--writeconf` command to re-generate the configuration logs. See Chapter 14, *Lustre Maintenance* (Changing a Server NID).

17.4. Restoring a File-Level Backup

To restore data from a file-level backup, you need to format the device, restore the file data and then restore the EA data.

1. Format the new device.

   ```
   [oss]# mkfs.lustre --ost --index {OST index} {other options} /dev/{newdev}
   ```

2. Set the file system label.

   ```
   [oss]# e2label {fsname}-OST{index in hex} /mnt/ost
   ```

3. Mount the file system.

   ```
   [oss]# mount -t ldiskfs /dev/{newdev} /mnt/ost
   ```

4. Change to the new file system mount point.

```
[oss]# cd /mnt/ost
```

5. Restore the file system backup.

```
[oss]# tar xzvpf {backup file} [--xattrs] --sparse
```

6. Restore the file system extended attributes.

```
[oss]# setfattr --restore=ea-${date}.bak
```

Note

If `--xattrs` option is supported by tar and specified in the step above, this step is redundant.

7. Verify that the extended attributes were restored.

```
[oss]# getfattr -d -m ".*" -e hex O/0/d0/100992 trusted.fid= \
0x0d822200000000004a8a73e500000000808a0100000000000000000000000000
```

8. Change directory out of the file system.

```
[oss]# cd -
```

9. Unmount the new file system.

```
[oss]# umount /mnt/ost
```

If the file system was used between the time the backup was made and when it was restored, then the online `LFSCK` tool (part of Lustre code) will automatically be run to ensure the file system is coherent. If all of the device file systems were backed up at the same time after the entire Lustre file system was stopped, this is not necessary. In either case, the file system should be immediately usable even if `LFSCK` is not run, though there may be I/O errors reading from files that are present on the MDT but not the OSTs, and files that were created after the MDT backup will not be accessible/visible. See Section 28.4, “Checking the file system with LFSCK” for details on using LFSCK.

17.5. Using LVM Snapshots with the Lustre File System

If you want to perform disk-based backups (because, for example, access to the backup system needs to be as fast as to the primary Lustre file system), you can use the Linux LVM snapshot tool to maintain multiple, incremental file system backups.

Because LVM snapshots cost CPU cycles as new files are written, taking snapshots of the main Lustre file system will probably result in unacceptable performance losses. You should create a new, backup Lustre file system and periodically (e.g., nightly) back up new/changed files to it. Periodic snapshots can be taken of this backup file system to create a series of "full" backups.

Note

Creating an LVM snapshot is not as reliable as making a separate backup, because the LVM snapshot shares the same disks as the primary MDT device, and depends on the primary MDT

device for much of its data. If the primary MDT device becomes corrupted, this may result in the snapshot being corrupted.

17.5.1. Creating an LVM-based Backup File System

Use this procedure to create a backup Lustre file system for use with the LVM snapshot mechanism.

1. Create LVM volumes for the MDT and OSTs.

   Create LVM devices for your MDT and OST targets. Make sure not to use the entire disk for the targets; save some room for the snapshots. The snapshots start out as 0 size, but grow as you make changes to the current file system. If you expect to change 20% of the file system between backups, the most recent snapshot will be 20% of the target size, the next older one will be 40%, etc. Here is an example:

   ```
   cfs21:~# pvcreate /dev/sda1
      Physical volume "/dev/sda1" successfully created
   cfs21:~# vgcreate vgmain /dev/sda1
      Volume group "vgmain" successfully created
   cfs21:~# lvcreate -L200G -nMDT0 vgmain
      Logical volume "MDT0" created
   cfs21:~# lvcreate -L200G -nOST0 vgmain
      Logical volume "OST0" created
   cfs21:~# lvscan
      ACTIVE                  '/dev/vgmain/MDT0' [200.00 GB] inherit
      ACTIVE                  '/dev/vgmain/OST0' [200.00 GB] inherit
   ```

2. Format the LVM volumes as Lustre targets.

   In this example, the backup file system is called `main` and designates the current, most up-to-date backup.

   ```
   cfs21:~# mkfs.lustre --fsname=main --mdt --index=0 /dev/vgmain/MDT0
    No management node specified, adding MGS to this MDT.
       Permanent disk data:
    Target:     main-MDT0000
    Index:      0
    Lustre FS:  main
    Mount type: ldiskfs
    Flags:      0x75
                  (MDT MGS first_time update )
    Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
    Parameters:
   checking for existing Lustre data
    device size = 200GB
    formatting backing filesystem ldiskfs on /dev/vgmain/MDT0
            target name  main-MDT0000
            4k blocks     0
            options        -i 4096 -I 512 -q -O dir_index -F
    mkfs_cmd = mkfs.ext2 -j -b 4096 -L main-MDT0000  -i 4096 -I 512 -q
     -O dir_index -F /dev/vgmain/MDT0
    Writing CONFIGS/mountdata
   cfs21:~# mkfs.lustre --mgsnode=cfs21 --fsname=main --ost --index=0 /dev/vgmain/O
       Permanent disk data:
    Target:     main-OST0000
    Index:      0
   ```

```
   Lustre FS:  main
   Mount type: ldiskfs
   Flags:      0x72
                 (OST first_time update )
   Persistent mount opts: errors=remount-ro,extents,mballoc
   Parameters: mgsnode=192.168.0.21@tcp
  checking for existing Lustre data
   device size = 200GB
   formatting backing filesystem ldiskfs on /dev/vgmain/OST0
           target name  main-OST0000
           4k blocks     0
           options        -I 256 -q -O dir_index -F
   mkfs_cmd = mkfs.ext2 -j -b 4096 -L lustre-OST0000 -J size=400 -I 256
    -i 262144 -O extents,uninit_bg,dir_nlink,huge_file,flex_bg -G 256
    -E resize=4290772992,lazy_journal_init, -F /dev/vgmain/OST0
   Writing CONFIGS/mountdata
  cfs21:~# mount -t lustre /dev/vgmain/MDT0 /mnt/mdt
  cfs21:~# mount -t lustre /dev/vgmain/OST0 /mnt/ost
  cfs21:~# mount -t lustre cfs21:/main /mnt/main
```

17.5.2. Backing up New/Changed Files to the Backup File System

At periodic intervals e.g., nightly, back up new and changed files to the LVM-based backup file system.

```
cfs21:~# cp /etc/passwd /mnt/main

cfs21:~# cp /etc/fstab /mnt/main

cfs21:~# ls /mnt/main
fstab  passwd
```

17.5.3. Creating Snapshot Volumes

Whenever you want to make a "checkpoint" of the main Lustre file system, create LVM snapshots of all target MDT and OSTs in the LVM-based backup file system. You must decide the maximum size of a snapshot ahead of time, although you can dynamically change this later. The size of a daily snapshot is dependent on the amount of data changed daily in the main Lustre file system. It is likely that a two-day old snapshot will be twice as big as a one-day old snapshot.

You can create as many snapshots as you have room for in the volume group. If necessary, you can dynamically add disks to the volume group.

The snapshots of the target MDT and OSTs should be taken at the same point in time. Make sure that the cronjob updating the backup file system is not running, since that is the only thing writing to the disks. Here is an example:

```
cfs21:~# modprobe dm-snapshot
cfs21:~# lvcreate -L50M -s -n MDT0.b1 /dev/vgmain/MDT0
   Rounding up size to full physical extent 52.00 MB
   Logical volume "MDT0.b1" created
cfs21:~# lvcreate -L50M -s -n OST0.b1 /dev/vgmain/OST0
   Rounding up size to full physical extent 52.00 MB
```

```
  Logical volume "OST0.b1" created
```

After the snapshots are taken, you can continue to back up new/changed files to "main". The snapshots will not contain the new files.

```
cfs21:~# cp /etc/termcap /mnt/main
cfs21:~# ls /mnt/main
fstab  passwd  termcap
```

17.5.4. Restoring the File System From a Snapshot

Use this procedure to restore the file system from an LVM snapshot.

1. Rename the LVM snapshot.

   Rename the file system snapshot from "main" to "back" so you can mount it without unmounting "main". This is recommended, but not required. Use the `--reformat` flag to `tunefs.lustre` to force the name change. For example:

```
cfs21:~# tunefs.lustre --reformat --fsname=back --writeconf /dev/vgmain/MDT0.b1
 checking for existing Lustre data
 found Lustre data
 Reading CONFIGS/mountdata
Read previous values:
 Target:     main-MDT0000
 Index:      0
 Lustre FS:  main
 Mount type: ldiskfs
 Flags:      0x5
              (MDT MGS )
 Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
 Parameters:
Permanent disk data:
 Target:     back-MDT0000
 Index:      0
 Lustre FS:  back
 Mount type: ldiskfs
 Flags:      0x105
              (MDT MGS writeconf )
 Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
 Parameters:
Writing CONFIGS/mountdata
cfs21:~# tunefs.lustre --reformat --fsname=back --writeconf /dev/vgmain/OST0.b1
 checking for existing Lustre data
 found Lustre data
 Reading CONFIGS/mountdata
Read previous values:
 Target:     main-OST0000
 Index:      0
 Lustre FS:  main
 Mount type: ldiskfs
 Flags:      0x2
              (OST )
 Persistent mount opts: errors=remount-ro,extents,mballoc
```

```
 Parameters: mgsnode=192.168.0.21@tcp
Permanent disk data:
 Target:     back-OST0000
 Index:      0
 Lustre FS:  back
 Mount type: ldiskfs
 Flags:      0x102
              (OST writeconf )
 Persistent mount opts: errors=remount-ro,extents,mballoc
 Parameters: mgsnode=192.168.0.21@tcp
Writing CONFIGS/mountdata
```

When renaming a file system, we must also erase the last_rcvd file from the snapshots

```
cfs21:~# mount -t ldiskfs /dev/vgmain/MDT0.b1 /mnt/mdtback
cfs21:~# rm /mnt/mdtback/last_rcvd
cfs21:~# umount /mnt/mdtback
cfs21:~# mount -t ldiskfs /dev/vgmain/OST0.b1 /mnt/ostback
cfs21:~# rm /mnt/ostback/last_rcvd
cfs21:~# umount /mnt/ostback
```

2. Mount the file system from the LVM snapshot. For example:

```
cfs21:~# mount -t lustre /dev/vgmain/MDT0.b1 /mnt/mdtback
cfs21:~# mount -t lustre /dev/vgmain/OST0.b1 /mnt/ostback
cfs21:~# mount -t lustre cfs21:/back /mnt/back
```

3. Note the old directory contents, as of the snapshot time. For example:

```
cfs21:~/cfs/b1_5/lustre/utils# ls /mnt/back
fstab  passwds
```

17.5.5. Deleting Old Snapshots

To reclaim disk space, you can erase old snapshots as your backup policy dictates. Run:

```
lvremove /dev/vgmain/MDT0.b1
```

17.5.6. Changing Snapshot Volume Size

You can also extend or shrink snapshot volumes if you find your daily deltas are smaller or larger than expected. Run:

```
lvextend -L10G /dev/vgmain/MDT0.b1
```

Note

Extending snapshots seems to be broken in older LVM. It is working in LVM v2.02.01.

Chapter 18. Managing File Layout (Striping) and Free Space

This chapter describes file layout (striping) and I/O options, and includes the following sections:

- Section 18.1, " How Lustre File System Striping Works"
- Section 18.2, " Lustre File Layout (Striping) Considerations"
- Section 18.3, "Setting the File Layout/Striping Configuration (`lfs setstripe`)"
- Section 18.4, "Retrieving File Layout/Striping Information (`getstripe`)"
- Section 18.5, "Managing Free Space"
- Section 18.6, "Lustre Striping Internals"

18.1. How Lustre File System Striping Works

In a Lustre file system, the MDS allocates objects to OSTs using either a round-robin algorithm or a weighted algorithm. When the amount of free space is well balanced (i.e., by default, when the free space across OSTs differs by less than 17%), the round-robin algorithm is used to select the next OST to which a stripe is to be written. Periodically, the MDS adjusts the striping layout to eliminate some degenerated cases in which applications that create very regular file layouts (striping patterns) preferentially use a particular OST in the sequence.

Normally the usage of OSTs is well balanced. However, if users create a small number of exceptionally large files or incorrectly specify striping parameters, imbalanced OST usage may result. When the free space across OSTs differs by more than a specific amount (17% by default), the MDS then uses weighted random allocations with a preference for allocating objects on OSTs with more free space. (This can reduce I/O performance until space usage is rebalanced again.) For a more detailed description of how striping is allocated, see Section 18.5, "Managing Free Space".

Introduced in Lustre 2.2

Files can only be striped over a finite number of OSTs. Prior to Lustre software release 2.2, the maximum number of OSTs that a file could be striped across was limited to 160. As of Lustre software release 2.2, the maximum number of OSTs is 2000. For more information, see Section 18.6, "Lustre Striping Internals".

18.2. Lustre File Layout (Striping) Considerations

Whether you should set up file striping and what parameter values you select depends on your needs. A good rule of thumb is to stripe over as few objects as will meet those needs and no more.

Some reasons for using striping include:

- **Providing high-bandwidth access.** Many applications require high-bandwidth access to a single file, which may be more bandwidth than can be provided by a single OSS. Examples are a scientific application that writes to a single file from hundreds of nodes, or a binary executable that is loaded by many nodes when an application starts.

In cases like these, a file can be striped over as many OSSs as it takes to achieve the required peak aggregate bandwidth for that file. Striping across a larger number of OSSs should only be used when the file size is very large and/or is accessed by many nodes at a time. Currently, Lustre files can be striped across up to 2000 OSTs, the maximum stripe count for an `ldiskfs` file system.

- **Improving performance when OSS bandwidth is exceeded.** Striping across many OSSs can improve performance if the aggregate client bandwidth exceeds the server bandwidth and the application reads and writes data fast enough to take advantage of the additional OSS bandwidth. The largest useful stripe count is bounded by the I/O rate of the clients/jobs divided by the performance per OSS.
- **Providing space for very large files.** Striping is useful when a single OST does not have enough free space to hold the entire file.

Some reasons to minimize or avoid striping:

- **Increased overhead.** Striping results in more locks and extra network operations during common operations such as `stat` and `unlink`. Even when these operations are performed in parallel, one network operation takes less time than 100 operations.

  Increased overhead also results from server contention. Consider a cluster with 100 clients and 100 OSSs, each with one OST. If each file has exactly one object and the load is distributed evenly, there is no contention and the disks on each server can manage sequential I/O. If each file has 100 objects, then the clients all compete with one another for the attention of the servers, and the disks on each node seek in 100 different directions resulting in needless contention.
- **Increased risk.** When files are striped across all servers and one of the servers breaks down, a small part of each striped file is lost. By comparison, if each file has exactly one stripe, fewer files are lost, but they are lost in their entirety. Many users would prefer to lose some of their files entirely than all of their files partially.

18.2.1. Choosing a Stripe Size

Choosing a stripe size is a balancing act, but reasonable defaults are described below. The stripe size has no effect on a single-stripe file.

- **The stripe size must be a multiple of the page size.** Lustre software tools enforce a multiple of 64 KB (the maximum page size on ia64 and PPC64 nodes) so that users on platforms with smaller pages do not accidentally create files that might cause problems for ia64 clients.
- **The smallest recommended stripe size is 512 KB.** Although you can create files with a stripe size of 64 KB, the smallest practical stripe size is 512 KB because the Lustre file system sends 1MB chunks over the network. Choosing a smaller stripe size may result in inefficient I/O to the disks and reduced performance.
- **A good stripe size for sequential I/O using high-speed networks is between 1 MB and 4 MB.** In most situations, stripe sizes larger than 4 MB may result in longer lock hold times and contention during shared file access.
- **The maximum stripe size is 4 GB.** Using a large stripe size can improve performance when accessing very large files. It allows each client to have exclusive access to its own part of a file. However, a large stripe size can be counterproductive in cases where it does not match your I/O pattern.
- **Choose a stripe pattern that takes into account the write patterns of your application.** Writes that cross an object boundary are slightly less efficient than writes that go entirely to one server. If the file is written in a consistent and aligned way, make the stripe size a multiple of the `write()` size.

18.3. Setting the File Layout/Striping Configuration (`lfs setstripe`)

Use the `lfs setstripe` command to create new files with a specific file layout (stripe pattern) configuration.

```
lfs setstripe [--size|-s stripe_size] [--count|-c stripe_count] \
[--index|-i start_ost] [--pool|-p pool_name] filename|dirname
```

`stripe_size`

The `stripe_size` indicates how much data to write to one OST before moving to the next OST. The default `stripe_size` is 1 MB. Passing a `stripe_size` of 0 causes the default stripe size to be used. Otherwise, the `stripe_size` value must be a multiple of 64 KB.

`stripe_count`

The `stripe_count` indicates how many OSTs to use. The default `stripe_count` value is 1. Setting `stripe_count` to 0 causes the default stripe count to be used. Setting `stripe_count` to -1 means stripe over all available OSTs (full OSTs are skipped).

`start_ost`

The start OST is the first OST to which files are written. The default value for `start_ost` is -1, which allows the MDS to choose the starting index. This setting is strongly recommended, as it allows space and load balancing to be done by the MDS as needed. If the value of `start_ost` is set to a value other than -1, the file starts on the specified OST index. OST index numbering starts at 0.

Note

If the specified OST is inactive or in a degraded mode, the MDS will silently choose another target.

Note

If you pass a `start_ost` value of 0 and a `stripe_count` value of *1*, all files are written to OST 0, until space is exhausted. *This is probably not what you meant to do.* If you only want to adjust the stripe count and keep the other parameters at their default settings, do not specify any of the other parameters:

```
client# lfs setstripe -c stripe_count filename
```

`pool_name`

The `pool_name` specifies the OST pool to which the file will be written. This allows limiting the OSTs used to a subset of all OSTs in the file system. For more details about using OST pools, see Creating and Managing OST Pools [ManagingFileSystemIO.html#50438211_75549].

18.3.1. Specifying a File Layout (Striping Pattern) for a Single File

It is possible to specify the file layout when a new file is created using the command `lfs setstripe`. This allows users to override the file system default parameters to tune the file layout more optimally for their application. Execution of an `lfs setstripe` command fails if the file already exists.

18.3.1.1. Setting the Stripe Size

The command to create a new file with a specified stripe size is similar to:

```
[client]# lfs setstripe -s 4M /mnt/lustre/new_file
```

This example command creates the new file `/mnt/lustre/new_file` with a stripe size of 4 MB.

Now, when a file is created, the new stripe setting evenly distributes the data over all the available OSTs:

```
 [client]# lfs getstripe /mnt/lustre/new_file
/mnt/lustre/4mb_file
lmm_stripe_count:   1
lmm_stripe_size:    4194304
lmm_stripe_offset:  1
obdidx     objid        objid           group
1          690550       0xa8976         0
```

In this example, the stripe size is 4 MB.

18.3.1.2. Setting the Stripe Count

The command below creates a new file with a stripe count of `-1` to specify striping over all available OSTs:

```
[client]# lfs setstripe -c -1 /mnt/lustre/full_stripe
```

The example below indicates that the file `full_stripe` is striped over all six active OSTs in the configuration:

```
[client]# lfs getstripe /mnt/lustre/full_stripe
/mnt/lustre/full_stripe
  obdidx   objid   objid   group
  0        8       0x8     0
  1        4       0x4     0
  2        5       0x5     0
  3        5       0x5     0
  4        4       0x4     0
  5        2       0x2     0
```

This is in contrast to the output in Section 18.3.1.1, “Setting the Stripe Size”, which shows only a single object for the file.

18.3.2. Setting the Striping Layout for a Directory

In a directory, the `lfs setstripe` command sets a default striping configuration for files created in the directory. The usage is the same as `lfs setstripe` for a regular file, except that the directory must exist prior to setting the default striping configuration. If a file is created in a directory with a default stripe configuration (without otherwise specifying striping), the Lustre file system uses those striping parameters instead of the file system default for the new file.

To change the striping pattern for a sub-directory, create a directory with desired file layout as described above. Sub-directories inherit the file layout of the root/parent directory.

18.3.3. Setting the Striping Layout for a File System

Setting the striping specification on the `root` directory determines the striping for all new files created in the file system unless an overriding striping specification takes precedence (such as a striping layout specified by the application, or set using `lfs setstripe`, or specified for the parent directory).

Note

The striping settings for a `root` directory are, by default, applied to any new child directories created in the root directory, unless striping settings have been specified for the child directory.

18.3.4. Creating a File on a Specific OST

You can use `lfs setstripe` to create a file on a specific OST. In the following example, the file `file1` is created on the first OST (OST index is 0).

```
$ lfs setstripe --count 1 --index 0 file1
$ dd if=/dev/zero of=file1 count=1 bs=100M
1+0 records in
1+0 records out

$ lfs getstripe file1
/mnt/testfs/file1
/mnt/testfs/file1
lmm_stripe_count:   1
lmm_stripe_size:    1048576
lmm_stripe_offset:  0
     obdidx    objid   objid    group
     0         37364   0x91f4   0
```

18.4. Retrieving File Layout/Striping Information (`getstripe`)

The `lfs getstripe` command is used to display information that shows over which OSTs a file is distributed. For each OST, the index and UUID is displayed, along with the OST index and object ID for each stripe in the file. For directories, the default settings for files created in that directory are displayed.

18.4.1. Displaying the Current Stripe Size

To see the current stripe size for a Lustre file or directory, use the `lfs getstripe` command. For example, to view information for a directory, enter a command similar to:

```
[client]# lfs getstripe /mnt/lustre
```

This command produces output similar to:

```
/mnt/lustre
(Default) stripe_count: 1 stripe_size: 1M stripe_offset: -1
```

In this example, the default stripe count is `1` (data blocks are striped over a single OST), the default stripe size is 1 MB, and the objects are created over all available OSTs.

To view information for a file, enter a command similar to:

```
$ lfs getstripe /mnt/lustre/foo
/mnt/lustre/foo
  obdidx   objid    objid      group
  2        835487   m0xcbf9f   0
```

In this example, the file is located on `obdidx 2`, which corresponds to the OST `lustre-OST0002`. To see which node is serving that OST, run:

```
$ lctl get_param osc.lustre-OST0002-osc.ost_conn_uuid
osc.lustre-OST0002-osc.ost_conn_uuid=192.168.20.1@tcp
```

18.4.2. Inspecting the File Tree

To inspect an entire tree of files, use the `lfs find` command:

```
lfs find [--recursive | -r] file|directory ...
```

18.4.3. Locating the MDT for a remote directory

Introduced in Lustre 2.4

Lustre software release 2.4 can be configured with multiple MDTs in the same file system. Each sub-directory can have a different MDT. To identify on which MDT a given subdirectory is located, pass the `getstripe -M` parameters to `lfs`. An example of this command is provided in the section Section 14.8.1, “Removing a MDT from the File System”.

18.5. Managing Free Space

To optimize file system performance, the MDT assigns file stripes to OSTs based on two allocation algorithms. The *round-robin* allocator gives preference to location (spreading out stripes across OSSs to increase network bandwidth utilization) and the weighted allocator gives preference to available space (balancing loads across OSTs). Threshold and weighting factors for these two algorithms can be adjusted by the user. This section describes how to check available free space on disks and how free space is allocated. It then describes how to set the threshold and weighting factors for the allocation algorithms.

18.5.1. Checking File System Free Space

Free space is an important consideration in assigning file stripes. The `lfs df` command can be used to show available disk space on the mounted Lustre file system and space consumption per OST. If multiple Lustre file systems are mounted, a path may be specified, but is not required. Options to the `lfs df` command are shown below.

Option	Description
`-h`	Displays sizes in human readable format (for example: 1K, 234M, 5G).
`-i, --inodes`	Lists inodes instead of block usage.

Note

The `df -i` and `lfs df -i` commands show the *minimum* number of inodes that can be created in the file system at the current time. If the total number of objects available across all of the

OSTs is smaller than those available on the MDT(s), taking into account the default file striping, then `df -i` will also report a smaller number of inodes than could be created. Running `lfs df -i` will report the actual number of inodes that are free on each target.

For ZFS file systems, the number of inodes that can be created is dynamic and depends on the free space in the file system. The Free and Total inode counts reported for a ZFS file system are only an estimate based on the current usage for each target. The Used inode count is the actual number of inodes used by the file system.

Examples

```
[client1] $ lfs df
UUID                1K-blockS  Used      Available Use% Mounted on
mds-lustre-0_UUID   9174328    1020024   8154304   11%  /mnt/lustre[MDT:0]
ost-lustre-0_UUID   94181368   56330708  37850660  59%  /mnt/lustre[OST:0]
ost-lustre-1_UUID   94181368   56385748  37795620  59%  /mnt/lustre[OST:1]
ost-lustre-2_UUID   94181368   54352012  39829356  57%  /mnt/lustre[OST:2]
filesystem summary: 282544104  167068468 39829356  57%  /mnt/lustre

[client1] $ lfs df -h
UUID                bytes    Used    Available   Use%  Mounted on
mds-lustre-0_UUID   8.7G     996.1M  7.8G        11%   /mnt/lustre[MDT:0]
ost-lustre-0_UUID   89.8G    53.7G   36.1G       59%   /mnt/lustre[OST:0]
ost-lustre-1_UUID   89.8G    53.8G   36.0G       59%   /mnt/lustre[OST:1]
ost-lustre-2_UUID   89.8G    51.8G   38.0G       57%   /mnt/lustre[OST:2]
filesystem summary: 269.5G   159.3G  110.1G      59%   /mnt/lustre

[client1] $ lfs df -i
UUID                Inodes  IUsed IFree   IUse% Mounted on
mds-lustre-0_UUID   2211572 41924 2169648 1%    /mnt/lustre[MDT:0]
ost-lustre-0_UUID   737280  12183 725097  1%    /mnt/lustre[OST:0]
ost-lustre-1_UUID   737280  12232 725048  1%    /mnt/lustre[OST:1]
ost-lustre-2_UUID   737280  12214 725066  1%    /mnt/lustre[OST:2]
filesystem summary: 2211572 41924 2169648 1%    /mnt/lustre[OST:2]
```

18.5.2. Stripe Allocation Methods

Two stripe allocation methods are provided:

- **Round-robin allocator** - When the OSTs have approximately the same amount of free space, the round-robin allocator alternates stripes between OSTs on different OSSs, so the OST used for stripe 0 of each file is evenly distributed among OSTs, regardless of the stripe count. In a simple example with eight OSTs numbered 0-7, objects would be allocated like this:

  ```
  File 1: OST1, OST2, OST3, OST4
  File 2: OST5, OST6, OST7
  File 3: OST0, OST1, OST2, OST3, OST4, OST5
  File 4: OST6, OST7, OST0
  ```

  Here are several more sample round-robin stripe orders (each letter represents a different OST on a single OSS):

3: AAA	One 3-OST OSS
3x3: ABABAB	Two 3-OST OSSs

3x4: BBABABA	One 3-OST OSS (A) and one 4-OST OSS (B)
3x5: BBABBABA	One 3-OST OSS (A) and one 5-OST OSS (B)
3x3x3: ABCABCABC	Three 3-OST OSSs

- **Weighted allocator** - When the free space difference between the OSTs becomes significant, the weighting algorithm is used to influence OST ordering based on size (amount of free space available on each OST) and location (stripes evenly distributed across OSTs). The weighted allocator fills the emptier OSTs faster, but uses a weighted random algorithm, so the OST with the most free space is not necessarily chosen each time.

The allocation method is determined by the amount of free-space imbalance on the OSTs. When free space is relatively balanced across OSTs, the faster round-robin allocator is used, which maximizes network balancing. The weighted allocator is used when any two OSTs are out of balance by more than the specified threshold (17% by default). The threshold between the two allocation methods is defined in the file `/proc/fs/fsname/lov/fsname-mdtlov/qos_threshold_rr`.

To set the `qos_threshold_r` to `25`, enter this command on the MGS:

```
lctl set_param lov.fsname-mdtlov.quos_threshold_rr=25
```

18.5.3. Adjusting the Weighting Between Free Space and Location

The weighting priority used by the weighted allocator is set in the file `/proc/fs/fsname/lov/fsname-mdtlov/qos_prio_free`. Increasing the value of `qos_prio_free` puts more weighting on the amount of free space available on each OST and less on how stripes are distributed across OSTs. The default value is `91` (percent). When the free space priority is set to `100` (percent), weighting is based entirely on free space and location is no longer used by the striping algorithm.

To change the allocator weighting to `100`, enter this command on the MGS:

```
lctl conf_param fsname-MDT0000.lov.qos_prio_free=100
```

.

Note

When `qos_prio_free` is set to `100`, a weighted random algorithm is still used to assign stripes, so, for example, if OST2 has twice as much free space as OST1, OST2 is twice as likely to be used, but it is not guaranteed to be used.

18.6. Lustre Striping Internals

For Lustre releases prior to Lustre software release 2.2, files can be striped across a maximum of 160 OSTs. Lustre inodes use an extended attribute to record the location of each object (the object ID and the number of the OST on which it is stored). The size of the extended attribute limits the maximum stripe count to 160 objects.

Introduced in Lustre 2.2

In Lustre software release 2.2 and subsequent releases, the maximum number of OSTs over which files can be striped has been raised to 2000 by allocating a new block on which to store the extended attribute that holds the object information. This feature, known as "wide striping," only allocates the additional extended attribute data block if the file is striped with a stripe count greater than 160. The file layout (object ID,

OST number) is stored on the new data block with a pointer to this block stored in the original Lustre inode for the file. For files smaller than 160 objects, the Lustre inode is used to store the file layout.

Chapter 19. Managing the File System and I/O

This chapter describes file striping and I/O options, and includes the following sections:

- Section 19.1, " Handling Full OSTs"
- Section 19.2, " Creating and Managing OST Pools"
- Section 19.3, "Adding an OST to a Lustre File System"
- Section 19.4, "Performing Direct I/O"
- Section 19.5, "Other I/O Options"

19.1. Handling Full OSTs

Sometimes a Lustre file system becomes unbalanced, often due to incorrectly-specified stripe settings, or when very large files are created that are not striped over all of the OSTs. If an OST is full and an attempt is made to write more information to the file system, an error occurs. The procedures below describe how to handle a full OST.

The MDS will normally handle space balancing automatically at file creation time, and this procedure is normally not needed, but may be desirable in certain circumstances (e.g. when creating very large files that would consume more than the total free space of the full OSTs).

19.1.1. Checking OST Space Usage

The example below shows an unbalanced file system:

```
client# lfs df -h
UUID                       bytes           Used            Available       \
Use%            Mounted on
lustre-MDT0000_UUID        4.4G            214.5M          3.9G            \
4%              /mnt/lustre[MDT:0]
lustre-OST0000_UUID        2.0G            751.3M          1.1G            \
37%             /mnt/lustre[OST:0]
lustre-OST0001_UUID        2.0G            755.3M          1.1G            \
37%             /mnt/lustre[OST:1]
lustre-OST0002_UUID        2.0G            1.7G            155.1M          \
86%             /mnt/lustre[OST:2] <-
lustre-OST0003_UUID        2.0G            751.3M          1.1G            \
37%             /mnt/lustre[OST:3]
lustre-OST0004_UUID        2.0G            747.3M          1.1G            \
37%             /mnt/lustre[OST:4]
lustre-OST0005_UUID        2.0G            743.3M          1.1G            \
36%             /mnt/lustre[OST:5]

filesystem summary:        11.8G           5.4G            5.8G            \
45%             /mnt/lustre
```

In this case, OST:2 is almost full and when an attempt is made to write additional information to the file system (even with uniform striping over all the OSTs), the write command fails as follows:

```
client# lfs setstripe /mnt/lustre 4M 0 -1
client# dd if=/dev/zero of=/mnt/lustre/test_3 bs=10M count=100
dd: writing '/mnt/lustre/test_3': No space left on device
98+0 records in
97+0 records out
1017192448 bytes (1.0 GB) copied, 23.2411 seconds, 43.8 MB/s
```

19.1.2. Taking a Full OST Offline

To avoid running out of space in the file system, if the OST usage is imbalanced and one or more OSTs are close to being full while there are others that have a lot of space, the full OSTs may optionally be deactivated at the MDS to prevent the MDS from allocating new objects there.

1. Log into the MDS server:

   ```
   client# ssh root@192.168.0.10
   root@192.168.0.10's password:
   Last login: Wed Nov 26 13:35:12 2008 from 192.168.0.6
   ```

2. Use the `lctl dl` command to show the status of all file system components:

   ```
   mds# lctl dl
   0 UP mgs MGS MGS 9
   1 UP mgc MGC192.168.0.10@tcp e384bb0e-680b-ce25-7bc9-81655dd1e813 5
   2 UP mdt MDS MDS_uuid 3
   3 UP lov lustre-mdtlov lustre-mdtlov_UUID 4
   4 UP mds lustre-MDT0000 lustre-MDT0000_UUID 5
   5 UP osc lustre-OST0000-osc lustre-mdtlov_UUID 5
   6 UP osc lustre-OST0001-osc lustre-mdtlov_UUID 5
   7 UP osc lustre-OST0002-osc lustre-mdtlov_UUID 5
   8 UP osc lustre-OST0003-osc lustre-mdtlov_UUID 5
   9 UP osc lustre-OST0004-osc lustre-mdtlov_UUID 5
   10 UP osc lustre-OST0005-osc lustre-mdtlov_UUID 5
   ```

3. Use `lctl` deactivate to take the full OST offline:

   ```
   mds# lctl --device 7 deactivate
   ```

4. Display the status of the file system components:

   ```
   mds# lctl dl
   0 UP mgs MGS MGS 9
   1 UP mgc MGC192.168.0.10@tcp e384bb0e-680b-ce25-7bc9-81655dd1e813 5
   2 UP mdt MDS MDS_uuid 3
   3 UP lov lustre-mdtlov lustre-mdtlov_UUID 4
   4 UP mds lustre-MDT0000 lustre-MDT0000_UUID 5
   5 UP osc lustre-OST0000-osc lustre-mdtlov_UUID 5
   6 UP osc lustre-OST0001-osc lustre-mdtlov_UUID 5
   7 IN osc lustre-OST0002-osc lustre-mdtlov_UUID 5
   8 UP osc lustre-OST0003-osc lustre-mdtlov_UUID 5
   9 UP osc lustre-OST0004-osc lustre-mdtlov_UUID 5
   10 UP osc lustre-OST0005-osc lustre-mdtlov_UUID 5
   ```

The device list shows that OST0002 is now inactive. When new files are created in the file system, they will only use the remaining active OSTs. Either manual space rebalancing can be done by migrating data to other OSTs, as shown in the next section, or normal file deletion and creation can be allowed to passively rebalance the space usage.

19.1.3. Migrating Data within a File System

As stripes cannot be moved within the file system, data must be migrated manually by copying and renaming the file, removing the original file, and renaming the new file with the original file name. The simplest way to do this is to use the `lfs_migrate` command (see Section 33.2, " `lfs_migrate` "). However, the steps for migrating a file by hand are also shown here for reference.

1. Identify the file(s) to be moved.

   In the example below, output from the `getstripe` command indicates that the file `test_2` is located entirely on OST2:

   ```
   client# lfs getstripe /mnt/lustre/test_2
   /mnt/lustre/test_2
   obdidx     objid   objid   group
        2      8     0x8       0
   ```

2. To move single object(s), create a new copy and remove the original. Enter:

   ```
   client# cp -a /mnt/lustre/test_2 /mnt/lustre/test_2.tmp
   client# mv /mnt/lustre/test_2.tmp /mnt/lustre/test_2
   ```

3. To migrate large files from one or more OSTs, enter:

   ```
   client# lfs find --ost ost_name -size +1G | lfs_migrate -y
   ```

4. Check the file system balance.

   The `df` output in the example below shows a more balanced system compared to the `df` output in the example in Section 19.1, " Handling Full OSTs".

   ```
   client# lfs df -h
   UUID                 bytes         Used            Available       Use%    \
           Mounted on
   lustre-MDT0000_UUID   4.4G         214.5M          3.9G            4%      \
           /mnt/lustre[MDT:0]
   lustre-OST0000_UUID   2.0G         1.3G            598.1M          65%     \
           /mnt/lustre[OST:0]
   lustre-OST0001_UUID   2.0G         1.3G            594.1M          65%     \
           /mnt/lustre[OST:1]
   lustre-OST0002_UUID   2.0G         913.4M          1000.0M         45%     \
           /mnt/lustre[OST:2]
   lustre-OST0003_UUID   2.0G         1.3G            602.1M          65%     \
           /mnt/lustre[OST:3]
   lustre-OST0004_UUID   2.0G         1.3G            606.1M          64%     \
           /mnt/lustre[OST:4]
   lustre-OST0005_UUID   2.0G         1.3G            610.1M          64%     \
           /mnt/lustre[OST:5]

   filesystem summary:  11.8G 7.3G           3.9G    61%                     \
   /mnt/lustre
   ```

19.1.4. Returning an Inactive OST Back Online

Once the deactivated OST(s) no longer are severely imbalanced, due to either active or passive data redistribution, they should be reactivated so they will again have new files allocated on them.

```
[mds]# lctl --device 7 activate
[mds]# lctl dl
  0 UP mgs MGS MGS 9
  1 UP mgc MGC192.168.0.10@tcp e384bb0e-680b-ce25-7bc9-816dd1e813 5
  2 UP mdt MDS MDS_uuid 3
  3 UP lov lustre-mdtlov lustre-mdtlov_UUID 4
  4 UP mds lustre-MDT0000 lustre-MDT0000_UUID 5
  5 UP osc lustre-OST0000-osc lustre-mdtlov_UUID 5
  6 UP osc lustre-OST0001-osc lustre-mdtlov_UUID 5
  7 UP osc lustre-OST0002-osc lustre-mdtlov_UUID 5
  8 UP osc lustre-OST0003-osc lustre-mdtlov_UUID 5
  9 UP osc lustre-OST0004-osc lustre-mdtlov_UUID 5
 10 UP osc lustre-OST0005-osc lustre-mdtlov_UUID
```

19.2. Creating and Managing OST Pools

The OST pools feature enables users to group OSTs together to make object placement more flexible. A 'pool' is the name associated with an arbitrary subset of OSTs in a Lustre cluster.

OST pools follow these rules:

- An OST can be a member of multiple pools.
- No ordering of OSTs in a pool is defined or implied.
- Stripe allocation within a pool follows the same rules as the normal stripe allocator.
- OST membership in a pool is flexible, and can change over time.

When an OST pool is defined, it can be used to allocate files. When file or directory striping is set to a pool, only OSTs in the pool are candidates for striping. If a stripe_index is specified which refers to an OST that is not a member of the pool, an error is returned.

OST pools are used only at file creation. If the definition of a pool changes (an OST is added or removed or the pool is destroyed), already-created files are not affected.

Note

An error (`EINVAL`) results if you create a file using an empty pool.

Note

If a directory has pool striping set and the pool is subsequently removed, the new files created in this directory have the (non-pool) default striping pattern for that directory applied and no error is returned.

19.2.1. Working with OST Pools

OST pools are defined in the configuration log on the MGS. Use the lctl command to:

- Create/destroy a pool
- Add/remove OSTs in a pool
- List pools and OSTs in a specific pool

The lctl command MUST be run on the MGS. Another requirement for managing OST pools is to either have the MDT and MGS on the same node or have a Lustre client mounted on the MGS node, if it is separate from the MDS. This is needed to validate the pool commands being run are correct.

Caution

Running the `writeconf` command on the MDS erases all pools information (as well as any other parameters set using `lctl conf_param`). We recommend that the pools definitions (and `conf_param` settings) be executed using a script, so they can be reproduced easily after a `writeconf` is performed.

To create a new pool, run:

```
mgs# lctl pool_new fsname.poolname
```

Note

The pool name is an ASCII string up to 16 characters.

To add the named OST to a pool, run:

```
mgs# lctl pool_add fsname.poolname ost_list
```

Where:

- *ost_list* is *fsname*-OST*index_range*
- *index_range* is *ost_index_start-ost_index_end[,index_range]* or *ost_index_start-ost_index_end/step*

If the leading *fsname* and/or ending _UUID are missing, they are automatically added.

For example, to add even-numbered OSTs to `pool1` on file system `lustre`, run a single command (`pool_add`) to add many OSTs to the pool at one time:

```
lctl pool_add lustre.pool1 OST[0-10/2]
```

Note

Each time an OST is added to a pool, a new `llog` configuration record is created. For convenience, you can run a single command.

To remove a named OST from a pool, run:

```
mgs# lctl pool_remove fsname.poolname ost_list
```

To destroy a pool, run:

```
mgs# lctl pool_destroy fsname.poolname
```

Note

All OSTs must be removed from a pool before it can be destroyed.

To list pools in the named file system, run:

```
mgs# lctl pool_list fsname|pathname
```

To list OSTs in a named pool, run:

```
lctl pool_list fsname.poolname
```

19.2.1.1. Using the lfs Command with OST Pools

Several lfs commands can be run with OST pools. Use the `lfs setstripe` command to associate a directory with an OST pool. This causes all new regular files and directories in the directory to be created in the pool. The lfs command can be used to list pools in a file system and OSTs in a named pool.

To associate a directory with a pool, so all new files and directories will be created in the pool, run:

```
client# lfs setstripe --pool|-p pool_name filename|dirname
```

To set striping patterns, run:

```
client# lfs setstripe [--size|-s stripe_size] [--offset|-o start_ost]
           [--count|-c stripe_count] [--pool|-p pool_name]
           dir|filename
```

Note

If you specify striping with an invalid pool name, because the pool does not exist or the pool name was mistyped, `lfs setstripe` returns an error. Run `lfs pool_list` to make sure the pool exists and the pool name is entered correctly.

Note

The `--pool` option for lfs setstripe is compatible with other modifiers. For example, you can set striping on a directory to use an explicit starting index.

19.2.2. Tips for Using OST Pools

Here are several suggestions for using OST pools.

- A directory or file can be given an extended attribute (EA), that restricts striping to a pool.
- Pools can be used to group OSTs with the same technology or performance (slower or faster), or that are preferred for certain jobs. Examples are SATA OSTs versus SAS OSTs or remote OSTs versus local OSTs.
- A file created in an OST pool tracks the pool by keeping the pool name in the file LOV EA.

19.3. Adding an OST to a Lustre File System

To add an OST to existing Lustre file system:

1. Add a new OST by passing on the following commands, run:

```
oss# mkfs.lustre --fsname=spfs --mgsnode=mds16@tcp0 --ost --index=12 /dev/sda
oss# mkdir -p /mnt/test/ost12
oss# mount -t lustre /dev/sda /mnt/test/ost12
```

2. Migrate the data (possibly).

   The file system is quite unbalanced when new empty OSTs are added. New file creations are automatically balanced. If this is a scratch file system or files are pruned at a regular interval, then no further work may be needed. Files existing prior to the expansion can be rebalanced with an in-place copy, which can be done with a simple script.

   The basic method is to copy existing files to a temporary file, then move the temp file over the old one. This should not be attempted with files which are currently being written to by users or applications. This operation redistributes the stripes over the entire set of OSTs.

   A very clever migration script would do the following:

   - Examine the current distribution of data.
   - Calculate how much data should move from each full OST to the empty ones.
   - Search for files on a given full OST (using `lfs getstripe`).
   - Force the new destination OST (using `lfs setstripe`).
   - Copy only enough files to address the imbalance.

If a Lustre file system administrator wants to explore this approach further, per-OST disk-usage statistics can be found under `/proc/fs/lustre/osc/*/rpc_stats`

19.4. Performing Direct I/O

The Lustre software supports the `O_DIRECT` flag to open.

Applications using the `read()` and `write()` calls must supply buffers aligned on a page boundary (usually 4 K). If the alignment is not correct, the call returns `-EINVAL`. Direct I/O may help performance in cases where the client is doing a large amount of I/O and is CPU-bound (CPU utilization 100%).

19.4.1. Making File System Objects Immutable

An immutable file or directory is one that cannot be modified, renamed or removed. To do this:

`chattr +i` *`file`*

To remove this flag, use `chattr -i`

19.5. Other I/O Options

This section describes other I/O options, including checksums, and the ptlrpcd thread pool.

19.5.1. Lustre Checksums

To guard against network data corruption, a Lustre client can perform two types of data checksums: in-memory (for data in client memory) and wire (for data sent over the network). For each checksum type,

a 32-bit checksum of the data read or written on both the client and server is computed, to ensure that the data has not been corrupted in transit over the network. The `ldiskfs` backing file system does NOT do any persistent checksumming, so it does not detect corruption of data in the OST file system.

The checksumming feature is enabled, by default, on individual client nodes. If the client or OST detects a checksum mismatch, then an error is logged in the syslog of the form:

```
LustreError: BAD WRITE CHECKSUM: changed in transit before arrival at OST: \
from 192.168.1.1@tcp inum 8991479/2386814769 object 1127239/0 extent [10240\
0-106495]
```

If this happens, the client will re-read or re-write the affected data up to five times to get a good copy of the data over the network. If it is still not possible, then an I/O error is returned to the application.

To enable both types of checksums (in-memory and wire), run:

```
lctl set_param llite.*.checksum_pages=1
```

To disable both types of checksums (in-memory and wire), run:

```
lctl set_param llite.*.checksum_pages=0
```

To check the status of a wire checksum, run:

```
lctl get_param osc.*.checksums
```

19.5.1.1. Changing Checksum Algorithms

By default, the Lustre software uses the adler32 checksum algorithm, because it is robust and has a lower impact on performance than crc32. The Lustre file system administrator can change the checksum algorithm via `lctl get_param`, depending on what is supported in the kernel.

To check which checksum algorithm is being used by the Lustre software, run:

```
$ lctl get_param osc.*.checksum_type
```

To change the wire checksum algorithm, run:

```
$ lctl set_param osc.*.checksum_type=algorithm
```

Note

The in-memory checksum always uses the adler32 algorithm, if available, and only falls back to crc32 if adler32 cannot be used.

In the following example, the `lctl get_param` command is used to determine that the Lustre software is using the adler32 checksum algorithm. Then the `lctl set_param` command is used to change the checksum algorithm to crc32. A second `lctl get_param` command confirms that the crc32 checksum algorithm is now in use.

```
$ lctl get_param osc.*.checksum_type
osc.lustre-OST0000-osc-ffff81012b2c48e0.checksum_type=crc32 [adler]
$ lctl set_param osc.*.checksum_type=crc32
osc.lustre-OST0000-osc-ffff81012b2c48e0.checksum_type=crc32
$ lctl get_param osc.*.checksum_type
osc.lustre-OST0000-osc-ffff81012b2c48e0.checksum_type=[crc32] adler
```

19.5.2. Ptlrpc Thread Pool

Releases prior to Lustre software release 2.2 used two portal RPC daemons for each client/server pair. One daemon handled all synchronous IO requests, and the second daemon handled all asynchronous (non-IO) RPCs. The increasing use of large SMP nodes for Lustre servers exposed some scaling issues. The lack of threads for large SMP nodes resulted in cases where a single CPU would be 100% utilized and other CPUs would be relativity idle. This is especially noticeable when a single client traverses a large directory.

Lustre software release 2.2.x implements a ptlrpc thread pool, so that multiple threads can be created to serve asynchronous RPC requests. The number of threads spawned is controlled at module load time using module options. By default one thread is spawned per CPU, with a minimum of 2 threads spawned irrespective of module options.

One of the issues with thread operations is the cost of moving a thread context from one CPU to another with the resulting loss of CPU cache warmth. To reduce this cost, ptlrpc threads can be bound to a CPU. However, if the CPUs are busy, a bound thread may not be able to respond quickly, as the bound CPU may be busy with other tasks and the thread must wait to schedule.

Because of these considerations, the pool of ptlrpc threads can be a mixture of bound and unbound threads. The system operator can balance the thread mixture based on system size and workload.

19.5.2.1. ptlrpcd parameters

These parameters should be set in `/etc/modprobe.conf` or in the `etc/modprobe.d` directory, as options for the ptlrpc module.

```
options ptlrpcd max_ptlrpcds=XXX
```

Sets the number of ptlrpcd threads created at module load time. The default if not specified is one thread per CPU, including hyper-threaded CPUs. The lower bound is 2 (old prlrpcd behaviour)

```
options ptlrpcd ptlrpcd_bind_policy=[1-4]
```

Controls the binding of threads to CPUs. There are four policy options.

- `PDB_POLICY_NONE` (ptlrpcd_bind_policy=1) All threads are unbound.
- `PDB_POLICY_FULL` (ptlrpcd_bind_policy=2) All threads attempt to bind to a CPU.
- `PDB_POLICY_PAIR` (ptlrpcd_bind_policy=3) This is the default policy. Threads are allocated as a bound/unbound pair. Each thread (bound or free) has a partner thread. The partnering is used by the ptlrpcd load policy, which determines how threads are allocated to CPUs.
- `PDB_POLICY_NEIGHBOR` (ptlrpcd_bind_policy=4) Threads are allocated as a bound/unbound pair. Each thread (bound or free) has two partner threads.

Chapter 20. Lustre File System Failover and Multiple-Mount Protection

This chapter describes the multiple-mount protection (MMP) feature, which protects the file system from being mounted simultaneously to more than one node. It includes the following sections:

- Section 20.1, " Overview of Multiple-Mount Protection"
- Section 20.2, "Working with Multiple-Mount Protection"

> **Note**
>
> For information about configuring a Lustre file system for failover, see Chapter 11, *Configuring Failover in a Lustre File System*

20.1. Overview of Multiple-Mount Protection

The multiple-mount protection (MMP) feature protects the Lustre file system from being mounted simultaneously to more than one node. This feature is important in a shared storage environment (for example, when a failover pair of OSSs share a LUN).

The backend file system, `ldiskfs`, supports the MMP mechanism. A block in the file system is updated by a `kmmpd` daemon at one second intervals, and a sequence number is written in this block. If the file system is cleanly unmounted, then a special "clean" sequence is written to this block. When mounting the file system, `ldiskfs` checks if the MMP block has a clean sequence or not.

Even if the MMP block has a clean sequence, `ldiskfs` waits for some interval to guard against the following situations:

- If I/O traffic is heavy, it may take longer for the MMP block to be updated.
- If another node is trying to mount the same file system, a "race" condition may occur.

With MMP enabled, mounting a clean file system takes at least 10 seconds. If the file system was not cleanly unmounted, then the file system mount may require additional time.

> **Note**
>
> The MMP feature is only supported on Linux kernel versions newer than 2.6.9.

20.2. Working with Multiple-Mount Protection

On a new Lustre file system, MMP is automatically enabled by `mkfs.lustre` at format time if failover is being used and the kernel and `e2fsprogs` version support it. On an existing file system, a Lustre file system administrator can manually enable MMP when the file system is unmounted.

Use the following commands to determine whether MMP is running in the Lustre file system and to enable or disable the MMP feature.

To determine if MMP is enabled, run:

```
dumpe2fs -h /dev/block_device | grep mmp
```

Here is a sample command:

```
dumpe2fs -h /dev/sdc | grep mmp
Filesystem features: has_journal ext_attr resize_inode dir_index
filetype extent mmp sparse_super large_file uninit_bg
```

To manually disable MMP, run:

```
tune2fs -O ^mmp /dev/block_device
```

To manually enable MMP, run:

```
tune2fs -O mmp /dev/block_device
```

When MMP is enabled, if `ldiskfs` detects multiple mount attempts after the file system is mounted, it blocks these later mount attempts and reports the time when the MMP block was last updated, the node name, and the device name of the node where the file system is currently mounted.

Chapter 21. Configuring and Managing Quotas

This chapter describes how to configure quotas and includes the following sections:

- Section 21.1, " Working with Quotas"
- Section 21.2, "Enabling Disk Quotas"
- Section 21.3, "Quota Administration"
- Section 21.4, "Quota Allocation"
- Section 21.5, "Interoperability"
- Section 21.6, "Granted Cache and Quota Limits"
- Section 21.7, "Lustre Quota Statistics"

21.1. Working with Quotas

Quotas allow a system administrator to limit the amount of disk space a user or group can use. Quotas are set by root, and can be specified for individual users and/or groups. Before a file is written to a partition where quotas are set, the quota of the creator's group is checked. If a quota exists, then the file size counts towards the group's quota. If no quota exists, then the owner's user quota is checked before the file is written. Similarly, inode usage for specific functions can be controlled if a user over-uses the allocated space.

Lustre quota enforcement differs from standard Linux quota enforcement in several ways:

- Quotas are administered via the `lfs` and `lctl` commands (post-mount).
- Quotas are distributed (as the Lustre file system is a distributed file system), which has several ramifications.
- Quotas are allocated and consumed in a quantized fashion.
- Client does not set the `usrquota` or `grpquota` options to mount. As of Lustre software release 2.4, space accounting is always enabled by default and quota enforcement can be enabled/disabled on a per-file system basis with `lctl conf_param`. It is worth noting that both `lfs quotaon` and `quota_type` are deprecated as of Lustre software release 2.4.0.

Caution

Although a quota feature is available in the Lustre software, root quotas are NOT enforced.

`lfs setquota -u root` (limits are not enforced)

`lfs quota -u root` (usage includes internal Lustre data that is dynamic in size and does not accurately reflect mount point visible block and inode usage).

21.2. Enabling Disk Quotas

Prior to Lustre software release 2.4.0, enabling quota involved a full file system scan via `lfs quotacheck`. All file systems formatted with Lustre software release 2.4.0 or newer no longer require quotacheck to be run since up-to-date accounting information are now always maintained by the OSD layer, regardless of the quota enforcement status.

Introduced in Lustre 2.4

Although quota enforcement is managed by the Lustre software, each OSD implementation relies on the backend file system to maintain per-user/group block and inode usage:

- For ldiskfs backend, `mkfs.lustre` now creates empty quota files and enables the QUOTA feature flag in the superblock which turns quota accounting on at mount time automatically. e2fsck was also modified to fix the quota files when the QUOTA feature flag is present.
- For ZFS backend, accounting ZAPs are created and maintained by the ZFS file system itself. While ZFS tracks per-user and group block usage, it does not handle inode accounting. The ZFS OSD implements its own support for inode tracking. Two options are available:
  1. The ZFS OSD can estimate the number of inodes in-use based on the number of blocks used by a given user or group. This mode can be enabled by running the following command on the server running the target: `lctl set_param osd-zfs.${FSNAME}-${TARGETNAME}.quota_iused_estimate=1`.
  2. Similarly to block accounting, dedicated ZAPs are also created the ZFS OSD to maintain per-user and group inode usage. This is the default mode which corresponds to `quota_iused_estimate` set to 0.

As a result, `lfs quotacheck` is now deprecated and not required any more when running Lustre software release 2.4 on the servers.

Lustre file systems formatted with a Lustre release prior to 2.4.0 can be still safely upgraded to release 2.4.0, but won't have functional space usage report until `tunefs.lustre --quota` is run against all targets. This command sets the QUOTA feature flag in the superblock and runs e2fsck (as a result, the target must be offline) to build the per-UID/GID disk usage database.

Caution

Lustre software release 2.4 and beyond requires a version of e2fsprogs that supports quota (i.e. newer or equal to 1.42.3.wc1) to be installed on the server nodes using ldiskfs backend (e2fsprogs isn't needed with ZFS backend). In general, we recommend to use the latest e2fsprogs version available on http://downloads.hpdd.intel.com/public/e2fsprogs/ [http://downloads.hpdd.intel.com/e2fsprogs/].

The ldiskfs OSD relies on the standard Linux quota to maintain accounting information on disk. As a consequence, the Linux kernel running on the Lustre servers using ldiskfs backend must have `CONFIG_QUOTA`, `CONFIG_QUOTACTL` and `CONFIG_QFMT_V2` enabled.

As of Lustre software release 2.4.0, quota enforcement is thus turned on/off independently of space accounting which is always enabled. `lfs quota`*`on|off`* as well as the per-target `quota_type` parameter are deprecated in favor of a single per-file system quota parameter controlling inode/block quota enforcement. Like all permanent parameters, this quota parameter can be set via `lctl conf_param` on the MGS via the following syntax:

`lctl conf_param` *`fsname`*`.quota.`*`ost|mdt=u|g|ug|none`*

- `ost` -- to configure block quota managed by OSTs
- `mdt` -- to configure inode quota managed by MDTs
- `u` -- to enable quota enforcement for users only
- `g` -- to enable quota enforcement for groups only
- `ug` -- to enable quota enforcement for both users and groups
- `none` -- to disable quota enforcement for both users and groups

Examples:

To turn on user and group quotas for block only on file system `testfs1`, run:

```
$ lctl conf_param testfs1.quota.ost=ug
```

To turn on group quotas for inodes on file system `testfs2`, run:

```
$ lctl conf_param testfs2.quota.mdt=g
```

To turn off user and group quotas for both inode and block on file system `testfs3`, run:

```
$ lctl conf_param testfs3.quota.ost=none

$ lctl conf_param testfs3.quota.mdt=none
```

Once the quota parameter set on the MGS, all targets which are part of the file system will be notified of the new quota settings and enable/disable quota enforcement as needed. The per-target enforcement status can still be verified by running the following command on the Lustre servers:

```
$ lctl get_param osd-*.*.quota_slave.info
osd-zfs.testfs-MDT0000.quota_slave.info=
target name:    testfs-MDT0000
pool ID:        0
type:           md
quota enabled:  ug
conn to master: setup
user uptodate:  glb[1],slv[1],reint[0]
group uptodate: glb[1],slv[1],reint[0]
```

Caution

Lustre software release 2.4 comes with a new quota protocol and a new on-disk format, be sure to check the Interoperability section below (see Section 21.5, “Interoperability”.) when migrating to release 2.4

21.2.2. Enabling Disk Quotas (Lustre Releases Previous to Release 2.4)

Note

In Lustre software releases previous to release 2.4, when new OSTs are added to the file system, quotas are not automatically propagated to the new OSTs. As a workaround, clear and then reset

quotas for each user or group using the `lfs setquota` command. In the example below, quotas are cleared and reset for user `bob` on file system `testfs`:

```
$ lfs setquota -u bob -b 0 -B 0 -i 0 -I 0 /mnt/testfs
$ lfs setquota -u bob -b 307200 -B 309200 -i 10000 -I 11000 /mnt/testfs
```

For Lustre software releases older than release 2.4, `lfs quotacheck` must be first run from a client node to create quota files on the Lustre targets (i.e. the MDT and OSTs). `lfs quotacheck` requires the file system to be quiescent (i.e. no modifying operations like write, truncate, create or delete should run concurrently). Failure to follow this caution may result in inaccurate user/group disk usage. Operations that do not change Lustre files (such as read or mount) are okay to run. `lfs quotacheck` performs a scan on all the Lustre targets to calculates the block/inode usage for each user/group. If the Lustre file system has many files, `quotacheck` may take a long time to complete. Several options can be passed to `lfs quotacheck`:

```
# lfs quotacheck -ug /mnt/testfs
```

- `u` -- checks the user disk quota information
- `g` -- checks the group disk quota information

By default, quota is turned on after `quotacheck` completes. However, this setting isn't persistent and quota will have to be enabled again (via `lfs quotaon`) if one of the Lustre targets is restarted. `lfs quotaoff` is used to turn off quota.

To enable quota permanently with a Lustre software release older than release 2.4, the `quota_type` parameter must be used. This requires setting `mdd.quota_type` and `ost.quota_type`, respectively, on the MDT and OSTs. `quota_type` can be set to the string `u` (user), `g` (group) or `ug` for both users and groups. This parameter can be specified at `mkfs` time (`mkfs.lustre --param mdd.quota_type=ug`) or with `tunefs.lustre`. As an example:

```
tunefs.lustre --param ost.quota_type=ug $ost_dev
```

When using `mkfs.lustre --param mdd.quota_type=ug` or `tunefs.lustre --param ost.quota_type=ug`, be sure to run the command on all OSTs and the MDT. Otherwise, abnormal results may occur.

21.3. Quota Administration

Once the file system is up and running, quota limits on blocks and files can be set for both user and group. This is controlled via three quota parameters:

Grace period -- The period of time (in seconds) within which users are allowed to exceed their soft limit. There are four types of grace periods:

- user block soft limit
- user inode soft limit
- group block soft limit
- group inode soft limit

The grace period applies to all users. The user block soft limit is for all users who are using a blocks quota.

Soft limit -- The grace timer is started once the soft limit is exceeded. At this point, the user/group can still allocate block/inode. When the grace time expires and if the user is still above the soft limit, the soft

limit becomes a hard limit and the user/group can't allocate any new block/inode any more. The user/group should then delete files to be under the soft limit. The soft limit MUST be smaller than the hard limit. If the soft limit is not needed, it should be set to zero (0).

Hard limit -- Block or inode allocation will fail with `EDQUOT` (i.e. quota exceeded) when the hard limit is reached. The hard limit is the absolute limit. When a grace period is set, one can exceed the soft limit within the grace period if under the hard limit.

Due to the distributed nature of a Lustre file system and the need to mainain performance under load, those quota parameters may not be 100% accurate. The quota settings can be manipulated via the `lfs` command which includes several options to work with quotas:

- `quota` -- displays general quota information (disk usage and limits)
- `setquota` -- specifies quota limits and tunes the grace period. By default, the grace period is one week.

Usage:

```
lfs quota [-q] [-v] [-h] [-o obd_uuid] [-u|-g uname|uid|gname|gid]  /mount_point
lfs quota -t -u|-g /mount_point
lfs setquota -u|--user|-g|--group username|groupname [-b block-softlimit] \
             [-B block_hardlimit] [-i inode_softlimit] \
             [-I inode_hardlimit] /mount_point
```

To display general quota information (disk usage and limits) for the user running the command and his primary group, run:

```
$ lfs quota /mnt/testfs
```

To display general quota information for a specific user ("`bob`" in this example), run:

```
$ lfs quota -u bob /mnt/testfs
```

To display general quota information for a specific user ("`bob`" in this example) and detailed quota statistics for each MDT and OST, run:

```
$ lfs quota -u bob -v /mnt/testfs
```

To display general quota information for a specific group ("`eng`" in this example), run:

```
$ lfs quota -g eng /mnt/testfs
```

To display block and inode grace times for user quotas, run:

```
$ lfs quota -t -u /mnt/testfs
```

To set user or group quotas for a specific ID ("bob" in this example), run:

```
$ lfs setquota -u bob -b 307200 -B 309200 -i 10000 -I 11000 /mnt/testfs
```

In this example, the quota for user "bob" is set to 300 MB (309200*1024) and the hard limit is 11,000 files. Therefore, the inode hard limit should be 11000.

The quota command displays the quota allocated and consumed by each Lustre target. Using the previous `setquota` example, running this `lfs` quota command:

```
$ lfs quota -u bob -v /mnt/testfs
```

displays this command output:

```
Disk quotas for user bob (uid 6000):
Filesystem          kbytes quota limit grace files quota limit grace
/mnt/testfs         0      30720 30920 -     0     10000 11000 -
testfs-MDT0000_UUID 0      -      8192 -     0     -     2560  -
testfs-OST0000_UUID 0      -      8192 -     0     -     0     -
testfs-OST0001_UUID 0      -      8192 -     0     -     0     -
Total allocated inode limit: 2560, total allocated block limit: 24576
```

Global quota limits are stored in dedicated index files (there is one such index per quota type) on the quota master target (aka QMT). The QMT runs on MDT0000 and exports the global indexes via /proc. The global indexes can thus be dumped via the following command:

```
# lctl get_param qmt.testfs-QMT0000.*.glb-*
```

The format of global indexes depends on the OSD type. The ldiskfs OSD uses an IAM files while the ZFS OSD creates dedicated ZAPs.

Each slave also stores a copy of this global index locally. When the global index is modified on the master, a glimpse callback is issued on the global quota lock to notify all slaves that the global index has been modified. This glimpse callback includes information about the identifier subject to the change. If the global index on the QMT is modified while a slave is disconnected, the index version is used to determine whether the slave copy of the global index isn't uptodate any more. If so, the slave fetches the whole index again and updates the local copy. The slave copy of the global index is also exported via /proc and can be accessed via the following command:

```
lctl get_param osd-*.*.quota_slave.limit*
```

Note

Prior to 2.4, global quota limits used to be stored in administrative quota files using the on-disk format of the linux quota file. When upgrading MDT0000 to 2.4, those administrative quota files are converted into IAM indexes automatically, conserving existing quota limits previously set by the administrator.

21.4. Quota Allocation

In a Lustre file system, quota must be properly allocated or users may experience unnecessary failures. The file system block quota is divided up among the OSTs within the file system. Each OST requests an allocation which is increased up to the quota limit. The quota allocation is then *quantized* to reduce the number of quota-related request traffic.

The Lustre quota system distributes quotas from the Quota Master Target (aka QMT). Only one QMT instance is supported for now and only runs on the same node as MDT0000. All OSTs and MDTs set up a Quota Slave Device (aka QSD) which connects to the QMT to allocate/release quota space. The QSD is setup directly from the OSD layer.

To reduce quota requests, quota space is initially allocated to QSDs in very large chunks. How much unused quota space can be hold by a target is controlled by the qunit size. When quota space for a given ID is close to exhaustion on the QMT, the qunit size is reduced and QSDs are notified of the new qunit size value via a glimpse callback. Slaves are then responsible for releasing quota space above the new qunit value. The qunit size isn't shrunk indefinitely and there is a minimal value of 1MB for blocks and 1,024 for inodes. This means that the quota space rebalancing process will stop when this mininum value

is reached. As a result, quota exceeded can be returned while many slaves still have 1MB or 1,024 inodes of spare quota space.

If we look at the `setquota` example again, running this `lfs quota` command:

```
# lfs quota -u bob -v /mnt/testfs
```

displays this command output:

```
Disk quotas for user bob (uid 500):
Filesystem          kbytes quota limit grace       files  quota limit grace
/mnt/testfs         30720* 30720 30920 6d23h56m44s 10101* 10000 11000 6d23h59m50s
testfs-MDT0000_UUID 0      -     0     -           10101  -     10240
testfs-OST0000_UUID 0      -     1024  -           -      -     -
testfs-OST0001_UUID 30720* -     29896 -           -      -     -
Total allocated inode limit: 10240, total allocated block limit: 30920
```

The total quota limit of 30,920 is allocated to user bob, which is further distributed to two OSTs.

Values appended with '*' show that the quota limit has been exceeded, causing the following error when trying to write or create a file:

```
$ cp: writing `/mnt/testfs/foo`: Disk quota exceeded.
```

Note

It is very important to note that the block quota is consumed per OST and the inode quota per MDS. Therefore, when the quota is consumed on one OST (resp. MDT), the client may not be able to create files regardless of the quota available on other OSTs (resp. MDTs).

Setting the quota limit below the minimal qunit size may prevent the user/group from all file creation. It is thus recommended to use soft/hard limits which are a multiple of the number of OSTs * the minimal qunit size.

To determine the total number of inodes, use `lfs df -i` (and also `lctl get_param *.*.filestotal`). For more information on using the `lfs df -i` command and the command output, see Section 18.5.1, “Checking File System Free Space”.

Unfortunately, the `statfs` interface does not report the free inode count directly, but instead reports the total inode and used inode counts. The free inode count is calculated for `df` from (total inodes - used inodes). It is not critical to know the total inode count for a file system. Instead, you should know (accurately), the free inode count and the used inode count for a file system. The Lustre software manipulates the total inode count in order to accurately report the other two values.

21.5. Interoperability

The new quota protocol introduced in Lustre software release 2.4.0 **isn't compatible** with the old one. As a consequence, **all Lustre servers must be upgraded to release 2.4.0 for quota to be functional**. Quota limits set on the Lustre file system prior to the upgrade will be automatically migrated to the new quota index format. As for accounting information with ldiskfs backend, they will be regenerated by running `tunefs.lustre --quota` against all targets. It is worth noting that running `tunefs.lustre --quota` is **mandatory** for all targets formatted with a Lustre software release older than release 2.4.0, otherwise quota enforcement as well as accounting won't be functional.

Besides, the quota protocol in release 2.4 takes for granted that the Lustre client supports the `OBD_CONNECT_EINPROGRESS` connect flag. Clients supporting this flag will retry indefinitely when

the server returns `EINPROGRESS` in a reply. Here is the list of Lustre client version which are compatible with release 2.4:

- Release 2.3-based clients and beyond
- Release 1.8 clients newer or equal to release 1.8.9-wc1
- Release 2.1 clients newer or equal to release 2.1.4

21.6. Granted Cache and Quota Limits

In a Lustre file system, granted cache does not respect quota limits. In this situation, OSTs grant cache to a Lustre client to accelerate I/O. Granting cache causes writes to be successful in OSTs, even if they exceed the quota limits, and will overwrite them.

The sequence is:

1. A user writes files to the Lustre file system.
2. If the Lustre client has enough granted cache, then it returns 'success' to users and arranges the writes to the OSTs.
3. Because Lustre clients have delivered success to users, the OSTs cannot fail these writes.

Because of granted cache, writes always overwrite quota limitations. For example, if you set a 400 GB quota on user A and use IOR to write for user A from a bundle of clients, you will write much more data than 400 GB, and cause an out-of-quota error (`EDQUOT`).

Note

The effect of granted cache on quota limits can be mitigated, but not eradicated. Reduce the maximum amount of dirty data on the clients (minimal value is 1MB):

- `lctl set_param osc.*.max_dirty_mb=8`

21.7. Lustre Quota Statistics

The Lustre software includes statistics that monitor quota activity, such as the kinds of quota RPCs sent during a specific period, the average time to complete the RPCs, etc. These statistics are useful to measure performance of a Lustre file system.

Each quota statistic consists of a quota event and `min_time`, `max_time` and `sum_time` values for the event.

Quota Event	Description
sync_acq_req	Quota slaves send a acquiring_quota request and wait for its return.
sync_rel_req	Quota slaves send a releasing_quota request and wait for its return.
async_acq_req	Quota slaves send an acquiring_quota request and do not wait for its return.
async_rel_req	Quota slaves send a releasing_quota request and do not wait for its return.

Quota Event	Description
wait_for_blk_quota (lquota_chkquota)	Before data is written to OSTs, the OSTs check if the remaining block quota is sufficient. This is done in the lquota_chkquota function.
wait_for_ino_quota (lquota_chkquota)	Before files are created on the MDS, the MDS checks if the remaining inode quota is sufficient. This is done in the lquota_chkquota function.
wait_for_blk_quota (lquota_pending_commit)	After blocks are written to OSTs, relative quota information is updated. This is done in the lquota_pending_commit function.
wait_for_ino_quota (lquota_pending_commit)	After files are created, relative quota information is updated. This is done in the lquota_pending_commit function.
wait_for_pending_blk_quota_req (qctxt_wait_pending_dqacq)	On the MDS or OSTs, there is one thread sending a quota request for a specific UID/GID for block quota at any time. At that time, if other threads need to do this too, they should wait. This is done in the qctxt_wait_pending_dqacq function.
wait_for_pending_ino_quota_req (qctxt_wait_pending_dqacq)	On the MDS, there is one thread sending a quota request for a specific UID/GID for inode quota at any time. If other threads need to do this too, they should wait. This is done in the qctxt_wait_pending_dqacq function.
nowait_for_pending_blk_quota_req (qctxt_wait_pending_dqacq)	On the MDS or OSTs, there is one thread sending a quota request for a specific UID/GID for block quota at any time. When threads enter qctxt_wait_pending_dqacq, they do not need to wait. This is done in the qctxt_wait_pending_dqacq function.
nowait_for_pending_ino_quota_req (qctxt_wait_pending_dqacq)	On the MDS, there is one thread sending a quota request for a specific UID/GID for inode quota at any time. When threads enter qctxt_wait_pending_dqacq, they do not need to wait. This is done in the qctxt_wait_pending_dqacq function.
quota_ctl	The quota_ctl statistic is generated when lfs `setquota`, `lfs quota` and so on, are issued.
adjust_qunit	Each time qunit is adjusted, it is counted.

21.7.1. Interpreting Quota Statistics

Quota statistics are an important measure of the performance of a Lustre file system. Interpreting these statistics correctly can help you diagnose problems with quotas, and may indicate adjustments to improve system performance.

For example, if you run this command on the OSTs:

```
lctl get_param lquota.testfs-OST0000.stats
```

You will get a result similar to this:

```
snapshot_time                             1219908615.506895 secs.usecs
async_acq_req                           1 samples [us]  32 32 32
async_rel_req                           1 samples [us]  5 5 5
nowait_for_pending_blk_quota_req(qctxt_wait_pending_dqacq) 1 samples [us] 2\
 2 2
quota_ctl                         4 samples [us]  80 3470 4293
adjust_qunit                            1 samples [us]  70 70 70
....
```

In the first line, `snapshot_time` indicates when the statistics were taken. The remaining lines list the quota events and their associated data.

In the second line, the `async_acq_req` event occurs one time. The `min_time`, `max_time` and `sum_time` statistics for this event are 32, 32 and 32, respectively. The unit is microseconds (µs).

In the fifth line, the quota_ctl event occurs four times. The `min_time`, `max_time` and `sum_time` statistics for this event are 80, 3470 and 4293, respectively. The unit is microseconds (µs).

Introduced in Lustre 2.5

Chapter 22. Hierarchical Storage Management (HSM)

This chapter describes how to bind Lustre to a Hierarchical Storage Management (HSM) solution.

22.1. Introduction

The Lustre file system can bind to a Hierarchical Storage Management (HSM) solution using a specific set of functions. These functions enable connecting a Lustre file system to one or more external storage systems, typically HSMs. With a Lustre file system bound to a HSM solution, the Lustre file system acts as a high speed cache in front of these slower HSM storage systems.

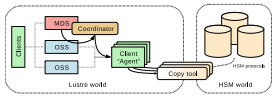
The Lustre file system integration with HSM provides a mechanism for files to simultaneously exist in a HSM solution and have a metadata entry in the Lustre file system that can be examined. Reading, writing or truncating the file will trigger the file data to be fetched from the HSM storage back into the Lustre file system.

The process of copying a file into the HSM storage is known as *archive*. Once the archive is complete, the Lustre file data can be delete (know as *release*.) The process of returning data from the HSM storage to the Lustre file system is called *restore*. The archive and restore operations require a Lustre file system component called an *Agent*.

An Agent is a specially designed Lustre client node that mounts the Lustre file system in question. On an Agent, a user space program called a copytool is run to coordinate the archive and restore of files between the Lustre file system and the HSM solution.

Requests to restore a given file are registered and dispatched by a facet on the MDT called the Coordinator.

Figure 22.1. Overview of the Lustre file system HSM



22.2. Setup

22.2.1. Requirements

To setup a Lustre/HSM configuration you need:

- a standard Lustre file system (version 2.5.0 and above)

- a minimum of 2 clients, 1 used for your chosen computation task that generates useful data, and 1 used as agent.

Multiple agents can be employed. All the agents need to share access to their backend storage. For the POSIX copytool, a POSIX namespace like NFS or another Lustre file system is suitable.

22.2.2. Coordinator

To bind a Lustre file system to a HSM system a coordinator must be activated on each of your filesystem MDTs. This can be achieved with the command:

```
$ lctl set_param mdt.$FSNAME-MDT0000.hsm_control=enabled
mdt.lustre-MDT0000.hsm_control=enabled
```

To verify if the coordinator is running correctly

```
$ lctl get_param mdt.$FSNAME-MDT0000.hsm_control
mdt.lustre-MDT0000.hsm_control=enabled
```

22.2.3. Agents

Once a coordinator is started launch the copytool on each agent node to connect to your HSM storage. If your HSM storage has POSIX access this command will be of the form:

```
lhsmtool_posix --daemon --hsm-root $HSMPATH --archive=1 $LUSTREPATH
```

POSIX copytool must be stopped sending it a TERM signal.

22.3. Agents and copytool

Agents are Lustre file system clients running copytool. copytool is a userspace daemon that transfers data between Lustre and a HSM solution. Because different HSM solutions use different APIs, copytools can typically only work with a specific HSM. Only one copytool could be run by agent node.

The following rule applies regarding copytool instances: a Lustre file system only supports a single copytool process, per ARCHIVE ID (see below), per client node. Due to a Lustre software limitation, this constraint is irrespective of the number of Lustre file systems mounted by the Agent.

Bundled with Lustre tools, the POSIX copytool can work with any HSM or external storage that exports a POSIX API.

22.3.1. Archive ID, multiple backends

A Lustre file system can be bound to several different HSM solutions. Each bound HSM solution is identified by a number referred to as ARCHIVE ID. A unique value of ARCHIVE ID must be chosen for each bound HSM solution. ARCHIVE ID must be in the range 1 to 32.

A Lustre file system supports an unlimited number of copytool instances. You need, at least, one copytool per ARCHIVE ID. When using the POSIX copytool, this ID is defined using `--archive` switch.

For example: if a single Lustre file system is bound to 2 different HSMs (A and B,) ARCHIVE ID “1” can be chosen for HSM A and ARCHIVE ID “2” for HSM B. If you start 3 copytool instances for ARCHIVE ID 1, all of them will use Archive ID “1”. Same rule applies for copytool instances dealing with the HSM B, using Archive ID “2”.

When issuing HSM requests, you can use `--archive` switch to choose the backend you want to use. In this example, file `foo` will be archived into backend ARCHIVE ID “5”:

```
$ lfs hsm_archive --archive=5 /mnt/lustre/foo
```

A default ARCHIVE ID can be defined when this switch is not specified:

```
$ lctl set_param -P mdt.lustre-MDT0000.hsm.default_archive_id=5
```

The ARCHIVE ID of archived files can be checked using `lfs hsm_state` command:

```
$ lfs hsm_state /mnt/lustre/foo
/mnt/lustre/foo: (0x00000009) exists archived, archive_id:5
```

22.3.2. Registered agents

A Lustre file system allocates a unique UUID per client mount point, for each filesystem. Only one copytool can be registered for each Lustre mount point. As a consequence the UUID uniquely identifies a copytool, per filesystem.

The currently registered copytool instances (agents UUID) can be retrieved running the following command, per MDT, on MDS nodes:

```
$ lctl get_param -n mdt.$FSNAME-MDT0000.hsm.agents
uuid=a19b2416-0930-fc1f-8c58-c985ba5127ad archive_id=1 requests=[current:0
```

The returned fields have the following meaning:

- `UUID` the client mount used by the corresponding copytool.
- `archive_id` comma-separated list of ARCHIVE ID accessible by this copytool.
- `requests` various statistics of number of processed requests by this copytool.

22.3.3. Timeout

One or more copytool instances may experience conditions that cause them to become unresponsive. To avoid blocking access to the related files a timeout value is defined for request processing. A copytool must be able to fully complete a request within this time. The default is 3600 seconds.

```
$ lctl set_param -n mdt.lustre-MDT0000.hsm.active_request_timeout
```

22.4. Requests

Data management between a Lustre file system and HSM solutions is driven by requests. There are tfive types:

- `ARCHIVE` Copy data from a Lustre file system file into the HSM solution.
- `RELEASE` Remove file data from the Lustre file system.

- `RESTORE` Copy back data from the HSM solution into the corresponding Lustre file system file.
- `REMOVE` Delete the copy of the data from the HSM solution.
- `CANCEL` Cancel an undergoing or pending request.

Only the `RELEASE` is performed synchronously and does not involve the coordinator. Other requests are handled by Coordinators. Each MDT coordinator is resiliently managing them.

22.4.1. Commands

Requests are submitted using `lfs` command:

```
$ lfs hsm_archive [--archive=ID] FILE1 [FILE2...]
$ lfs hsm_release FILE1 [FILE2...]
$ lfs hsm_restore FILE1 [FILE2...]
$ lfs hsm_remove  FILE1 [FILE2...]
```

Requests are sent by default to the default ARCHIVE ID or the specified one (See Section 22.3.1, “ Archive ID, multiple backends ”.)

22.4.2. Automatic restore

Released files are automatically restored when a process tries to read or modify them. The corresponding I/O will block waiting for the file to be restored. This is transparent to the process. For example, the following command automatically restores the file if released.

```
$ cat /mnt/lustre/released_file
```

22.4.3. Request monitoring

The list of registered requests and their status can be monitored, per MDT, with the following command:

```
$ lctl get_param -n mdt.lustre-MDT0000.hsm.actions
```

The list of request currently being processed by a copytool is available with:

```
$ lctl get_param -n mdt.lustre-MDT0000.hsm.active_requests
```

22.5. File states

When files are archived or released, their state in the Lustre file system changes. This state can be read using the following `lfs` command:

```
$ lfs hsm_state FILE1 [FILE2...]
```

There is also a list of specific policy flags which could be set to have a per-file specific policy:

- `NOARCHIVE` This file will never be archived.
- `NORELEASE` This file will never be released. This value cannot be set if the flag is currently set to `RELEASED`

- `DIRTY` This file has been modified since a copy of it was made in the HSM solution. `DIRTY` files should be archived again. The `DIRTY` flag can only be set if `EXIST` is set.

The following options can only be set by root user.

- `LOST` This file previously archived but the copy was lost on the HSM solution for some reasons in the backend (for example, by a corrupted tape), and could not be restored. If the file is not in the state of `RELEASE` it needs to be archived again. If the file state is in `RELEASE`, file data is lost.

Some flags can be manually set or cleared using the following commands:

```
$ lfs hsm_set [FLAGS] FILE1 [FILE2...]
$ lfs hsm_clear [FLAGS] FILE1 [FILE2...]
```

22.6. Tuning

22.6.1. hsm_controlpolicy

`hsm_control` controls coordinator activity and can also purge the action list.

```
$ lctl set_param mdt.$FSNAME-MDT0000.hsm_control=purge
```

Possible values are:

- `enabled` Start coordinator thread. Requests are dispatched on available copytool instances.
- `disabled` Pause coordinator activity. No new request will be scheduled. No timeout will be handled. New requests will be registered but will be handled only when the coordinator is enabled again.
- `shutdown` Stop coordinator thread. No request could be submitted.
- `purge` Clear all recorded requests. Do not change coordinator state.

22.6.2. max_requests

`max_requests` is the maximum number of active requests at the same time. This is a per coordinator value, and independent of the number of agents.

For example, if 2 MDT and 4 agents are present, the agents will never have to handle more than 2 x `max_requests`.

```
$ lctl set_param mdt.$FSNAME-MDT0000.hsm.max_requests=10
```

22.6.3. policy

Change system behavior. Value could be combined or removed prefixing them by '+' or '-'.

```
$ lctl set_param mdt.$FSNAME-MDT0000.hsm.policy=+NRA
```

Possible values are a combination of:

- `NRA` No Retry Action. If a restore fails, do not reschedule it automatically.

- `NBR` Non Blocking Restore. No automatic restore is triggered. Access to a released file returns `ENODATA`.

22.6.4. `grace_delay`

`grace_delay` is the delay, expressed in seconds, before a successful or failed request is cleared from the whole request list.

```
$ lctl set_param mdt.$FSNAME-MDT0000.hsm.grace_delay=10
```

22.7. change logs

A changelog record type “HSM“ was added for Lustre file system logs that relate to HSM events.

```
16HSM   13:49:47.469433938 2013.10.01 0x280 t=[0x200000400:0x1:0x0]
```

Two items of information are available for each HSM record: the FID of the modified file and a bit mask. The bit mask codes the following information (lowest bits first):

- Error code, if any (7 bits)
- HSM event (3 bits)
  - `HE_ARCHIVE = 0` File has been archived.
  - `HE_ARCHIVE = 1` File has been restored.
  - `HE_ARCHIVE = 2` A request for this file has been canceled.
  - `HE_ARCHIVE = 3` File has been released.
  - `HE_ARCHIVE = 4` A remove request has been executed automatically.
  - `HE_ARCHIVE = 5` File flags have changed.
- HSM flags (3 bits)
  - `CLF_HSM_DIRTY=0x1`

In the above example, `0x280` means error code is 0 and event is HE_STATE.

When using `liblustreapi`, there is a list of helper functions to easily extract the different values from this bitmask, like: `hsm_get_cl_event()`, `hsm_get_cl_flags()`, `hsm_get_cl_error()`

22.8. Policy engine

A Lustre file system does not have an internal component responsible for automatically scheduling archive requests and release requests under any conditions (like low space). Automatically scheduling archive operations is the role of the policy engine.

It is recommended that the Policy Engine runs on a dedicated client, similar to an agent node, with a Lustre version 2.5+.

A policy engine is a userspace program using the Lustre file system HSM specific API to monitor the file system and schedule requests.

Robinhood is the recommended policy engine.

22.8.1. Robinhood

Robinhood is a Policy engine and reporting tool for large file systems. It maintains a replicate of file system medatada in a database that can be queried at will. Robinhood makes it possible to schedule mass action on file system entries by defining attribute-based policies, provides fast find and `du` enhanced clones, gives to administrators an overall view of file system content through a web user interface and command line tools.

Robinhood can be used for various configuration. Robinhood is an external project and further information can be found on the website: https://sourceforge.net/apps/trac/robinhood/wiki/Doc.

Chapter 23. Managing Security in a Lustre File System

This chapter describes security features of the Lustre file system and includes the following sections:

- Section 23.1, “Using ACLs”
- Section 23.2, “Using Root Squash”

23.1. Using ACLs

An access control list (ACL), is a set of data that informs an operating system about permissions or access rights that each user or group has to specific system objects, such as directories or files. Each object has a unique security attribute that identifies users who have access to it. The ACL lists each object and user access privileges such as read, write or execute.

23.1.1. How ACLs Work

Implementing ACLs varies between operating systems. Systems that support the Portable Operating System Interface (POSIX) family of standards share a simple yet powerful file system permission model, which should be well-known to the Linux/UNIX administrator. ACLs add finer-grained permissions to this model, allowing for more complicated permission schemes. For a detailed explanation of ACLs on a Linux operating system, refer to the SUSE Labs article, *Posix Access Control Lists on Linux*:

http://www.suse.de/~agruen/acl/linux-acls/online/

We have implemented ACLs according to this model. The Lustre software works with the standard Linux ACL tools, setfacl, getfacl, and the historical chacl, normally installed with the ACL package.

Note

ACL support is a system-range feature, meaning that all clients have ACL enabled or not. You cannot specify which clients should enable ACL.

23.1.2. Using ACLs with the Lustre Software

POSIX Access Control Lists (ACLs) can be used with the Lustre software. An ACL consists of file entries representing permissions based on standard POSIX file system object permissions that define three classes of user (owner, group and other). Each class is associated with a set of permissions [read (r), write (w) and execute (x)].

- Owner class permissions define access privileges of the file owner.
- Group class permissions define access privileges of the owning group.
- Other class permissions define access privileges of all users not in the owner or group class.

The `ls -l` command displays the owner, group, and other class permissions in the first column of its output (for example, `-rw-r- --` for a regular file with read and write access for the owner class, read access for the group class, and no access for others).

Minimal ACLs have three entries. Extended ACLs have more than the three entries. Extended ACLs also contain a mask entry and may contain any number of named user and named group entries.

The MDS needs to be configured to enable ACLs. Use `--mountfsoptions` to enable ACLs when creating your configuration:

```
$ mkfs.lustre --fsname spfs --mountfsoptions=acl --mdt -mgs /dev/sda
```

Alternately, you can enable ACLs at run time by using the `--acl` option with `mkfs.lustre`:

```
$ mount -t lustre -o acl /dev/sda /mnt/mdt
```

To check ACLs on the MDS:

```
$ lctl get_param -n mdc.home-MDT0000-mdc-*.connect_flags | grep acl acl
```

To mount the client with no ACLs:

```
$ mount -t lustre -o noacl ibmds2@o2ib:/home /home
```

ACLs are enabled in a Lustre file system on a system-wide basis; either all clients enable ACLs or none do. Activating ACLs is controlled by MDS mount options `acl` / `noacl` (enable/disable ACLs). Client-side mount options acl/noacl are ignored. You do not need to change the client configuration, and the 'acl' string will not appear in the client /etc/mtab. The client acl mount option is no longer needed. If a client is mounted with that option, then this message appears in the MDS syslog:

```
...MDS requires ACL support but client does not
```

The message is harmless but indicates a configuration issue, which should be corrected.

If ACLs are not enabled on the MDS, then any attempts to reference an ACL on a client return an Operation not supported error.

23.1.3. Examples

These examples are taken directly from the POSIX paper referenced above. ACLs on a Lustre file system work exactly like ACLs on any Linux file system. They are manipulated with the standard tools in the standard manner. Below, we create a directory and allow a specific user access.

```
[root@client lustre]# umask 027
[root@client lustre]# mkdir rain
[root@client lustre]# ls -ld rain
drwxr-x---  2 root root 4096 Feb 20 06:50 rain
[root@client lustre]# getfacl rain
# file: rain
# owner: root
# group: root
user::rwx
group::r-x
other::---

[root@client lustre]# setfacl -m user:chirag:rwx rain
[root@client lustre]# ls -ld rain
drwxrwx---+ 2 root root 4096 Feb 20 06:50 rain
[root@client lustre]# getfacl --omit-header rain
user::rwx
```

```
user:chirag:rwx
group::r-x
mask::rwx
other::---
```

23.2. Using Root Squash

Root squash is a security feature which restricts super-user access rights to a Lustre file system. Without the root squash feature enabled, Lustre file system users on untrusted clients could access or modify files owned by root on the file system, including deleting them. Using the root squash feature restricts file access/modifications as the root user to only the specified clients. Note, however, that this does *not* prevent users on insecure clients from accessing files owned by *other* users.

The root squash feature works by re-mapping the user ID (UID) and group ID (GID) of the root user to a UID and GID specified by the system administrator, via the Lustre configuration management server (MGS). The root squash feature also enables the Lustre fle system administrator to specify a set of client for which UID/GID re-mapping does not apply.

23.2.1. Configuring Root Squash

Root squash functionality is managed by two configuration parameters, `root_squash` and `nosquash_nids`.

- The `root_squash` parameter specifies the UID and GID with which the root user accesses the Lustre file system.
- The `nosquash_nids` parameter specifies the set of clients to which root squash does not apply. LNET NID range syntax is used for this parameter (see the NID range syntax rules described in Section 23.2.2, “Enabling and Tuning Root Squash”). For example:

```
nosquash_nids=172.16.245.[0-255/2]@tcp
```

In this example, root squash does not apply to TCP clients on subnet 172.16.245.0 that have an even number as the last component of their IP address.

23.2.2. Enabling and Tuning Root Squash

The default value for `nosquash_nids` is NULL, which means that root squashing applies to all clients. Setting the root squash UID and GID to 0 turns root squash off.

Root squash parameters can be set when the MDT is created (`mkfs.lustre --mdt`). For example:

```
mds# mkfs.lustre --reformat --fsname=testfs --mdt --mgs \
       --param "mdt.root_squash=500:501" \
       --param "mdt.nosquash_nids='0@elan1 192.168.1.[10,11]'" /dev/sda1
```

Root squash parameters can also be changed on an unmounted device with `tunefs.lustre`. For example:

```
tunefs.lustre --param "mdt.root_squash=65534:65534"  \
--param "mdt.nosquash_nids=192.168.0.13@tcp0" /dev/sda1
```

Root squash parameters can also be changed with the `lctl conf_param` command. For example:

```
mgs# lctl conf_param testfs.mdt.root_squash="1000:101"
```

```
mgs# lctl conf_param testfs.mdt.nosquash_nids="*@tcp"
```

Note

When using the lctl conf_param command, keep in mind:

- `lctl conf_param` must be run on a live MGS
- `lctl conf_param` causes the parameter to change on all MDSs
- `lctl conf_param` is to be used once per a parameter

The `nosquash_nids` list can be cleared with:

```
mgs# lctl conf_param testfs.mdt.nosquash_nids="NONE"
```

- OR -

```
mgs# lctl conf_param testfs.mdt.nosquash_nids="clear"
```

If the `nosquash_nids` value consists of several NID ranges (e.g. `0@elan`, `1@elan1`), the list of NID ranges must be quoted with single (') or double (") quotation marks. List elements must be separated with a space. For example:

```
mds# mkfs.lustre ... --param "mdt.nosquash_nids='0@elan1 1@elan2'" /dev/sda1
lctl conf_param testfs.mdt.nosquash_nids="24@elan 15@elan1"
```

These are examples of incorrect syntax:

```
mds# mkfs.lustre ... --param "mdt.nosquash_nids=0@elan1 1@elan2" /dev/sda1
lctl conf_param testfs.mdt.nosquash_nids=24@elan 15@elan1
```

To check root squash parameters, use the lctl get_param command:

```
mds# lctl get_param mdt.testfs-MDT0000.root_squash
lctl get_param mdt.*.nosquash_nids
```

Note

An empty nosquash_nids list is reported as NONE.

23.2.3. Tips on Using Root Squash

Lustre configuration management limits root squash in several ways.

- The `lctl conf_param` value overwrites the parameter's previous value. If the new value uses an incorrect syntax, then the system continues with the old parameters and the previously-correct value is lost on remount. That is, be careful doing root squash tuning.
- `mkfs.lustre` and `tunefs.lustre` do not perform parameter syntax checking. If the root squash parameters are incorrect, they are ignored on mount and the default values are used instead.
- Root squash parameters are parsed with rigorous syntax checking. The root_squash parameter should be specified as `<decnum>:<decnum>`. The `nosquash_nids` parameter should follow LNET NID range list syntax.

LNET NID range syntax:

```
<nidlist>     :== <nidrange> [ ' ' <nidrange> ]
<nidrange>   :== <addrrange> '@' <net>
<addrrange>  :== '*' |
           <ipaddr_range> |
           <numaddr_range>
<ipaddr_range>       :==
<numaddr_range>.<numaddr_range>.<numaddr_range>.<numaddr_range>
<numaddr_range>      :== <number> |
                   <expr_list>
<expr_list>  :== '[' <range_expr> [ ',' <range_expr>] ']'
<range_expr> :== <number> |
           <number> '-' <number> |
           <number> '-' <number> '/' <number>
<net>          :== <netname> | <netname><number>
<netname>    :== "lo" | "tcp" | "o2ib" | "cib" | "openib" | "iib" |
           "vib" | "ra" | "elan" | "gm" | "mx" | "ptl"
<number>     :== <nonnegative decimal> | <hexadecimal>
```

Note

For networks using numeric addresses (e.g. elan), the address range must be specified in the `<numaddr_range>` syntax. For networks using IP addresses, the address range must be in the `<ipaddr_range>`. For example, if elan is using numeric addresses, `1.2.3.4@elan` is incorrect.

Part IV. Tuning a Lustre File System for Performance

Part IV describes tools and procedures used to tune a Lustre file system for optimum performance. You will find information in this section about:

- Testing Lustre Network Performance (LNET Self-Test)
- Benchmarking Lustre File System Performance (Lustre I/O Kit)
- Tuning a Lustre File System

Chapter 24. Testing Lustre Network Performance (LNET Self-Test)

This chapter describes the LNET self-test, which is used by site administrators to confirm that Lustre networking (LNET) has been properly installed and configured, and that underlying network software and hardware are performing according to expectations. The chapter includes:

- Section 24.1, " LNET Self-Test Overview"
- Section 24.2, "Using LNET Self-Test"
- Section 24.3, "LNET Self-Test Command Reference"

24.1. LNET Self-Test Overview

LNET self-test is a kernel module that runs over LNET and the Lustre network drivers (LNDs). It is designed to:

- Test the connection ability of the Lustre network
- Run regression tests of the Lustre network
- Test performance of the Lustre network

After you have obtained performance results for your Lustre network, refer to Chapter 26, *Tuning a Lustre File System* for information about parameters that can be used to tune LNET for optimum performance.

Note

Apart from the performance impact, LNET self-test is invisible to the Lustre file system.

An LNET self-test cluster includes two types of nodes:

- **Console node** - A node used to control and monitor an LNET self-test cluster. The console node serves as the user interface of the LNET self-test system and can be any node in the test cluster. All self-test commands are entered from the console node. From the console node, a user can control and monitor the status of the entire LNET self-test cluster (session). The console node is exclusive in that a user cannot control two different sessions from one console node.
- **Test nodes** - The nodes on which the tests are run. Test nodes are controlled by the user from the console node; the user does not need to log into them directly.

LNET self-test has two user utilities:

- **`lst`** - The user interface for the self-test console (run on the *console node*). It provides a list of commands to control the entire test system, including commands to create a session, create test groups, etc.
- **`lstclient`** - The userspace LNET self-test program (run on a *test node*). The `lstclient` utility is linked with userspace LNDs and LNET. This utility is not needed if only kernel space LNET and LNDs are used.

Note

Test nodes can be in either kernel or userspace. A *console node* can invite a kernel *test node* to join the session by running `lst add_group NID`, but the *console node* cannot actively add a userspace *test node* to the session. A *console node* can passively accept a *test node* to the session while the *test node* is running `lstclient` to connect to the *console node*.

24.1.1. Prerequisites

To run LNET self-test, these modules must be loaded on both *console nodes* and *test nodes*:

- `libcfs`
- `net`
- `lnet_selftest`
- `klnds`: A kernel Lustre network driver (LND) (i.e, `ksocklnd`, `ko2iblnd`...) as needed by your network configuration.

To load the required modules, run:

```
modprobe lnet_selftest
```

This command recursively loads the modules on which LNET self-test depends.

Note

While the *console node* and *test nodes* require all the prerequisite modules to be loaded, userspace test nodes do not require these modules.

24.2. Using LNET Self-Test

This section describes how to create and run an LNET self-test. The examples shown are for a test that simulates the traffic pattern of a set of Lustre servers on a TCP network accessed by Lustre clients on an InfiniBand network connected via LNET routers. In this example, half the clients are reading and half the clients are writing.

24.2.1. Creating a Session

A *session* is a set of processes that run on a *test node*. Only one session can be run at a time on a test node to ensure that the session has exclusive use of the node. The console node is used to create, change or destroy a session (`new_session`, `end_session`, `show_session`). For more about session parameters, see Section 24.3.1, “Session Commands”.

Almost all operations should be performed within the context of a session. From the *console node*, a user can only operate nodes in his own session. If a session ends, the session context in all test nodes is stopped.

The following commands set the `LST_SESSION` environment variable to identify the session on the console node and create a session called `read_write`:

```
export LST_SESSION=$$
```

```
lst new_session read_write
```

24.2.2. Setting Up Groups

A *group* is a named collection of nodes. Any number of groups can exist in a single LNET self-test session. Group membership is not restricted in that a *test node* can be included in any number of groups.

Each node in a group has a rank, determined by the order in which it was added to the group. The rank is used to establish test traffic patterns.

A user can only control nodes in his/her session. To allocate nodes to the session, the user needs to add nodes to a group (of the session). All nodes in a group can be referenced by the group name. A node can be allocated to multiple groups of a session.

In the following example, three groups are established on a console node:

```
lst add_group servers 192.168.10.[8,10,12-16]@tcp
lst add_group readers 192.168.1.[1-253/2]@o2ib
lst add_group writers 192.168.1.[2-254/2]@o2ib
```

These three groups include:

- Nodes that will function as 'servers' to be accessed by 'clients' during the LNET self-test session
- Nodes that will function as 'clients' that will simulate *reading* data from the 'servers'
- Nodes that will function as 'clients' that will simulate *writing* data to the 'servers'

Note

A *console node* can associate kernel space *test nodes* with the session by running `lst add_group` *`NIDs`*, but a userspace test node cannot be actively added to the session. A console node can passively "accept" a test node to associate with a test session while the test node running `lstclient` connects to the console node, i.e: `lstclient --sesid` *`CONSOLE_NID`* `--group` *`NAME`*).

24.2.3. Defining and Running the Tests

A *test* generates a network load between two groups of nodes, a source group identified using the `--from` parameter and a target group identified using the `--to` parameter. When a test is running, each node in the `--from` *`group`* simulates a client by sending requests to nodes in the `--to` *`group`*, which are simulating a set of servers, and then receives responses in return. This activity is designed to mimic Lustre file system RPC traffic.

A *batch* is a collection of tests that are started and stopped together and run in parallel. A test must always be run as part of a batch, even if it is just a single test. Users can only run or stop a test batch, not individual tests.

Tests in a batch are non-destructive to the file system, and can be run in a normal Lustre file system environment (provided the performance impact is acceptable).

A simple batch might contain a single test, for example, to determine whether the network bandwidth presents an I/O bottleneck. In this example, the `--to` *`group`* could be comprised of Lustre OSSs and `--from` *`group`* the compute nodes. A second test could be added to perform pings from a login node to the MDS to see how checkpointing affects the `ls -l` process.

Two types of tests are available:

- **`ping`** - A `ping` generates a short request message, which results in a short response. Pings are useful to determine latency and small message overhead and to simulate Lustre metadata traffic.

- **`brw`** - In a `brw` ('bulk read write') test, data is transferred from the target to the source (`brwread`) or data is transferred from the source to the target (`brwwrite`). The size of the bulk transfer is set using the `size` parameter. A brw test is useful to determine network bandwidth and to simulate Lustre I/O traffic.

In the example below, a batch is created called `bulk_rw`. Then two `brw` tests are added. In the first test, 1M of data is sent from the servers to the clients as a simulated read operation with a simple data validation check. In the second test, 4K of data is sent from the clients to the servers as a simulated write operation with a full data validation check.

```
lst add_batch bulk_rw
lst add_test --batch bulk_rw --from readers --to servers \
  brw read check=simple size=1M
lst add_test --batch bulk_rw --from writers --to servers \
  brw write check=full size=4K
```

The traffic pattern and test intensity is determined by several properties such as test type, distribution of test nodes, concurrency of test, and RDMA operation type. For more details, see Section 24.3.3, “Batch and Test Commands”.

24.2.4. Sample Script

This sample LNET self-test script simulates the traffic pattern of a set of Lustre servers on a TCP network, accessed by Lustre clients on an InfiniBand network (connected via LNET routers). In this example, half the clients are reading and half the clients are writing.

Run this script on the console node:

```
#!/bin/bash
export LST_SESSION=$$
lst new_session read/write
lst add_group servers 192.168.10.[8,10,12-16]@tcp
lst add_group readers 192.168.1.[1-253/2]@o2ib
lst add_group writers 192.168.1.[2-254/2]@o2ib
lst add_batch bulk_rw
lst add_test --batch bulk_rw --from readers --to servers \
brw read check=simple size=1M
lst add_test --batch bulk_rw --from writers --to servers \
brw write check=full size=4K
# start running
lst run bulk_rw
# display server stats for 30 seconds
lst stat servers & sleep 30; kill $!
# tear down
lst end_session
```

Note

This script can be easily adapted to pass the group NIDs by shell variables or command line arguments (making it good for general-purpose use).

24.3. LNET Self-Test Command Reference

The LNET self-test (`lst`) utility is used to issue LNET self-test commands. The `lst` utility takes a number of command line arguments. The first argument is the command name and subsequent arguments are command-specific.

24.3.1. Session Commands

This section describes `lst` session commands.

LST_FEATURES

The `lst` utility uses the `LST_FEATURES` environmental variable to determine what optional features should be enabled. All features are disabled by default. The supported values for `LST_FEATURES` are:

- **`1`** - Enable the Variable Page Size feature for LNet Selftest.

Example:

```
export LST_FEATURES=1
```

LST_SESSION

The `lst` utility uses the `LST_SESSION` environmental variable to identify the session locally on the self-test console node. This should be a numeric value that uniquely identifies all session processes on the node. It is convenient to set this to the process ID of the shell both for interactive use and in shell scripts. Almost all `lst` commands require `LST_SESSION` to be set.

Example:

```
export LST_SESSION=$$
```

`new_session [--timeout` *`SECONDS`*`] [--force]` *`SESSNAME`*

Creates a new session session named *`SESSNAME`*.

Parameter	Description
`--timeout` *`seconds`*	Console timeout value of the session. The session ends automatically if it remains idle (i.e., no commands are issued) for this period.
`--force`	Ends conflicting sessions. This determines who 'wins' when one session conflicts with another. For example, if there is already an active session on this node, then the attempt to create a new session fails unless the `--force` flag is specified. If the `--force` flag is specified, then the active session is ended. Similarly, if a session attempts to add a node that is already 'owned' by another session, the `--force` flag allows this session to 'steal' the node.
`name`	A human-readable string to print when listing sessions or reporting session conflicts.

Example:

```
$ lst new_session --force read_write
```

```
end_session
```

Stops all operations and tests in the current session and clears the session's status.

```
$ lst end_session
```

```
show_session
```

Shows the session information. This command prints information about the current session. It does not require LST_SESSION to be defined in the process environment.

```
$ lst show_session
```

24.3.2. Group Commands

This section describes `lst` group commands.

`add_group` *`name NIDS`* `[`*`NIDs...`*`]`

Creates the group and adds a list of test nodes to the group.

Parameter	Description
`name`	Name of the group.
`NIDs`	A string that may be expanded to include one or more LNET NIDs.

Example:

```
$ lst add_group servers 192.168.10.[35,40-45]@tcp
$ lst add_group clients 192.168.1.[10-100]@tcp 192.168.[2,4].\
  [10-20]@tcp
```

`update_group` *`name`* `[--refresh] [--clean` *`status`*`] [--remove` *`NIDs`*`]`

Updates the state of nodes in a group or adjusts a group's membership. This command is useful if some nodes have crashed and should be excluded from the group.

Parameter	Description	
`--refresh`	Refreshes the state of all inactive nodes in the group.	
`--clean` *`status`*	Removes nodes with a specified status from the group. Status may be:	
	active	The node is in the current session.
	busy	The node is now owned by another session.
	down	The node has been marked down.
	unknown	The node's status has yet to be determined.
	invalid	Any state but active.
`--remove` *`NIDs`*	Removes specified nodes from the group.	

Example:

```
$ lst update_group clients --refresh
$ lst update_group clients --clean busy
$ lst update_group clients --clean invalid // \
  invalid == busy || down || unknown
$ lst update_group clients --remove \192.168.1.[10-20]@tcp
```

`list_group [`*`name`*`] [--active] [--busy] [--down] [--unknown] [--all]`

Prints information about a group or lists all groups in the current session if no group is specified.

Parameter	Description
name	The name of the group.
`--active`	Lists the active nodes.
`--busy`	Lists the busy nodes.
`--down`	Lists the down nodes.
`--unknown`	Lists unknown nodes.
`--all`	Lists all nodes.

Example:

```
$ lst list_group
1) clients
2) servers
Total 2 groups
$ lst list_group clients
ACTIVE BUSY DOWN UNKNOWN TOTAL
3 1 2 0 6
$ lst list_group clients --all
192.168.1.10@tcp Active
192.168.1.11@tcp Active
192.168.1.12@tcp Busy
192.168.1.13@tcp Active
192.168.1.14@tcp DOWN
192.168.1.15@tcp DOWN
Total 6 nodes
$ lst list_group clients --busy
192.168.1.12@tcp Busy
Total 1 node
```

`del_group` *`name`*

Removes a group from the session. If the group is referred to by any test, then the operation fails. If nodes in the group are referred to only by this group, then they are kicked out from the current session; otherwise, they are still in the current session.

```
$ lst del_group clients
```

`lstclient --sesid` *`NID`* `--group` *`name`* `[--server_mode]`

Use `lstclient` to run the userland self-test client. The `lstclient` command should be executed after creating a session on the console. There are only two mandatory options for `lstclient`:

Parameter	Description
`--sesid` *NID*	The first console's NID.
`--group` *name*	The test group to join.
`--server_mode`	When included, forces LNET to behave as a server, such as starting an acceptor if the underlying NID needs it or using privileged ports. Only root is allowed to use the `--server_mode` option.

Example:

```
Console $ lst new_session testsession
Client1 $ lstclient --sesid 192.168.1.52@tcp --group clients
```

Example:

```
Client1 $ lstclient --sesid 192.168.1.52@tcp |--group clients --server_mode
```

24.3.3. Batch and Test Commands

This section describes `lst` batch and test commands.

`add_batch` *name*

A default batch test set named batch is created when the session is started. You can specify a batch name by using `add_batch`:

```
$ lst add_batch bulkperf
```

Creates a batch test called `bulkperf`.

```
add_test --batch batchname [--loop loop_count] [--concurrency active_count] [--dis
         --from group --to group brw|ping test_options
```

Adds a test to a batch. The parameters are described below.

Parameter	Description
`--batch` *batchname*	Names a group of tests for later execution.
`--loop` *loop_count*	Number of times to run the test.
`--concurrency` *active_count*	The number of requests that are active at one time.
`--distribute` *source_count*`:`*sink_count*	Determines the ratio of client nodes to server nodes for the specified test. This allows you to specify a wide range of topologies, including one-to-one and all-to-all. Distribution divides the source group into subsets, which are paired with equivalent subsets from the target group so only nodes in matching subsets communicate.
`--from` *group*	The source group (test client).
`--to` *group*	The target group (test server).
`ping`	Sends a small request message, resulting in a small reply message. For more details, see Section 24.2.3, “Defining and Running the Tests”. `ping` does not have any additional options.

Parameter	Description	
brw	Sends a small request message followed by a bulk data transfer, resulting in a small reply message. Section 24.2.3, “Defining and Running the Tests”. Options are:	
	read | write	Read or write. The default is read.
	size=*bytes[KM]*	I/O size in bytes, kilobytes, or Megabytes (i.e., size=1024, size=4K, size=1M). The default is 4 kilobytes.
	check=full|simple	A data validation check (checksum of data). The default is that no check is done.

Examples showing use of the distribute parameter:

```
Clients: (C1, C2, C3, C4, C5, C6)
Server: (S1, S2, S3)
--distribute 1:1 (C1->S1), (C2->S2), (C3->S3), (C4->S1), (C5->S2),
\(C6->S3) /* -> means test conversation */ --distribute 2:1 (C1,C2->S1), (C3,C4->S
--distribute 3:1 (C1,C2,C3->S1), (C4,C5,C6->S2), (NULL->S3)
--distribute 3:2 (C1,C2,C3->S1,S2), (C4,C5,C6->S3,S1)
--distribute 4:1 (C1,C2,C3,C4->S1), (C5,C6->S2), (NULL->S3)
--distribute 4:2 (C1,C2,C3,C4->S1,S2), (C5, C6->S3, S1)
--distribute 6:3 (C1,C2,C3,C4,C5,C6->S1,S2,S3)
```

The setting --distribute 1:1 is the default setting where each source node communicates with one target node.

When the setting --distribute 1: *n* (where *n* is the size of the target group) is used, each source node communicates with every node in the target group.

Note that if there are more source nodes than target nodes, some source nodes may share the same target nodes. Also, if there are more target nodes than source nodes, some higher-ranked target nodes will be idle.

Example showing a brw test:

```
$ lst add_group clients 192.168.1.[10-17]@tcp
$ lst add_group servers 192.168.10.[100-103]@tcp
$ lst add_batch bulkperf
$ lst add_test --batch bulkperf --loop 100 --concurrency 4 \
  --distribute 4:2 --from clients brw WRITE size=16K
```

In the example above, a batch test called bulkperf that will do a 16 kbyte bulk write request. In this test, two groups of four clients (sources) write to each of four servers (targets) as shown below:

- 192.168.1.[10-13] will write to 192.168.10.[100,101]
- 192.168.1.[14-17] will write to 192.168.10.[102,103]

list_batch [*name*] [--test *index*] [--active] [--invalid] [--server|client]

Lists batches in the current session or lists client and server nodes in a batch or a test.

Parameter	Description	
`--test` *index*	Lists tests in a batch. If no option is used, all tests in the batch are listed. If one of these options are used, only specified tests in the batch are listed:	
	`active`	Lists only active batch tests.
	`invalid`	Lists only invalid batch tests.
	`server | client`	Lists client and server nodes in a batch test.

Example:

```
$ lst list_batchbulkperf
$ lst list_batch bulkperf
Batch: bulkperf Tests: 1 State: Idle
ACTIVE BUSY DOWN UNKNOWN TOTAL
client 8 0 0 0 8
server 4 0 0 0 4
Test 1(brw) (loop: 100, concurrency: 4)
ACTIVE BUSY DOWN UNKNOWN TOTAL
client 8 0 0 0 8
server 4 0 0 0 4
$ lst list_batch bulkperf --server --active
192.168.10.100@tcp Active
192.168.10.101@tcp Active
192.168.10.102@tcp Active
192.168.10.103@tcp Active
```

`run` *name*

Runs the batch.

```
$ lst run bulkperf
```

`stop` *name*

Stops the batch.

```
$ lst stop bulkperf
```

`query` *`name`* `[--test` *`index`*`] [--timeout` *`seconds`*`] [--loop` *`loopcount`*`] [--delay` *`seconds`*`] [--all]`

Queries the batch status.

Parameter	Description
`--test` *index*	Only queries the specified test. The test index starts from 1.
`--timeout` *seconds*	The timeout value to wait for RPC. The default is 5 seconds.
`--loop` *#*	The loop count of the query.
`--delay` *seconds*	The interval of each query. The default is 5 seconds.

Parameter	Description
`--all`	The list status of all nodes in a batch or a test.

Example:

```
$ lst run bulkperf
$ lst query bulkperf --loop 5 --delay 3
Batch is running
Batch is running
Batch is running
Batch is running
Batch is running
$ lst query bulkperf --all
192.168.1.10@tcp Running
192.168.1.11@tcp Running
192.168.1.12@tcp Running
192.168.1.13@tcp Running
192.168.1.14@tcp Running
192.168.1.15@tcp Running
192.168.1.16@tcp Running
192.168.1.17@tcp Running
$ lst stop bulkperf
$ lst query bulkperf
Batch is idle
```

24.3.4. Other Commands

This section describes other `lst` commands.

`ping [-session] [--group` *`name`*`] [--nodes` *`NIDs`*`] [--batch` *`name`*`] [--server] [--timeout` *`seconds`*`]`

Sends a 'hello' query to the nodes.

Parameter	Description
`--session`	Pings all nodes in the current session.
`--group` *`name`*	Pings all nodes in a specified group.
`--nodes` *`NIDs`*	Pings all specified nodes.
`--batch` *`name`*	Pings all client nodes in a batch.
`--server`	Sends RPC to all server nodes instead of client nodes. This option is only used with `--batch` *`name`*.
`--timeout` *`seconds`*	The RPC timeout value.

Example:

```
# lst ping 192.168.10.[15-20]@tcp
192.168.1.15@tcp Active [session: liang id: 192.168.1.3@tcp]
192.168.1.16@tcp Active [session: liang id: 192.168.1.3@tcp]
192.168.1.17@tcp Active [session: liang id: 192.168.1.3@tcp]
192.168.1.18@tcp Busy [session: Isaac id: 192.168.10.10@tcp]
```

```
192.168.1.19@tcp Down [session: <NULL> id: LNET_NID_ANY]
192.168.1.20@tcp Down [session: <NULL> id: LNET_NID_ANY]
```

```
stat [--bw] [--rate] [--read] [--write] [--max] [--min] [--avg] " " [--
timeout seconds] [--delay seconds] group|NIDs [group|NIDs]
```

The collection performance and RPC statistics of one or more nodes.

Parameter	Description
`--bw`	Displays the bandwidth of the specified group/ nodes.
`--rate`	Displays the rate of RPCs of the specified group/ nodes.
`--read`	Displays the read statistics of the specified group/ nodes.
`--write`	Displays the write statistics of the specified group/ nodes.
`--max`	Displays the maximum value of the statistics.
`--min`	Displays the minimum value of the statistics.
`--avg`	Displays the average of the statistics.
`--timeout` *seconds*	The timeout of the statistics RPC. The default is 5 seconds.
`--delay` *seconds*	The interval of the statistics (in seconds).

Example:

```
$ lst run bulkperf
$ lst stat clients
[LNet Rates of clients]
[W] Avg: 1108 RPC/s Min: 1060 RPC/s Max: 1155 RPC/s
[R] Avg: 2215 RPC/s Min: 2121 RPC/s Max: 2310 RPC/s
[LNet Bandwidth of clients]
[W] Avg: 16.60 MB/s Min: 16.10 MB/s Max: 17.1 MB/s
[R] Avg: 40.49 MB/s Min: 40.30 MB/s Max: 40.68 MB/s
```

Specifying a group name (*group*) causes statistics to be gathered for all nodes in a test group. For example:

```
$ lst stat servers
```

where servers is the name of a test group created by `lst add_group`

Specifying a *NID* range (*NIDs*) causes statistics to be gathered for selected nodes. For example:

```
$ lst stat 192.168.0.[1-100/2]@tcp
```

Only LNET performance statistics are available. By default, all statistics information is displayed. Users can specify additional information with these options.

```
show_error [--session] [group|NIDs]...
```

Lists the number of failed RPCs on test nodes.

Parameter	Description
`--session`	Lists errors in the current test session. With this option, historical RPC errors are not listed.

Example:

```
$ lst show_error client
sclients
12345-192.168.1.15@tcp: [Session: 1 brw errors, 0 ping errors] \
  [RPC: 20 errors, 0 dropped,
12345-192.168.1.16@tcp: [Session: 0 brw errors, 0 ping errors] \
  [RPC: 1 errors, 0 dropped, Total 2 error nodes in clients
$ lst show_error --session clients
clients
12345-192.168.1.15@tcp: [Session: 1 brw errors, 0 ping errors]
Total 1 error nodes in clients
```

Chapter 25. Benchmarking Lustre File System Performance (Lustre I/O Kit)

This chapter describes the Lustre I/O kit, a collection of I/O benchmarking tools for a Lustre cluster, and PIOS, a parallel I/O simulator for Linux and Solaris* operating systems. It includes:

- Section 25.1, " Using Lustre I/O Kit Tools"
- Section 25.2, "Testing I/O Performance of Raw Hardware (`sgpdd-survey`)"
- Section 25.3, "Testing OST Performance (`obdfilter-survey`)"
- Section 25.4, "Testing OST I/O Performance (`ost-survey`)"
- Section 25.5, "Testing MDS Performance (`mds-survey`)"
- Section 25.6, "Collecting Application Profiling Information (`stats-collect`)"

25.1. Using Lustre I/O Kit Tools

The tools in the Lustre I/O Kit are used to benchmark Lustre file system hardware and validate that it is working as expected before you install the Lustre software. It can also be used to to validate the performance of the various hardware and software layers in the cluster and also to find and troubleshoot I/O issues.

Typically, performance is measured starting with single raw devices and then proceeding to groups of devices. Once raw performance has been established, other software layers are then added incrementally and tested.

25.1.1. Contents of the Lustre I/O Kit

The I/O kit contains three tests, each of which tests a progressively higher layer in the Lustre software stack:

- `sgpdd-survey` - Measure basic 'bare metal' performance of devices while bypassing the kernel block device layers, buffer cache, and file system.
- `obdfilter-survey` - Measure the performance of one or more OSTs directly on the OSS node or alternately over the network from a Lustre client.
- `ost-survey` - Performs I/O against OSTs individually to allow performance comparisons to detect if an OST is performing suboptimally due to hardware issues.

Typically with these tests, a Lustre file system should deliver 85-90% of the raw device performance.

A utility `stats-collect` is also provided to collect application profiling information from Lustre clients and servers. See Section 25.6, "Collecting Application Profiling Information (`stats-collect`)" for more information.

25.1.2. Preparing to Use the Lustre I/O Kit

The following prerequisites must be met to use the tests in the Lustre I/O kit:

- Password-free remote access to nodes in the system (provided by `ssh` or `rsh`).

- LNET self-test completed to test that Lustre networking has been properly installed and configured. See Chapter 24, *Testing Lustre Network Performance (LNET Self-Test)*.
- Lustre file system software installed.
- `sg3_utils` package providing the `sgp_dd` tool (`sg3_utils` is a separate RPM package available online using YUM).

Download the Lustre I/O kit (`lustre-iokit`)from:

http://downloads.hpdd.intel.com/

25.2. Testing I/O Performance of Raw Hardware (`sgpdd-survey`)

The `sgpdd-survey` tool is used to test bare metal I/O performance of the raw hardware, while bypassing as much of the kernel as possible. This survey may be used to characterize the performance of a SCSI device by simulating an OST serving multiple stripe files. The data gathered by this survey can help set expectations for the performance of a Lustre OST using this device.

The script uses `sgp_dd` to carry out raw sequential disk I/O. It runs with variable numbers of `sgp_dd` threads to show how performance varies with different request queue depths.

The script spawns variable numbers of `sgp_dd` instances, each reading or writing a separate area of the disk to demonstrate performance variance within a number of concurrent stripe files.

Several tips and insights for disk performance measurement are described below. Some of this information is specific to RAID arrays and/or the Linux RAID implementation.

- *Performance is limited by the slowest disk.*

  Before creating a RAID array, benchmark all disks individually. We have frequently encountered situations where drive performance was not consistent for all devices in the array. Replace any disks that are significantly slower than the rest.

- *Disks and arrays are very sensitive to request size.*

  To identify the optimal request size for a given disk, benchmark the disk with different record sizes ranging from 4 KB to 1 to 2 MB.

Caution

The `sgpdd-survey` script overwrites the device being tested, which results in the ***LOSS OF ALL DATA*** on that device. Exercise caution when selecting the device to be tested.

Note

Array performance with all LUNs loaded does not always match the performance of a single LUN when tested in isolation.

Prerequisites:

- `sgp_dd` tool in the `sg3_utils` package
- Lustre software is *NOT* required

The device(s) being tested must meet one of these two requirements:

- If the device is a SCSI device, it must appear in the output of `sg_map` (make sure the kernel module `sg` is loaded).
- If the device is a raw device, it must appear in the output of `raw -qa`.

Raw and SCSI devices cannot be mixed in the test specification.

Note

If you need to create raw devices to use the `sgpdd-survey` tool, note that raw device 0 cannot be used due to a bug in certain versions of the "raw" utility (including the version shipped with Red Hat Enterprise Linux 4U4.)

25.2.1. Tuning Linux Storage Devices

To get large I/O transfers (1 MB) to disk, it may be necessary to tune several kernel parameters as specified:

```
/sys/block/sdN/queue/max_sectors_kb = 4096
/sys/block/sdN/queue/max_phys_segments = 256
/proc/scsi/sg/allow_dio = 1
/sys/module/ib_srp/parameters/srp_sg_tablesize = 255
/sys/block/sdN/queue/scheduler
```

Note

Recommended schedulers are **deadline** and **noop**. The scheduler is set by default to **deadline**, unless it has already been set to **noop**.

25.2.2. Running sgpdd-survey

The `sgpdd-survey` script must be customized for the particular device being tested and for the location where the script saves its working and result files (by specifying the `${rslt}` variable). Customization variables are described at the beginning of the script.

When the `sgpdd-survey` script runs, it creates a number of working files and a pair of result files. The names of all the files created start with the prefix defined in the variable `${rslt}`. (The default value is `/tmp`.) The files include:

- File containing standard output data (same as `stdout`)

  `rslt_date_time``.summary`

- Temporary (tmp) files

  `rslt_date_time_*`

- Collected tmp files for post-mortem

  `rslt_date_time``.detail`

The `stdout` and the `.summary` file will contain lines like this:

```
total_size 8388608K rsz 1024 thr 1 crg 1 180.45 MB/s 1 x 180.50 \
        = 180.50 MB/s
```

Each line corresponds to a run of the test. Each test run will have a different number of threads, record size, or number of regions.

- `total_size` - Size of file being tested in KBs (8 GB in above example).
- `rsz` - Record size in KBs (1 MB in above example).
- `thr` - Number of threads generating I/O (1 thread in above example).
- `crg` - Current regions, the number of disjoint areas on the disk to which I/O is being sent (1 region in above example, indicating that no seeking is done).
- `MB/s` - Aggregate bandwidth measured by dividing the total amount of data by the elapsed time (180.45 MB/s in the above example).
- `MB/s` - The remaining numbers show the number of regions X performance of the slowest disk as a sanity check on the aggregate bandwidth.

If there are so many threads that the `sgp_dd` script is unlikely to be able to allocate I/O buffers, then `ENOMEM` is printed in place of the aggregate bandwidth result.

If one or more `sgp_dd` instances do not successfully report a bandwidth number, then `FAILED` is printed in place of the aggregate bandwidth result.

25.3. Testing OST Performance (`obdfilter-survey`)

The `obdfilter-survey` script generates sequential I/O from varying numbers of threads and objects (files) to simulate the I/O patterns of a Lustre client.

The `obdfilter-survey` script can be run directly on the OSS node to measure the OST storage performance without any intervening network, or it can be run remotely on a Lustre client to measure the OST performance including network overhead.

The `obdfilter-survey` is used to characterize the performance of the following:

- **Local file system** - In this mode, the `obdfilter-survey` script exercises one or more instances of the obdfilter directly. The script may run on one or more OSS nodes, for example, when the OSSs are all attached to the same multi-ported disk subsystem.

  Run the script using the `case=disk` parameter to run the test against all the local OSTs. The script automatically detects all local OSTs and includes them in the survey.

  To run the test against only specific OSTs, run the script using the `target=parameter` to list the OSTs to be tested explicitly. If some OSTs are on remote nodes, specify their hostnames in addition to the OST name (for example, `oss2:lustre-OST0004`).

  All `obdfilter` instances are driven directly. The script automatically loads the `obdecho` module (if required) and creates one instance of `echo_client` for each `obdfilter` instance in order to generate I/O requests directly to the OST.

  For more details, see Section 25.3.1, “Testing Local Disk Performance”.

- **Network** - In this mode, the Lustre client generates I/O requests over the network but these requests are not sent to the OST file system. The OSS node runs the obdecho server to receive the requests but discards them before they are sent to the disk.

Pass the parameters `case=network` and `target=`*`hostname|IP_of_server`* to the script. For each network case, the script does the required setup.

For more details, see Section 25.3.2, “Testing Network Performance”

- **Remote file system over the network** - In this mode the `obdfilter-survey` script generates I/O from a Lustre client to a remote OSS to write the data to the file system.

  To run the test against all the local OSCs, pass the parameter `case=netdisk` to the script. Alternately you can pass the target= parameter with one or more OSC devices (e.g., `lustre-OST0000-osc-ffff88007754bc00`) against which the tests are to be run.

  For more details, see Section 25.3.3, “Testing Remote Disk Performance”.

Caution

The `obdfilter-survey` script is potentially destructive and there is a small risk data may be lost. To reduce this risk, `obdfilter-survey` should not be run on devices that contain data that needs to be preserved. Thus, the best time to run `obdfilter-survey` is before the Lustre file system is put into production. The reason `obdfilter-survey` may be safe to run on a production file system is because it creates objects with object sequence 2. Normal file system objects are typically created with object sequence 0.

Note

If the `obdfilter-survey` test is terminated before it completes, some small amount of space is leaked. you can either ignore it or reformat the file system.

Note

The `obdfilter-survey` script is *NOT* scalable beyond tens of OSTs since it is only intended to measure the I/O performance of individual storage subsystems, not the scalability of the entire system.

Note

The `obdfilter-survey` script must be customized, depending on the components under test and where the script's working files should be kept. Customization variables are described at the beginning of the `obdfilter-survey` script. In particular, pay attention to the listed maximum values listed for each parameter in the script.

25.3.1. Testing Local Disk Performance

The `obdfilter-survey` script can be run automatically or manually against a local disk. This script profiles the overall throughput of storage hardware, including the file system and RAID layers managing the storage, by sending workloads to the OSTs that vary in thread count, object count, and I/O size.

When the `obdfilter-survey` script is run, it provides information about the performance abilities of the storage hardware and shows the saturation points.

The `plot-obdfilter` script generates from the output of the `obdfilter-survey` a CSV file and parameters for importing into a spreadsheet or gnuplot to visualize the data.

To run the `obdfilter-survey` script, create a standard Lustre file system configuration; no special setup is needed.

To perform an automatic run:

1. Start the Lustre OSTs.

   The Lustre OSTs should be mounted on the OSS node(s) to be tested. The Lustre client is not required to be mounted at this time.

2. Verify that the obdecho module is loaded. Run:

   ```
   modprobe obdecho
   ```

3. Run the `obdfilter-survey` script with the parameter `case=disk`.

   For example, to run a local test with up to two objects (nobjhi), up to two threads (thrhi), and 1024 MB transfer size (size):

   ```
   $ nobjhi=2 thrhi=2 size=1024 case=disk sh obdfilter-survey
   ```

To perform a manual run:

1. Start the Lustre OSTs.

   The Lustre OSTs should be mounted on the OSS node(s) to be tested. The Lustre client is not required to be mounted at this time.

2. Verify that the `obdecho` module is loaded. Run:

   ```
   modprobe obdecho
   ```

3. Determine the OST names.

   On the OSS nodes to be tested, run the `lctl dl` command. The OST device names are listed in the fourth column of the output. For example:

   ```
   $ lctl dl |grep obdfilter
   0 UP obdfilter lustre-OST0001 lustre-OST0001_UUID 1159
   2 UP obdfilter lustre-OST0002 lustre-OST0002_UUID 1159
   ...
   ```

4. List all OSTs you want to test.

   Use the `target=parameter` to list the OSTs separated by spaces. List the individual OSTs by name using the format *`fsname-OSTnumber`* (for example, `lustre-OST0001`). You do not have to specify an MDS or LOV.

5. Run the `obdfilter-survey` script with the `target=parameter`.

   For example, to run a local test with up to two objects (`nobjhi`), up to two threads (`thrhi`), and 1024 Mb (size) transfer size:

   ```
   $ nobjhi=2 thrhi=2 size=1024 targets="lustre-OST0001 \
       lustre-OST0002" sh obdfilter-survey
   ```

25.3.2. Testing Network Performance

The `obdfilter-survey` script can only be run automatically against a network; no manual test is provided.

To run the network test, a specific Lustre file system setup is needed. Make sure that these configuration requirements have been met.

To perform an automatic run:

1. Start the Lustre OSTs.

   The Lustre OSTs should be mounted on the OSS node(s) to be tested. The Lustre client is not required to be mounted at this time.

2. Verify that the `obdecho` module is loaded. Run:

   ```
   modprobe obdecho
   ```

3. Start `lctl` and check the device list, which must be empty. Run:

   ```
   lctl dl
   ```

4. Run the `obdfilter-survey` script with the parameters `case=network` and `targets=`*`hostname|ip_of_server`*. For example:

   ```
   $ nobjhi=2 thrhi=2 size=1024 targets="oss0 oss1" \
       case=network sh odbfilter-survey
   ```

5. On the server side, view the statistics at:

   `/proc/fs/lustre/obdecho/`*`echo_srv`*`/stats`

   where *`echo_srv`* is the `obdecho` server created by the script.

25.3.3. Testing Remote Disk Performance

The `obdfilter-survey` script can be run automatically or manually against a network disk. To run the network disk test, start with a standard Lustre configuration. No special setup is needed.

To perform an automatic run:

1. Start the Lustre OSTs.

   The Lustre OSTs should be mounted on the OSS node(s) to be tested. The Lustre client is not required to be mounted at this time.

2. Verify that the `obdecho` module is loaded. Run:

   ```
   modprobe obdecho
   ```

3. Run the `obdfilter-survey` script with the parameter `case=netdisk`. For example:

   ```
   $ nobjhi=2 thrhi=2 size=1024 case=netdisk sh obdfilter-survey
   ```

To perform a manual run:

1. Start the Lustre OSTs.

   The Lustre OSTs should be mounted on the OSS node(s) to be tested. The Lustre client is not required to be mounted at this time.

2. Verify that the `obdecho` module is loaded. Run:

modprobe obdecho

3. Determine the OSC names.

   On the OSS nodes to be tested, run the `lctl dl` command. The OSC device names are listed in the fourth column of the output. For example:

   ```
   $ lctl dl |grep obdfilter
   3 UP osc lustre-OST0000-osc-ffff88007754bc00 \
             54b91eab-0ea9-1516-b571-5e6df349592e 5
   4 UP osc lustre-OST0001-osc-ffff88007754bc00 \
             54b91eab-0ea9-1516-b571-5e6df349592e 5
   ...
   ```

4. List all OSCs you want to test.

   Use the `target=parameter` to list the OSCs separated by spaces. List the individual OSCs by name separated by spaces using the format *`fsname-OST_name`*`-osc-`*`instance`* (for example, `lustre-OST0000-osc-ffff88007754bc00`). You *do not have to specify an MDS or LOV.*

5. Run the `obdfilter-survey` script with the `target=`*`osc`* and `case=netdisk`.

   An example of a local test run with up to two objects (`nobjhi`), up to two threads (`thrhi`), and 1024 Mb (size) transfer size is shown below:

   ```
   $ nobjhi=2 thrhi=2 size=1024 \
             targets="lustre-OST0000-osc-ffff88007754bc00 \
             lustre-OST0001-osc-ffff88007754bc00" sh obdfilter-survey
   ```

25.3.4. Output Files

When the `obdfilter-survey` script runs, it creates a number of working files and a pair of result files. All files start with the prefix defined in the variable `${rslt}`.

File	Description
`${rslt}.summary`	Same as stdout
`${rslt}.script_*`	Per-host test script files
`${rslt}.detail_tmp*`	Per-OST result files
`${rslt}.detail`	Collected result files for post-mortem

The `obdfilter-survey` script iterates over the given number of threads and objects performing the specified tests and checks that all test processes have completed successfully.

Note

The `obdfilter-survey` script may not clean up properly if it is aborted or if it encounters an unrecoverable error. In this case, a manual cleanup may be required, possibly including killing any running instances of `lctl` (local or remote), removing `echo_client` instances created by the script and unloading `obdecho`.

25.3.4.1. Script Output

The `.summary` file and `stdout` of the `obdfilter-survey` script contain lines like:

```
ost 8 sz 67108864K rsz 1024 obj 8 thr 8 write 613.54 [ 64.00, 82.00]
```

Where:

Parameter and value	Description
ost 8	Total number of OSTs being tested.
sz 67108864K	Total amount of data read or written (in KB).
rsz 1024	Record size (size of each echo_client I/O, in KB).
obj 8	Total number of objects over all OSTs.
thr 8	Total number of threads over all OSTs and objects.
write	Test name. If more tests have been specified, they all appear on the same line.
613.54	Aggregate bandwidth over all OSTs (measured by dividing the total number of MB by the elapsed time).
[64, 82.00]	Minimum and maximum instantaneous bandwidths on an individual OST.

Note

Although the numbers of threads and objects are specified per-OST in the customization section of the script, the reported results are aggregated over all OSTs.

25.3.4.2. Visualizing Results

It is useful to import the `obdfilter-survey` script summary data (it is fixed width) into Excel (or any graphing package) and graph the bandwidth versus the number of threads for varying numbers of concurrent regions. This shows how the OSS performs for a given number of concurrently-accessed objects (files) with varying numbers of I/Os in flight.

It is also useful to monitor and record average disk I/O sizes during each test using the 'disk io size' histogram in the file `/proc/fs/lustre/obdfilter/` (see Section 32.3.5, “Monitoring the OST Block I/O Stream” for details). These numbers help identify problems in the system when full-sized I/Os are not submitted to the underlying disk. This may be caused by problems in the device driver or Linux block layer.

```
*/brw_stats
```

The `plot-obdfilter` script included in the I/O toolkit is an example of processing output files to a .csv format and plotting a graph using `gnuplot`.

25.4. Testing OST I/O Performance (`ost-survey`)

The `ost-survey` tool is a shell script that uses `lfs setstripe` to perform I/O against a single OST. The script writes a file (currently using `dd`) to each OST in the Lustre file system, and compares read and write speeds. The `ost-survey` tool is used to detect anomalies between otherwise identical disk subsystems.

Note

We have frequently discovered wide performance variations across all LUNs in a cluster. This may be caused by faulty disks, RAID parity reconstruction during the test, or faulty network hardware.

To run the `ost-survey` script, supply a file size (in KB) and the Lustre file system mount point. For example, run:

```
$ ./ost-survey.sh 10 /mnt/lustre
```

Typical output is:

```
Average read Speed:                 6.73
Average write Speed:                5.41
read - Worst OST indx 0             5.84 MB/s
write - Worst OST indx 0            3.77 MB/s
read - Best OST indx 1              7.38 MB/s
write - Best OST indx 1             6.31 MB/s
3 OST devices found
Ost index 0 Read speed              5.84        Write speed     3.77
Ost index 0 Read time               0.17        Write time      0.27
Ost index 1 Read speed              7.38        Write speed     6.31
Ost index 1 Read time               0.14        Write time      0.16
Ost index 2 Read speed              6.98        Write speed     6.16
Ost index 2 Read time               0.14        Write time      0.16
```

25.5. Testing MDS Performance (`mds-survey`)

`mds-survey` is available in Lustre software release 2.2 and beyond. The `mds-survey` script tests the local metadata performance using the echo_client to drive different layers of the MDS stack: mdd, mdt, osd (the Lustre software only supports mdd stack). It can be used with the following classes of operations:

- `Open-create/mkdir/create`
- `Lookup/getattr/setxattr`
- `Delete/destroy`
- `Unlink/rmdir`

These operations will be run by a variable number of concurrent threads and will test with the number of directories specified by the user. The run can be executed such that all threads operate in a single directory (dir_count=1) or in private/unique directory (dir_count=x thrlo=x thrhi=x).

The mdd instance is driven directly. The script automatically loads the obdecho module if required and creates instance of echo_client.

This script can also create OST objects by providing stripe_count greater than zero.

To perform a run:

1. Start the Lustre MDT.

The Lustre MDT should be mounted on the MDS node to be tested.

2. Start the Lustre OSTs (optional, only required when test with OST objects)

   The Lustre OSTs should be mounted on the OSS node(s).

3. Run the `mds-survey` script as explain below

   The script must be customized according to the components under test and where it should keep its working files. Customization variables are described as followed:

   - `thrlo` - threads to start testing. skipped if less than `dir_count`
   - `thrhi` - maximum number of threads to test
   - `targets` - MDT instance
   - `file_count` - number of files per thread to test
   - `dir_count` - total number of directories to test. Must be less than or equal to `thrhi`
   - `stripe_count` - number stripe on OST objects
   - `tests_str` - test operations. Must have at least "create" and "destroy"
   - `start_number` - base number for each thread to prevent name collisions
   - `layer` - MDS stack's layer to be tested

   Run without OST objects creation:

   Setup the Lustre MDS without OST mounted. Then invoke the `mds-survey` script

   ```
   $ thrhi=64 file_count=200000 sh mds-survey
   ```

   Run with OST objects creation:

   Setup the Lustre MDS with at least one OST mounted. Then invoke the `mds-survey` script with `stripe_count` parameter

   ```
   $ thrhi=64 file_count=200000 stripe_count=2 sh mds-survey
   ```

   Note: a specific MDT instance can be specified using targets variable.

   ```
   $ targets=lustre-MDT0000 thrhi=64 file_count=200000 stripe_count=2 sh mds-survey
   ```

25.5.1. Output Files

When the `mds-survey` script runs, it creates a number of working files and a pair of result files. All files start with the prefix defined in the variable `${rslt}`.

File	Description
`${rslt}.summary`	Same as stdout
`${rslt}.script_*`	Per-host test script files
`${rslt}.detail_tmp*`	Per-mdt result files

File	Description
`${rslt}.detail`	Collected result files for post-mortem

The `mds-survey` script iterates over the given number of threads performing the specified tests and checks that all test processes have completed successfully.

Note

The `mds-survey` script may not clean up properly if it is aborted or if it encounters an unrecoverable error. In this case, a manual cleanup may be required, possibly including killing any running instances of `lctl`, removing `echo_client` instances created by the script and unloading `obdecho`.

25.5.2. Script Output

The `.summary` file and `stdout` of the `mds-survey` script contain lines like:

```
mdt 1 file 100000 dir 4 thr 4 create 5652.05 [ 999.01,46940.48] destroy 5797.79 [
```

Where:

Parameter and value	Description
mdt 1	Total number of MDT under test
file 100000	Total number of files per thread to operate
dir 4	Total number of directories to operate
thr 4	Total number of threads operate over all directories
create, destroy	Tests name. More tests will be displayed on the same line.
565.05	Aggregate operations over MDT measured by dividing the total number of operations by the elapsed time.
[999.01,46940.48]	Minimum and maximum instantaneous operation seen on any individual MDT

Note

If script output has "ERROR", this usually means there is issue during the run such as running out of space on the MDT and/or OST. More detailed debug information is available in the ${rslt}.detail file

25.6. Collecting Application Profiling Information (`stats-collect`)

The `stats-collect` utility contains the following scripts used to collect application profiling information from Lustre clients and servers:

- `lstat.sh` - Script for a single node that is run on each profile node.
- `gather_stats_everywhere.sh` - Script that collect statistics.

- `config.sh` - Script that contains customized configuration descriptions.

The `stats-collect` utility requires:

- Lustre software to be installed and set up on your cluster
- SSH and SCP access to these nodes without requiring a password

25.6.1. Using `stats-collect`

The stats-collect utility is configured by including profiling configuration variables in the config.sh script. Each configuration variable takes the following form, where 0 indicates statistics are to be collected only when the script starts and stops and *n* indicates the interval in seconds at which statistics are to be collected:

*statistic*_INTERVAL=*0*|*n*

Statistics that can be collected include:

- `VMSTAT` - Memory and CPU usage and aggregate read/write operations
- `SERVICE` - Lustre OST and MDT RPC service statistics
- `BRW` - OST bulk read/write statistics (brw_stats)
- `SDIO` - SCSI disk IO statistics (sd_iostats)
- `MBALLOC` - `ldiskfs` block allocation statistics
- `IO` - Lustre target operations statistics
- `JBD` - ldiskfs journal statistics
- `CLIENT` - Lustre OSC request statistics

To collect profile information:

Begin collecting statistics on each node specified in the config.sh script.

1. Starting the collect profile daemon on each node by entering:

   ```
   sh gather_stats_everywhere.sh config.sh start
   ```

2. Run the test.

3. Stop collecting statistics on each node, clean up the temporary file, and create a profiling tarball.

   Enter:

   ```
   sh gather_stats_everywhere.sh config.sh stop log_name.tgz
   ```

   When *log_name*`.tgz` is specified, a profile tarball */tmp/log_name*`.tgz` is created.

4. Analyze the collected statistics and create a csv tarball for the specified profiling data.

   ```
   sh gather_stats_everywhere.sh config.sh analyse log_tarball.tgz csv
   ```

Chapter 26. Tuning a Lustre File System

This chapter contains information about tuning a Lustre file system for better performance and includes the following sections:

- Section 26.1, " Optimizing the Number of Service Threads"
- Section 26.1.2, "Specifying the MDS Service Thread Count"
- Section 26.3, " Tuning LNET Parameters"
- Section 26.4, "libcfs Tuning"
- Section 26.5, "LND Tuning"
- Section 26.6, "Network Request Scheduler (NRS) Tuning"
- Section 26.7, "Lockless I/O Tunables"
- Section 26.8, "Improving Lustre File System Performance When Working with Small Files"
- Section 26.9, "Understanding Why Write Performance is Better Than Read Performance"

Note

Many options in the Lustre software are set by means of kernel module parameters. These parameters are contained in the `/etc/modprobe.d/lustre.conf` file.

26.1. Optimizing the Number of Service Threads

An OSS can have a minimum of two service threads and a maximum of 512 service threads. The number of service threads is a function of how much RAM and how many CPUs are on each OSS node (1 thread / 128MB * num_cpus). If the load on the OSS node is high, new service threads will be started in order to process more requests concurrently, up to 4x the initial number of threads (subject to the maximum of 512). For a 2GB 2-CPU system, the default thread count is 32 and the maximum thread count is 128.

Increasing the size of the thread pool may help when:

- Several OSTs are exported from a single OSS
- Back-end storage is running synchronously
- I/O completions take excessive time due to slow storage

Decreasing the size of the thread pool may help if:

- Clients are overwhelming the storage capacity
- There are lots of "slow I/O" or similar messages

Increasing the number of I/O threads allows the kernel and storage to aggregate many writes together for more efficient disk I/O. The OSS thread pool is shared--each thread allocates approximately 1.5 MB (maximum RPC size + 0.5 MB) for internal I/O buffers.

It is very important to consider memory consumption when increasing the thread pool size. Drives are only able to sustain a certain amount of parallel I/O activity before performance is degraded, due to the high number of seeks and the OST threads just waiting for I/O. In this situation, it may be advisable to decrease the load by decreasing the number of OST threads.

Determining the optimum number of OST threads is a process of trial and error, and varies for each particular configuration. Variables include the number of OSTs on each OSS, number and speed of disks, RAID configuration, and available RAM. You may want to start with a number of OST threads equal to the number of actual disk spindles on the node. If you use RAID, subtract any dead spindles not used for actual data (e.g., 1 of N of spindles for RAID5, 2 of N spindles for RAID6), and monitor the performance of clients during usual workloads. If performance is degraded, increase the thread count and see how that works until performance is degraded again or you reach satisfactory performance.

Note

If there are too many threads, the latency for individual I/O requests can become very high and should be avoided. Set the desired maximum thread count permanently using the method described above.

26.1.1. Specifying the OSS Service Thread Count

The `oss_num_threads` parameter enables the number of OST service threads to be specified at module load time on the OSS nodes:

```
options ost oss_num_threads={N}
```

After startup, the minimum and maximum number of OSS thread counts can be set via the `{service}.thread_{min,max,started}` tunable. To change the tunable at runtime, run:

```
lctl {get,set}_param {service}.thread_{min,max,started}
```

Lustre software release 2.3 introduced binding service threads to CPU partition. This works in a similar fashion to binding of threads on MDS. MDS thread tuning is covered in Section 26.2, “Binding MDS Service Thread to CPU Partitions”.

- `oss_cpts=[EXPRESSION]` binds the default OSS service on CPTs defined by `[EXPRESSION]`.
- `oss_io_cpts=[EXPRESSION]` binds the IO OSS service on CPTs defined by `[EXPRESSION]`.

For further details, see Section 32.9, “Setting MDS and OSS Thread Counts”.

26.1.2. Specifying the MDS Service Thread Count

The `mds_num_threads` parameter enables the number of MDS service threads to be specified at module load time on the MDS node:

```
options mds mds_num_threads={N}
```

After startup, the minimum and maximum number of MDS thread counts can be set via the `{service}.thread_{min,max,started}` tunable. To change the tunable at runtime, run:

```
lctl {get,set}_param {service}.thread_{min,max,started}
```

For details, see Section 32.9, “Setting MDS and OSS Thread Counts”.

At this time, no testing has been done to determine the optimal number of MDS threads. The default value varies, based on server size, up to a maximum of 32. The maximum number of threads (`MDS_MAX_THREADS`) is 512.

Note

The OSS and MDS automatically start new service threads dynamically, in response to server load within a factor of 4. The default value is calculated the same way as before. Setting the `_mu_threads` module parameter disables automatic thread creation behavior.

Lustre software release 2.3 introduced new parameters to provide more control to administrators.

- `mds_rdpg_num_threads` controls the number of threads in providing the read page service. The read page service handles file close and readdir operations.
- `mds_attr_num_threads` controls the number of threads in providing the setattr service to clients running Lustre software release 1.8.

Note

Default values for the thread counts are automatically selected. The values are chosen to best exploit the number of CPUs present in the system and to provide best overall performance for typical workloads.

Introduced in Lustre 2.3

With the introduction of Node Affinity (Node affinity) in Lustre software release 2.3, MDS threads can be bound to particular CPU partitions (CPTs). Default values for bindings are selected automatically to provide good overall performance for a given CPU count. However, an administrator can deviate from these setting if they choose.

- `mds_num_cpts=[EXPRESSION]` binds the default MDS service threads to CPTs defined by `EXPRESSION`. For example `mdt_num_cpts=[0-3]` will bind the MDS service threads to `CPT[0,1,2,3]`.
- `mds_rdpg_num_cpts=[EXPRESSION]` binds the read page service threads to CPTs defined by `EXPRESSION`. The read page service handles file close and readdir requests. For example `mdt_rdpg_num_cpts=[4]` will bind the read page threads to `CPT4`.
- `mds_attr_num_cpts=[EXPRESSION]` binds the setattr service threads to CPTs defined by `EXPRESSION`.

26.3. Tuning LNET Parameters

This section describes LNET tunables, the use of which may be necessary on some systems to improve performance. To test the performance of your Lustre network, see Chapter 24, *Testing Lustre Network Performance (LNET Self-Test)*.

26.3.1. Transmit and Receive Buffer Size

The kernel allocates buffers for sending and receiving messages on a network.

`ksocklnd` has separate parameters for the transmit and receive buffers.

```
options ksocklnd tx_buffer_size=0 rx_buffer_size=0
```

If these parameters are left at the default value (0), the system automatically tunes the transmit and receive buffer size. In almost every case, this default produces the best performance. Do not attempt to tune these parameters unless you are a network expert.

26.3.2. Hardware Interrupts (enable_irq_affinity)

The hardware interrupts that are generated by network adapters may be handled by any CPU in the system. In some cases, we would like network traffic to remain local to a single CPU to help keep the processor cache warm and minimize the impact of context switches. This is helpful when an SMP system has more than one network interface and ideal when the number of interfaces equals the number of CPUs. To enable the `enable_irq_affinity` parameter, enter:

```
options ksocklnd enable_irq_affinity=1
```

In other cases, if you have an SMP platform with a single fast interface such as 10 Gb Ethernet and more than two CPUs, you may see performance improve by turning this parameter off.

```
options ksocklnd enable_irq_affinity=0
```

By default, this parameter is off. As always, you should test the performance to compare the impact of changing this parameter.

Introduced in Lustre 2.3

Lustre software release 2.3 and beyond provide enhanced network interface control. The enhancement means that an administrator can bind an interface to one or more CPU partitions. Bindings are specified as options to the LNET modules. For more information on specifying module options, see Section 36.1, " Introduction"

For example, `o2ib0(ib0)[0,1]` will ensure that all messages for `o2ib0` will be handled by LND threads executing on `CPT0` and `CPT1`. An additional example might be: `tcp1(eth0)[0]`. Messages for `tcp1` are handled by threads on `CPT0`.

26.3.4. Network Interface Credits

Network interface (NI) credits are shared across all CPU partitions (CPT). For example, if a machine has four CPTs and the number of NI credits is 512, then each partition has 128 credits. If a large number of CPTs exist on the system, LNET checks and validates the NI credits for each CPT to ensure each CPT has a workable number of credits. For example, if a machine has 16 CPTs and the number of NI credits is 256, then each partition only has 16 credits. 16 NI credits is low and could negatively impact performance. As a result, LNET automatically adjusts the credits to 8*`peer_credits` (`peer_credits` is 8 by default), so each partition has 64 credits.

Increasing the number of `credits`/`peer_credits` can improve the performance of high latency networks (at the cost of consuming more memory) by enabling LNET to send more inflight messages to a specific network/peer and keep the pipeline saturated.

An administrator can modify the NI credit count using `ksoclnd` or `ko2iblnd`. In the example below, 256 credits are applied to TCP connections.

```
ksocklnd credits=256
```

Applying 256 credits to IB connections can be achieved with:

```
ko2iblnd credits=256
```

Introduced in Lustre 2.3

In Lustre software release 2.3 and beyond, LNET may revalidate the NI credits, so the administrator's request may not persist.

26.3.5. Router Buffers

When a node is set up as an LNET router, three pools of buffers are allocated: tiny, small and large. These pools are allocated per CPU partition and are used to buffer messages that arrive at the router to be forwarded to the next hop. The three different buffer sizes accommodate different size messages.

If a message arrives that can fit in a tiny buffer then a tiny buffer is used, if a message doesn't fit in a tiny buffer, but fits in a small buffer, then a small buffer is used. Finally if a message does not fit in either a tiny buffer or a small buffer, a large buffer is used.

Router buffers are shared by all CPU partitions. For a machine with a large number of CPTs, the router buffer number may need to be specified manually for best performance. A low number of router buffers risks starving the CPU partitions of resources.

- `tiny_router_buffers`: Zero payload buffers used for signals and acknowledgements.
- `small_router_buffers`: 4 KB payload buffers for small messages
- `large_router_buffers`: 1 MB maximum payload buffers, corresponding to the recommended RPC size of 1 MB.

The default setting for router buffers typically results in acceptable performance. LNET automatically sets a default value to reduce the likelihood of resource starvation. The size of a router buffer can be modified as shown in the example below. In this example, the size of the large buffer is modified using the `large_router_buffers` parameter.

```
lnet large_router_buffers=8192
```

Introduced in Lustre 2.3

In Lustre software release 2.3 and beyond, LNET may revalidate the router buffer setting, so the administrator's request may not persist.

26.3.6. Portal Round-Robin

Portal round-robin defines the policy LNET applies to deliver events and messages to the upper layers. The upper layers are PLRPC service or LNET selftest.

If portal round-robin is disabled, LNET will deliver messages to CPTs based on a hash of the source NID. Hence, all messages from a specific peer will be handled by the same CPT. This can reduce data traffic between CPUs. However, for some workloads, this behavior may result in poorly balancing loads across the CPU.

If portal round-robin is enabled, LNET will round-robin incoming events across all CPTs. This may balance load better across the CPU but can incur a cross CPU overhead.

The current policy can be changed by an administrator with `echo` *`value`* `> /proc/sys/lnet/portal_rotor`. There are four options for *`value`*:

- `OFF`

  Disable portal round-robin on all incoming requests.

- `ON`

  Enable portal round-robin on all incoming requests.

- RR_RT

  Enable portal round-robin only for routed messages.

- HASH_RT

  Routed messages will be delivered to the upper layer by hash of source NID (instead of NID of router.) This is the default value.

26.3.7. LNET Peer Health

Two options are available to help determine peer health:

- peer_timeout - The timeout (in seconds) before an aliveness query is sent to a peer. For example, if peer_timeout is set to 180sec, an aliveness query is sent to the peer every 180 seconds. This feature only takes effect if the node is configured as an LNET router.

  In a routed environment, the peer_timeout feature should always be on (set to a value in seconds) on routers. If the router checker has been enabled, the feature should be turned off by setting it to 0 on clients and servers.

  For a non-routed scenario, enabling the peer_timeout option provides health information such as whether a peer is alive or not. For example, a client is able to determine if an MGS or OST is up when it sends it a message. If a response is received, the peer is alive; otherwise a timeout occurs when the request is made.

  In general, peer_timeout should be set to no less than the LND timeout setting. For more information about LND timeouts, see Section 32.5.2, “Setting Static Timeouts”.

  When the o2iblnd (IB) driver is used, peer_timeout should be at least twice the value of the ko2iblnd keepalive option. for more information about keepalive options, see Section 36.2.2, “ SOCKLND Kernel TCP/IP LND”.

- avoid_asym_router_failure – When set to 1, the router checker running on the client or a server periodically pings all the routers corresponding to the NIDs identified in the routes parameter setting on the node to determine the status of each router interface. The default setting is 1. (For more information about the LNET routes parameter, see Section 9.4, “Using the routes Parameter”

  A router is considered down if any of its NIDs are down. For example, router X has three NIDs: Xnid1, Xnid2, and Xnid3. A client is connected to the router via Xnid1. The client has router checker enabled. The router checker periodically sends a ping to the router via Xnid1. The router responds to the ping with the status of each of its NIDs. In this case, it responds with Xnid1=up, Xnid2=up, Xnid3=down. If avoid_asym_router_failure==1, the router is considered down if any of its NIDs are down, so router X is considered down and will not be used for routing messages. If avoid_asym_router_failure==0, router X will continue to be used for routing messages.

The following router checker parameters must be set to the maximum value of the corresponding setting for this option on any client or server:

- dead_router_check_interval
- live_router_check_interval
- router_ping_timeout

For example, the dead_router_check_interval parameter on any router must be MAX.

26.4. libcfs Tuning

By default, the Lustre software will automatically generate CPU partitions (CPT) based on the number of CPUs in the system. The CPT number will be 1 if the online CPU number is less than five.

The CPT number can be explicitly set on the libcfs module using `cpu_npartitions=NUMBER`. The value of `cpu_npartitions` must be an integer between 1 and the number of online CPUs.

Tip

Setting CPT to 1 will disable most of the SMP Node Affinity functionality.

26.4.1. CPU Partition String Patterns

CPU partitions can be described using string pattern notation. For example:

- `cpu_pattern="0[0,2,4,6] 1[1,3,5,7]`

  Create two CPTs, CPT0 contains CPU[0, 2, 4, 6]. CPT1 contains CPU[1,3,5,7].

- `cpu_pattern="N 0[0-3] 1[4-7]`

  Create two CPTs, CPT0 contains all CPUs in NUMA node[0-3], CPT1 contains all CPUs in NUMA node [4-7].

The current configuration of the CPU partition can be read from `/proc/sys/lnet/cpu_partitions`

26.5. LND Tuning

LND tuning allows the number of threads per CPU partition to be specified. An administrator can set the threads for both `ko2iblnd` and `ksocklnd` using the `nscheds` parameter. This adjusts the number of threads for each partition, not the overall number of threads on the LND.

Note

Lustre software release 2.3 has greatly decreased the default number of threads for `ko2iblnd` and `ksocklnd` on high-core count machines. The current default values are automatically set and are chosen to work well across a number of typical scenarios.

Introduced in Lustre 2.4

The Network Request Scheduler (NRS) allows the administrator to influence the order in which RPCs are handled at servers, on a per-PTLRPC service basis, by providing different policies that can be activated and tuned in order to influence the RPC ordering. The aim of this is to provide for better performance, and possibly discrete performance characteristics using future policies.

The NRS policy state of a PTLRPC service can be read and set via the `{service}.nrs_policies` tunable. To read a PTLRPC service's NRS policy state, run:

```
lctl get_param {service}.nrs_policies
```

For example, to read the NRS policy state of the `ost_io` service, run:

```
$ lctl get_param ost.OSS.ost_io.nrs_policies
ost.OSS.ost_io.nrs_policies=
```

```
regular_requests:
  - name: fifo
    state: started
    fallback: yes
    queued: 0
    active: 0

  - name: crrn
    state: stopped
    fallback: no
    queued: 0
    active: 0

  - name: orr
    state: stopped
    fallback: no
    queued: 0
    active: 0

  - name: trr
    state: started
    fallback: no
    queued: 2420
    active: 268

high_priority_requests:
  - name: fifo
    state: started
    fallback: yes
    queued: 0
    active: 0

  - name: crrn
    state: stopped
    fallback: no
    queued: 0
    active: 0

  - name: orr
    state: stopped
    fallback: no
    queued: 0
    active: 0

  - name: trr
    state: stopped
    fallback: no
    queued: 0
    active: 0
```

NRS policy state is shown in either one or two sections, depending on the PTLRPC service being queried. The first section is named `regular_requests` and is available for all PTLRPC services, optionally

followed by a second section which is named `high_priority_requests`. This is because some PTLRPC services are able to treat some types of RPCs as higher priority ones, such that they are handled by the server with higher priority compared to other, regular RPC traffic. For PTLRPC services that do not support high-priority RPCs, you will only see the `regular_requests` section.

There is a separate instance of each NRS policy on each PTLRPC service for handling regular and high-priority RPCs (if the service supports high-priority RPCs). For each policy instance, the following fields are shown:

Field	Description
`name`	The name of the policy.
`state`	The state of the policy; this can be any of `invalid`, `stopping`, `stopped`, `starting`, `started`. A fully enabled policy is in the `started` state.
`fallback`	Whether the policy is acting as a fallback policy or not. A fallback policy is used to handle RPCs that other enabled policies fail to handle, or do not support the handling of. The possible values are `no`, `yes`. Currently, only the FIFO policy can act as a fallback policy.
`queued`	The number of RPCs that the policy has waiting to be serviced.
`active`	The number of RPCs that the policy is currently handling.

To enable an NRS policy on a PTLRPC service run:

```
lctl set_param {service}.nrs_policies=policy_name
```

This will enable the policy *`policy_name`* for both regular and high-priority RPCs (if the PLRPC service supports high-priority RPCs) on the given service. For example, to enable the CRR-N NRS policy for the ldlm_cbd service, run:

```
$ lctl set_param ldlm.services.ldlm_cbd.nrs_policies=crrn
ldlm.services.ldlm_cbd.nrs_policies=crrn
```

For PTLRPC services that support high-priority RPCs, you can also supply an optional *`reg|hp`* token, in order to enable an NRS policy for handling only regular or high-priority RPCs on a given PTLRPC service, by running:

```
lctl set_param {service}.nrs_policies="policy_name reg|hp"
```

For example, to enable the TRR policy for handling only regular, but not high-priority RPCs on the `ost_io` service, run:

```
$ lctl set_param ost.OSS.ost_io.nrs_policies="trr reg"
ost.OSS.ost_io.nrs_policies="trr reg"
```

Note

When enabling an NRS policy, the policy name must be given in lower-case characters, otherwise the operation will fail with an error message.

26.6.1. First In, First Out (FIFO) policy

The first in, first out (FIFO) policy handles RPCs in a service in the same order as they arrive from the LNET layer, so no special processing takes place to modify the RPC handling stream. FIFO is the default policy for all types of RPCs on all PTLRPC services, and is always enabled irrespective of the state of other policies, so that it can be used as a backup policy, in case a more elaborate policy that has been enabled fails to handle an RPC, or does not support handling a given type of RPC.

The FIFO policy has no tunables that adjust its behaviour.

26.6.2. Client Round-Robin over NIDs (CRR-N) policy

The client round-robin over NIDs (CRR-N) policy performs batched round-robin scheduling of all types of RPCs, with each batch consisting of RPCs originating from the same client node, as identified by its NID. CRR-N aims to provide for better resource utilization across the cluster, and to help shorten completion times of jobs in some cases, by distributing available bandwidth more evenly across all clients.

The CRR-N policy can be enabled on all types of PTLRPC services, and has the following tunable that can be used to adjust its behavior:

- `{service}.nrs_crrn_quantum`

  The `{service}.nrs_crrn_quantum` tunable determines the maximum allowed size of each batch of RPCs; the unit of measure is in number of RPCs. To read the maximum allowed batch size of a CRR-N policy, run:

  ```
  lctl get_param {service}.nrs_crrn_quantum
  ```

  For example, to read the maximum allowed batch size of a CRR-N policy on the ost_io service, run:

  ```
  $ lctl get_param ost.OSS.ost_io.nrs_crrn_quantum
  ost.OSS.ost_io.nrs_crrn_quantum=reg_quantum:16
  hp_quantum:8
  ```

  You can see that there is a separate maximum allowed batch size value for regular (`reg_quantum`) and high-priority (`hp_quantum`) RPCs (if the PTLRPC service supports high-priority RPCs).

  To set the maximum allowed batch size of a CRR-N policy on a given service, run:

  ```
  lctl set_param {service}.nrs_crrn_quantum=1-65535
  ```

  This will set the maximum allowed batch size on a given service, for both regular and high-priority RPCs (if the PLRPC service supports high-priority RPCs), to the indicated value.

  For example, to set the maximum allowed batch size on the ldlm_canceld service to 16 RPCs, run:

  ```
  $ lctl set_param ldlm.services.ldlm_canceld.nrs_crrn_quantum=16
  ldlm.services.ldlm_canceld.nrs_crrn_quantum=16
  ```

  For PTLRPC services that support high-priority RPCs, you can also specify a different maximum allowed batch size for regular and high-priority RPCs, by running:

```
$ lctl set_param {service}.nrs_crrn_quantum=reg_quantum|hp_quantum:1-65535"
```

For example, to set the maximum allowed batch size on the ldlm_canceld service, for high-priority RPCs to 32, run:

```
$ lctl set_param ldlm.services.ldlm_canceld.nrs_crrn_quantum="hp_quantum:32"
ldlm.services.ldlm_canceld.nrs_crrn_quantum=hp_quantum:32
```

By using the last method, you can also set the maximum regular and high-priority RPC batch sizes to different values, in a single command invocation.

26.6.3. Object-based Round-Robin (ORR) policy

The object-based round-robin (ORR) policy performs batched round-robin scheduling of bulk read write (brw) RPCs, with each batch consisting of RPCs that pertain to the same backend-file system object, as identified by its OST FID.

The ORR policy is only available for use on the ost_io service. The RPC batches it forms can potentially consist of mixed bulk read and bulk write RPCs. The RPCs in each batch are ordered in an ascending manner, based on either the file offsets, or the physical disk offsets of each RPC (only applicable to bulk read RPCs).

The aim of the ORR policy is to provide for increased bulk read throughput in some cases, by ordering bulk read RPCs (and potentially bulk write RPCs), and thus minimizing costly disk seek operations. Performance may also benefit from any resulting improvement in resource utilization, or by taking advantage of better locality of reference between RPCs.

The ORR policy has the following tunables that can be used to adjust its behaviour:

- `ost.OSS.ost_io.nrs_orr_quantum`

  The `ost.OSS.ost_io.nrs_orr_quantum` tunable determines the maximum allowed size of each batch of RPCs; the unit of measure is in number of RPCs. To read the maximum allowed batch size of the ORR policy, run:

  ```
  $ lctl get_param ost.OSS.ost_io.nrs_orr_quantum
  ost.OSS.ost_io.nrs_orr_quantum=reg_quantum:256
  hp_quantum:16
  ```

  You can see that there is a separate maximum allowed batch size value for regular (`reg_quantum`) and high-priority (`hp_quantum`) RPCs (if the PTLRPC service supports high-priority RPCs).

  To set the maximum allowed batch size for the ORR policy, run:

  ```
  $ lctl set_param ost.OSS.ost_io.nrs_orr_quantum=1-65535
  ```

  This will set the maximum allowed batch size for both regular and high-priority RPCs, to the indicated value.

  You can also specify a different maximum allowed batch size for regular and high-priority RPCs, by running:

  ```
  $ lctl set_param ost.OSS.ost_io.nrs_orr_quantum=reg_quantum|hp_quantum:1-65535
  ```

For example, to set the maximum allowed batch size for regular RPCs to 128, run:

```
$ lctl set_param ost.OSS.ost_io.nrs_orr_quantum=reg_quantum:128
ost.OSS.ost_io.nrs_orr_quantum=reg_quantum:128
```

By using the last method, you can also set the maximum regular and high-priority RPC batch sizes to different values, in a single command invocation.

- `ost.OSS.ost_io.nrs_orr_offset_type`

  The `ost.OSS.ost_io.nrs_orr_offset_type` tunable determines whether the ORR policy orders RPCs within each batch based on logical file offsets or physical disk offsets. To read the offset type value for the ORR policy, run:

  ```
  $ lctl get_param ost.OSS.ost_io.nrs_orr_offset_type
  ost.OSS.ost_io.nrs_orr_offset_type=reg_offset_type:physical
  hp_offset_type:logical
  ```

  You can see that there is a separate offset type value for regular (`reg_offset_type`) and high-priority (`hp_offset_type`) RPCs.

  To set the ordering type for the ORR policy, run:

  ```
  $ lctl set_param ost.OSS.ost_io.nrs_orr_offset_type=physical|logical
  ```

  This will set the offset type for both regular and high-priority RPCs, to the indicated value.

  You can also specify a different offset type for regular and high-priority RPCs, by running:

  ```
  $ lctl set_param ost.OSS.ost_io.nrs_orr_offset_type=reg_offset_type|hp_offset_ty
  ```

  For example, to set the offset type for high-priority RPCs to physical disk offsets, run:

  ```
  $ lctl set_param ost.OSS.ost_io.nrs_orr_offset_type=hp_offset_type:physical
  ost.OSS.ost_io.nrs_orr_offset_type=hp_offset_type:physical
  ```

  By using the last method, you can also set offset type for regular and high-priority RPCs to different values, in a single command invocation.

  ### Note

  Irrespective of the value of this tunable, only logical offsets can, and are used for ordering bulk write RPCs.

- `ost.OSS.ost_io.nrs_orr_supported`

  The `ost.OSS.ost_io.nrs_orr_supported` tunable determines the type of RPCs that the ORR policy will handle. To read the types of supported RPCs by the ORR policy, run:

  ```
  $ lctl get_param ost.OSS.ost_io.nrs_orr_supported
  ost.OSS.ost_io.nrs_orr_supported=reg_supported:reads
  hp_supported=reads_and_writes
  ```

You can see that there is a separate supported 'RPC types' value for regular (`reg_supported`) and high-priority (`hp_supported`) RPCs.

To set the supported RPC types for the ORR policy, run:

```
$ lctl set_param ost.OSS.ost_io.nrs_orr_supported=reads|writes|reads_and_writes
```

This will set the supported RPC types for both regular and high-priority RPCs, to the indicated value.

You can also specify a different supported 'RPC types' value for regular and high-priority RPCs, by running:

```
$ lctl set_param ost.OSS.ost_io.nrs_orr_supported=reg_supported|hp_supported:rea
```

For example, to set the supported RPC types to bulk read and bulk write RPCs for regular requests, run:

```
$ lctl set_param ost.OSS.ost_io.nrs_orr_supported=reg_supported:reads_and_writes
ost.OSS.ost_io.nrs_orr_supported=reg_supported:reads_and_writes
```

By using the last method, you can also set the supported RPC types for regular and high-priority RPC to different values, in a single command invocation.

26.6.4. Target-based Round-Robin (TRR) policy

The target-based round-robin (TRR) policy performs batched round-robin scheduling of brw RPCs, with each batch consisting of RPCs that pertain to the same OST, as identified by its OST index.

The TRR policy is identical to the object-based round-robin (ORR) policy, apart from using the brw RPC's target OST index instead of the backend-fs object's OST FID, for determining the RPC scheduling order. The goals of TRR are effectively the same as for ORR, and it uses the following tunables to adjust its behaviour:

- `ost.OSS.ost_io.nrs_trr_quantum`

  The purpose of this tunable is exactly the same as for the `ost.OSS.ost_io.nrs_orr_quantum` tunable for the ORR policy, and you can use it in exactly the same way.

- `ost.OSS.ost_io.nrs_trr_offset_type`

  The purpose of this tunable is exactly the same as for the `ost.OSS.ost_io.nrs_orr_offset_type` tunable for the ORR policy, and you can use it in exactly the same way.

- `ost.OSS.ost_io.nrs_trr_supported`

  The purpose of this tunable is exactly the same as for the `ost.OSS.ost_io.nrs_orr_supported` tunable for the ORR policy, and you can use it in exactly the sme way.

26.7. Lockless I/O Tunables

The lockless I/O tunable feature allows servers to ask clients to do lockless I/O (liblustre-style where the server does the locking) on contended files.

The lockless I/O patch introduces these tunables:

- **OST-side:**

  `/proc/fs/lustre/ldlm/namespaces/filter-lustre-*`

  `contended_locks` - If the number of lock conflicts in the scan of granted and waiting queues at contended_locks is exceeded, the resource is considered to be contended.

  `contention_seconds` - The resource keeps itself in a contended state as set in the parameter.

  `max_nolock_bytes` - Server-side locking set only for requests less than the blocks set in the `max_nolock_bytes` parameter. If this tunable is set to zero (0), it disables server-side locking for read/write requests.

- **Client-side:**

  `/proc/fs/lustre/llite/lustre-*`

  `contention_seconds` - `llite` inode remembers its contended state for the time specified in this parameter.

- **Client-side statistics:**

  The `/proc/fs/lustre/llite/lustre-*/stats` file has new rows for lockless I/O statistics.

  `lockless_read_bytes` and `lockless_write_bytes` - To count the total bytes read or written, the client makes its own decisions based on the request size. The client does not communicate with the server if the request size is smaller than the `min_nolock_size`, without acquiring locks by the client.

26.8. Improving Lustre File System Performance When Working with Small Files

An environment where an application writes small file chunks from many clients to a single file will result in bad I/O performance. To improve the performance of the Lustre file system with small files:

- Have the application aggregate writes some amount before submitting them to the Lustre file system. By default, the Lustre software enforces POSIX coherency semantics, so it results in lock ping-pong between client nodes if they are all writing to the same file at one time.

- Have the application do 4kB `O_DIRECT` sized I/O to the file and disable locking on the output file. This avoids partial-page IO submissions and, by disabling locking, you avoid contention between clients.

- Have the application write contiguous data.

- Add more disks or use SSD disks for the OSTs. This dramatically improves the IOPS rate. Consider creating larger OSTs rather than many smaller OSTs due to less overhead (journal, connections, etc).

- Use RAID-1+0 OSTs instead of RAID-5/6. There is RAID parity overhead for writing small chunks of data to disk.

26.9. Understanding Why Write Performance is Better Than Read Performance

Typically, the performance of write operations on a Lustre cluster is better than read operations. When doing writes, all clients are sending write RPCs asynchronously. The RPCs are allocated, and written to disk in the order they arrive. In many cases, this allows the back-end storage to aggregate writes efficiently.

In the case of read operations, the reads from clients may come in a different order and need a lot of seeking to get read from the disk. This noticeably hampers the read throughput.

Currently, there is no readahead on the OSTs themselves, though the clients do readahead. If there are lots of clients doing reads it would not be possible to do any readahead in any case because of memory consumption (consider that even a single RPC (1 MB) readahead for 1000 clients would consume 1 GB of RAM).

For file systems that use socklnd (TCP, Ethernet) as interconnect, there is also additional CPU overhead because the client cannot receive data without copying it from the network buffers. In the write case, the client CAN send data without the additional data copy. This means that the client is more likely to become CPU-bound during reads than writes.

Part V. Troubleshooting a Lustre File System

Part V provides information about troubleshooting a Lustre file system. You will find information in this section about:

- Lustre File System Troubleshooting
- Troubleshooting Recovery
- Debugging a Lustre File System

Chapter 27. Lustre File System Troubleshooting

This chapter provides information about troubleshooting a Lustre file system, submitting a bug to the Jira bug tracking system, and Lustre file system performance tips. It includes the following sections:

- Section 27.1, " Lustre Error Messages"
- Section 27.2, " Reporting a Lustre File System Bug"
- Section 27.3, "Common Lustre File System Problems"

27.1. Lustre Error Messages

Several resources are available to help troubleshoot an issue in a Lustre file system. This section describes error numbers, error messages and logs.

27.1.1. Error Numbers

Error numbers are generated by the Linux operating system and are located in `/usr/include/asm-generic/errno.h`. The Lustre software does not use all of the available Linux error numbers. The exact meaning of an error number depends on where it is used. Here is a summary of the basic errors that Lustre file system users may encounter.

Error Number	Error Name	Description
-1	`-EPERM`	Permission is denied.
-2	`-ENOENT`	The requested file or directory does not exist.
-4	`-EINTR`	The operation was interrupted (usually CTRL-C or a killing process).
-5	`-EIO`	The operation failed with a read or write error.
-19	`-ENODEV`	No such device is available. The server stopped or failed over.
-22	`-EINVAL`	The parameter contains an invalid value.
-28	`-ENOSPC`	The file system is out-of-space or out of inodes. Use `lfs df` (query the amount of file system space) or `lfs df -i` (query the number of inodes).
-30	`-EROFS`	The file system is read-only, likely due to a detected error.
-43	`-EIDRM`	The UID/GID does not match any known UID/GID on the MDS. Update etc/hosts and etc/group on

Error Number	Error Name	Description
		the MDS to add the missing user or group.
-107	-ENOTCONN	The client is not connected to this server.
-110	-ETIMEDOUT	The operation took too long and timed out.
-122	-EDQUOT	The operation exceeded the user disk quota and was aborted.

27.1.2. Viewing Error Messages

As Lustre software code runs on the kernel, single-digit error codes display to the application; these error codes are an indication of the problem. Refer to the kernel console log (dmesg) for all recent kernel messages from that node. On the node, `/var/log/messages` holds a log of all messages for at least the past day.

The error message initiates with "LustreError" in the console log and provides a short description of:

- What the problem is
- Which process ID had trouble
- Which server node it was communicating with, and so on.

Lustre logs are dumped to `/proc/sys/lnet/debug_path`.

Collect the first group of messages related to a problem, and any messages that precede "LBUG" or "assertion failure" errors. Messages that mention server nodes (OST or MDS) are specific to that server; you must collect similar messages from the relevant server console logs.

Another Lustre debug log holds information for a short period of time for action by the Lustre software, which, in turn, depends on the processes on the Lustre node. Use the following command to extract debug logs on each of the nodes, run

```
$ lctl dk filename
```

Note

LBUG freezes the thread to allow capture of the panic stack. A system reboot is needed to clear the thread.

27.2. Reporting a Lustre File System Bug

If you cannot resolve a problem by troubleshooting your Lustre file system, other options are:

- Post a question to the hppd-discuss [https://lists.01.org/mailman/listinfo/hpdd-discuss] email list or search the archives for information about your issue.
- Submit a ticket to the Jira [https://jira.hpdd.intel.com/secure/Dashboard.jspa][*] bug tracking and project management tool used for the Lustre software project. If you are a first-time user, you'll need to open an account by clicking on **Sign up** on the Welcome page.

To submit a Jira ticket, follow these steps:

1. To avoid filing a duplicate ticket, search for existing tickets for your issue. *For search tips, see Section 27.2.1, “Searching the Jira* Bug Tracker for Duplicate Tickets”*.

2. To create a ticket, click **+Create Issue** in the upper right corner. *Create a separate ticket for each issue you wish to submit.*

3. In the form displayed, enter the following information:

   - *Project* - Select **Lustre** or **Lustre Documentation** or an appropriate project.
   - *Issue type* - Select **Bug**.
   - *Summary* - Enter a short description of the issue. Use terms that would be useful for someone searching for a similar issue. A LustreError or ASSERT/panic message often makes a good summary.
   - *Affects version(s)* - Select your Lustre release.
   - *Environment* - Enter your kernel with version number.
   - *Description* - Include a detailed description of *visible symptoms* and, if possible, *how the problem is produced*. Other useful information may include *the behavior you expect to see* and *what you have tried so far to diagnose the problem*.
   - *Attachments* - Attach log sources such as Lustre debug log dumps (see Section 29.1, “ Diagnostic and Debugging Tools”), syslogs, or console logs. ***Note:*** Lustre debug logs must be processed using `lctl df` prior to attaching to a Jira ticket. For more information, see Section 29.2.2, “Using the lctl Tool to View Debug Messages”.

   Other fields in the form are used for project tracking and are irrelevant to reporting an issue. You can leave these in their default state.

27.2.1. Searching the Jira* Bug Tracker for Duplicate Tickets

Before submitting a ticket, always search the Jira bug tracker for an existing ticket for your issue.This avoids duplicating effort and may immediately provide you with a solution to your problem.

To do a search in the Jira bug tracker, select the **Issues** tab and click on **New filter**. Use the filters provided to select criteria for your search. To search for specific text, enter the text in the "Contains text" field and click the magnifying glass icon.

When searching for text such as an ASSERTION or LustreError message, you can remove NIDS and other installation-specific text from your search string by following the example below.

Original error message:

```
"(filter_io_26.c:791:filter_commitrw_write())           ASSERTION(oti-
>oti_transno   <=obd->obd_last_committed)   failed:   oti_transno   752
last_committed 750"
```

Optimized search string :

```
"(filter_io_26.c:"   AND ":filter_commitrw_write())   ASSERTION(oti-
>oti_transno <=obd->obd_last_committed) failed:
```

27.3. Common Lustre File System Problems

This section describes how to address common issues encountered with a Lustre file system.

27.3.1. OST Object is Missing or Damaged

If the OSS fails to find an object or finds a damaged object, this message appears:

```
OST object missing or damaged (OST "ost1", object 98148, error -2)
```

If the reported error is -2 (`-ENOENT`, or "No such file or directory"), then the object is missing. This can occur either because the MDS and OST are out of sync, or because an OST object was corrupted and deleted.

If you have recovered the file system from a disk failure by using e2fsck, then unrecoverable objects may have been deleted or moved to /lost+found on the raw OST partition. Because files on the MDS still reference these objects, attempts to access them produce this error.

If you have recovered a backup of the raw MDS or OST partition, then the restored partition is very likely to be out of sync with the rest of your cluster. No matter which server partition you restored from backup, files on the MDS may reference objects which no longer exist (or did not exist when the backup was taken); accessing those files produces this error.

If neither of those descriptions is applicable to your situation, then it is possible that you have discovered a programming error that allowed the servers to get out of sync. Please submit a Jira ticket (see Section 27.2, “ Reporting a Lustre File System Bug”).

If the reported error is anything else (such as -5, "`I/O error`"), it likely indicates a storage failure. The low-level file system returns this error if it is unable to read from the storage device.

Suggested Action

If the reported error is -2, you can consider checking in `/lost+found` on your raw OST device, to see if the missing object is there. However, it is likely that this object is lost forever, and that the file that references the object is now partially or completely lost. Restore this file from backup, or salvage what you can and delete it.

If the reported error is anything else, then you should immediately inspect this server for storage problems.

27.3.2. OSTs Become Read-Only

If the SCSI devices are inaccessible to the Lustre file system at the block device level, then `ldiskfs` remounts the device read-only to prevent file system corruption. This is a normal behavior. The status in `/proc/fs/lustre/health_check` also shows "not healthy" on the affected nodes.

To determine what caused the "not healthy" condition:

- Examine the consoles of all servers for any error indications
- Examine the syslogs of all servers for any LustreErrors or `LBUG`
- Check the health of your system hardware and network. (Are the disks working as expected, is the network dropping packets?)

- Consider what was happening on the cluster at the time. Does this relate to a specific user workload or a system load condition? Is the condition reproducible? Does it happen at a specific time (day, week or month)?

To recover from this problem, you must restart Lustre services using these file systems. There is no other way to know that the I/O made it to disk, and the state of the cache may be inconsistent with what is on disk.

27.3.3. Identifying a Missing OST

If an OST is missing for any reason, you may need to know what files are affected. Although an OST is missing, the files system should be operational. From any mounted client node, generate a list of files that reside on the affected OST. It is advisable to mark the missing OST as 'unavailable' so clients and the MDS do not time out trying to contact it.

1. Generate a list of devices and determine the OST's device number. Run:

   ```
   $ lctl dl
   ```

   The lctl dl command output lists the device name and number, along with the device UUID and the number of references on the device.

2. Deactivate the OST (on the OSS at the MDS). Run:

   ```
   $ lctl --device lustre_device_number deactivate
   ```

   The OST device number or device name is generated by the lctl dl command.

   The `deactivate` command prevents clients from creating new objects on the specified OST, although you can still access the OST for reading.

   ### Note

   If the OST later becomes available it needs to be reactivated, run:

   ```
   # lctl --device lustre_device_number activate
   ```

3. Determine all files that are striped over the missing OST, run:

   ```
   # lfs getstripe -r -O {OST_UUID} /mountpoint
   ```

   This returns a simple list of filenames from the affected file system.

4. If necessary, you can read the valid parts of a striped file, run:

   ```
   # dd if=filename of=new_filename bs=4k conv=sync,noerror
   ```

5. You can delete these files with the unlink or munlink command.

   ```
   # unlink|munlink filename {filename ...}
   ```

   ### Note

   There is no functional difference between the `unlink` and `munlink` commands. The unlink command is for newer Linux distributions. You can run `munlink` if `unlink` is not available.

   When you run the `unlink` or `munlink` command, the file on the MDS is permanently removed.

6. If you need to know, specifically, which parts of the file are missing data, then you first need to determine the file layout (striping pattern), which includes the index of the missing OST). Run:

   ```
   # lfs getstripe -v {filename}
   ```

7. Use this computation is to determine which offsets in the file are affected: [(C*N + X)*S, (C*N + X)*S + S - 1], N = { 0, 1, 2, ...}

   where:

   C = stripe count

   S = stripe size

   X = index of bad OST for this file

For example, for a 2 stripe file, stripe size = 1M, the bad OST is at index 0, and you have holes in the file at: [(2*N + 0)*1M, (2*N + 0)*1M + 1M - 1], N = { 0, 1, 2, ...}

If the file system cannot be mounted, currently there is no way that parses metadata directly from an MDS. If the bad OST does not start, options to mount the file system are to provide a loop device OST in its place or replace it with a newly-formatted OST. In that case, the missing objects are created and are read as zero-filled.

27.3.4. Fixing a Bad LAST_ID on an OST

Each OST contains a LAST_ID file, which holds the last object (pre-)created by the MDS [1]. The MDT contains a lov_objid file, with values that represent the last object the MDS has allocated to a file.

During normal operation, the MDT keeps some pre-created (but unallocated) objects on the OST, and the relationship between LAST_ID and lov_objid should be LAST_ID <= lov_objid. Any difference in the file values results in objects being created on the OST when it next connects to the MDS. These objects are never actually allocated to a file, since they are of 0 length (empty), but they do no harm. Creating empty objects enables the OST to catch up to the MDS, so normal operations resume.

However, in the case where lov_objid < LAST_ID, bad things can happen as the MDS is not aware of objects that have already been allocated on the OST, and it reallocates them to new files, overwriting their existing contents.

Here is the rule to avoid this scenario:

LAST_ID >= lov_objid and LAST_ID == last_physical_object and lov_objid >= last_used_object

Although the lov_objid value should be equal to the last_used_object value, the above rule suffices to keep the Lustre file system happy at the expense of a few leaked objects.

In situations where there is on-disk corruption of the OST, for example caused by running with write cache enabled on the disks, the LAST_ID value may become inconsistent and result in a message similar to:

```
"filter_precreate()) HOME-OST0003: Serious error:
objid 3478673 already exists; is this filesystem corrupt?"
```

A related situation may happen if there is a significant discrepancy between the record of previously-created objects on the OST and the previously-allocated objects on the MDS, for example if the MDS has

[1] The contents of the LAST_ID file must be accurate regarding the actual objects that exist on the OST.

been corrupted, or restored from backup, which may cause significant data loss if left unchecked. This produces a message like:

```
"HOME-OST0003: ignoring bogus orphan destroy request:
obdid 3438673 last_id 3478673"
```

To recover from this situation, determine and set a reasonable LAST_ID value.

Note

The file system must be stopped on all servers before performing this procedure.

For hex-to-decimal translations:

Use GDB:

```
(gdb) p /x 15028
$2 = 0x3ab4
```

Or `bc`:

```
echo "obase=16; 15028" | bc
```

1. Determine a reasonable value for the LAST_ID file. Check on the MDS:

   ```
   # mount -t ldiskfs /dev/mdt_device /mnt/mds
   # od -Ax -td8 /mnt/mds/lov_objid
   ```

   There is one entry for each OST, in OST index order. This is what the MDS thinks is the last in-use object.

2. Determine the OST index for this OST.

   ```
   # od -Ax -td4 /mnt/ost/last_rcvd
   ```

   It will have it at offset 0x8c.

3. Check on the OST. Use debugfs to check the LAST_ID value:

   ```
   debugfs -c -R 'dump /O/0/LAST_ID /tmp/LAST_ID' /dev/XXX ; od -Ax -td8 /tmp/\
   LAST_ID"
   ```

4. Check the objects on the OST:

   ```
   mount -rt ldiskfs /dev/{ostdev} /mnt/ost
   # note the ls below is a number one and not a letter L
   ls -1s /mnt/ost/O/0/d* | grep -v [a-z] |
   sort -k2 -n > /tmp/objects.{diskname}

   tail -30 /tmp/objects.{diskname}
   ```

   This shows you the OST state. There may be some pre-created orphans. Check for zero-length objects. Any zero-length objects with IDs higher than LAST_ID should be deleted. New objects will be pre-created.

If the OST LAST_ID value matches that for the objects existing on the OST, then it is possible the lov_objid file on the MDS is incorrect. Delete the lov_objid file on the MDS and it will be re-created from the LAST_ID on the OSTs.

If you determine the LAST_ID file on the OST is incorrect (that is, it does not match what objects exist, does not match the MDS lov_objid value), then you have decided on a proper value for LAST_ID.

Once you have decided on a proper value for LAST_ID, use this repair procedure.

1. Access:

   ```
   mount -t ldiskfs /dev/{ostdev} /mnt/ost
   ```

2. Check the current:

   ```
   od -Ax -td8 /mnt/ost/O/0/LAST_ID
   ```

3. Be very safe, only work on backups:

   ```
   cp /mnt/ost/O/0/LAST_ID /tmp/LAST_ID
   ```

4. Convert binary to text:

   ```
   xxd /tmp/LAST_ID /tmp/LAST_ID.asc
   ```

5. Fix:

   ```
   vi /tmp/LAST_ID.asc
   ```

6. Convert to binary:

   ```
   xxd -r /tmp/LAST_ID.asc /tmp/LAST_ID.new
   ```

7. Verify:

   ```
   od -Ax -td8 /tmp/LAST_ID.new
   ```

8. Replace:

   ```
   cp /tmp/LAST_ID.new /mnt/ost/O/0/LAST_ID
   ```

9. Clean up:

   ```
   umount /mnt/ost
   ```

27.3.5. Handling/Debugging "`Bind: Address already in use`" Error

During startup, the Lustre software may report a `bind: Address already in use` error and reject to start the operation. This is caused by a portmap service (often NFS locking) that starts before the Lustre file system and binds to the default port 988. You must have port 988 open from firewall or IP tables for incoming connections on the client, OSS, and MDS nodes. LNET will create three outgoing connections on available, reserved ports to each client-server pair, starting with 1023, 1022 and 1021.

Unfortunately, you cannot set sunprc to avoid port 988. If you receive this error, do the following:

- Start the Lustre file system before starting any service that uses sunrpc.
- Use a port other than 988 for the Lustre file system. This is configured in `/etc/modprobe.d/lustre.conf` as an option to the LNET module. For example:

  ```
  options lnet accept_port=988
  ```

- Add modprobe ptlrpc to your system startup scripts before the service that uses sunrpc. This causes the Lustre file system to bind to port 988 and sunrpc to select a different port.

  Note

  You can also use the `sysctl` command to mitigate the NFS client from grabbing the Lustre service port. However, this is a partial workaround as other user-space RPC servers still have the ability to grab the port.

27.3.6. Handling/Debugging Error "- 28"

A Linux error -28 (`ENOSPC`) that occurs during a write or sync operation indicates that an existing file residing on an OST could not be rewritten or updated because the OST was full, or nearly full. To verify if this is the case, on a client on which the OST is mounted, enter :

```
lfs df -h
```

To address this issue, you can do one of the following:

- Expand the disk space on the OST.
- Copy or stripe the file to a less full OST.

A Linux error -28 (`ENOSPC`) that occurs when a new file is being created may indicate that the MDS has run out of inodes and needs to be made larger. Newly created files do not written to full OSTs, while existing files continue to reside on the OST where they were initially created. To view inode information on the MDS, enter:

```
lfs df -i
```

Typically, the Lustre software reports this error to your application. If the application is checking the return code from its function calls, then it decodes it into a textual error message such as `No space left on device`. Both versions of the error message also appear in the system log.

For more information about the `lfs df` command, see Section 18.5.1, “Checking File System Free Space”.

Although it is less efficient, you can also use the grep command to determine which OST or MDS is running out of space. To check the free space and inodes on a client, enter:

```
grep '[0-9]' /proc/fs/lustre/osc/*/kbytes{free,avail,total}
grep '[0-9]' /proc/fs/lustre/osc/*/files{free,total}
grep '[0-9]' /proc/fs/lustre/mdc/*/kbytes{free,avail,total}
grep '[0-9]' /proc/fs/lustre/mdc/*/files{free,total}
```

Note

You can find other numeric error codes along with a short name and text description in `/usr/include/asm/errno.h`.

27.3.7. Triggering Watchdog for PID NNN

In some cases, a server node triggers a watchdog timer and this causes a process stack to be dumped to the console along with a Lustre kernel debug log being dumped into `/tmp` (by default). The presence of

a watchdog timer does NOT mean that the thread OOPSed, but rather that it is taking longer time than expected to complete a given operation. In some cases, this situation is expected.

For example, if a RAID rebuild is really slowing down I/O on an OST, it might trigger watchdog timers to trip. But another message follows shortly thereafter, indicating that the thread in question has completed processing (after some number of seconds). Generally, this indicates a transient problem. In other cases, it may legitimately signal that a thread is stuck because of a software error (lock inversion, for example).

```
Lustre: 0:0:(watchdog.c:122:lcw_cb())
```

The above message indicates that the watchdog is active for pid 933:

It was inactive for 100000ms:

```
Lustre: 0:0:(linux-debug.c:132:portals_debug_dumpstack())
```

Showing stack for process:

```
933 ll_ost_25     D F896071A     0   933      1    934   932 (L-TLB)
f6d87c60 00000046 00000000 f896071a f8def7cc 00002710 00001822 2da48cae
0008cf1a f6d7c220 f6d7c3d0 f6d86000 f3529648 f6d87cc4 f3529640 f8961d3d
00000010 f6d87c9c ca65a13c 00001fff 00000001 00000001 00000000 00000001
```

Call trace:

```
filter_do_bio+0x3dd/0xb90 [obdfilter]
default_wake_function+0x0/0x20
filter_direct_io+0x2fb/0x990 [obdfilter]
filter_preprw_read+0x5c5/0xe00 [obdfilter]
lustre_swab_niobuf_remote+0x0/0x30 [ptlrpc]
ost_brw_read+0x18df/0x2400 [ost]
ost_handle+0x14c2/0x42d0 [ost]
ptlrpc_server_handle_request+0x870/0x10b0 [ptlrpc]
ptlrpc_main+0x42e/0x7c0 [ptlrpc]
```

27.3.8. Handling Timeouts on Initial Lustre File System Setup

If you come across timeouts or hangs on the initial setup of your Lustre file system, verify that name resolution for servers and clients is working correctly. Some distributions configure `/etc/hosts` so the name of the local machine (as reported by the 'hostname' command) is mapped to local host (127.0.0.1) instead of a proper IP address.

This might produce this error:

```
LustreError:(ldlm_handle_cancel()) received cancel for unknown lock cookie
0xe74021a4b41b954e from nid 0x7f000001 (0:127.0.0.1)
```

27.3.9. Handling/Debugging "LustreError: xxx went back in time"

Each time the Lustre software changes the state of the disk file system, it records a unique transaction number. Occasionally, when committing these transactions to the disk, the last committed transaction

number displays to other nodes in the cluster to assist the recovery. Therefore, the promised transactions remain absolutely safe on the disappeared disk.

This situation arises when:

- You are using a disk device that claims to have data written to disk before it actually does, as in case of a device with a large cache. If that disk device crashes or loses power in a way that causes the loss of the cache, there can be a loss of transactions that you believe are committed. This is a very serious event, and you should run e2fsck against that storage before restarting the Lustre file system.
- As required by the Lustre software, the shared storage used for failover is completely cache-coherent. This ensures that if one server takes over for another, it sees the most up-to-date and accurate copy of the data. In case of the failover of the server, if the shared storage does not provide cache coherency between all of its ports, then the Lustre software can produce an error.

If you know the exact reason for the error, then it is safe to proceed with no further action. If you do not know the reason, then this is a serious issue and you should explore it with your disk vendor.

If the error occurs during failover, examine your disk cache settings. If it occurs after a restart without failover, try to determine how the disk can report that a write succeeded, then lose the Data Device corruption or Disk Errors.

27.3.10. Lustre Error: "Slow Start_Page_Write"

The slow `start_page_write` message appears when the operation takes an extremely long time to allocate a batch of memory pages. Use these pages to receive network traffic first, and then write to disk.

27.3.11. Drawbacks in Doing Multi-client O_APPEND Writes

It is possible to do multi-client `O_APPEND` writes to a single file, but there are few drawbacks that may make this a sub-optimal solution. These drawbacks are:

- Each client needs to take an `EOF` lock on all the OSTs, as it is difficult to know which OST holds the end of the file until you check all the OSTs. As all the clients are using the same `O_APPEND`, there is significant locking overhead.
- The second client cannot get all locks until the end of the writing of the first client, as the taking serializes all writes from the clients.
- To avoid deadlocks, the taking of these locks occurs in a known, consistent order. As a client cannot know which OST holds the next piece of the file until the client has locks on all OSTS, there is a need of these locks in case of a striped file.

27.3.12. Slowdown Occurs During Lustre File System Startup

When a Lustre file system starts, it needs to read in data from the disk. For the very first mdsrate run after the reboot, the MDS needs to wait on all the OSTs for object pre-creation. This causes a slowdown to occur when the file system starts up.

After the file system has been running for some time, it contains more data in cache and hence, the variability caused by reading critical metadata from disk is mostly eliminated. The file system now reads data from the cache.

27.3.13. Log Message 'Out of Memory' on OST

When planning the hardware for an OSS node, consider the memory usage of several components in the Lustre file system. If insufficient memory is available, an 'out of memory' message can be logged.

During normal operation, several conditions indicate insufficient RAM on a server node:

- kernel "`Out of memory`" and/or "`oom-killer`" messages
- Lustre "`kmalloc of 'mmm' (NNNN bytes) failed...`" messages
- Lustre or kernel stack traces showing processes stuck in "`try_to_free_pages`"

For information on determining the MDS memory and OSS memory requirements, see Section 5.4, “Determining Memory Requirements”.

27.3.14. Setting SCSI I/O Sizes

Some SCSI drivers default to a maximum I/O size that is too small for good Lustre file system performance. we have fixed quite a few drivers, but you may still find that some drivers give unsatisfactory performance with the Lustre file system. As the default value is hard-coded, you need to recompile the drivers to change their default. On the other hand, some drivers may have a wrong default set.

If you suspect bad I/O performance and an analysis of Lustre file system statistics indicates that I/O is not 1 MB, check `/sys/block/`*`device`*`/queue/max_sectors_kb`. If the `max_sectors_kb` value is less than 1024, set it to at least 1024 to improve performance. If changing `max_sectors_kb` does not change the I/O size as reported by the Lustre software, you may want to examine the SCSI driver code.

Chapter 28. Troubleshooting Recovery

This chapter describes what to do if something goes wrong during recovery. It describes:

- Section 28.1, “Recovering from Errors or Corruption on a Backing File System”
- Section 28.2, “Recovering from Corruption in the Lustre File System”
- Section 28.3, “Recovering from an Unavailable OST”
- Section 28.4, “Checking the file system with LFSCK”

28.1. Recovering from Errors or Corruption on a Backing File System

When an OSS, MDS, or MGS server crash occurs, it is not necessary to run e2fsck on the file system. `ldiskfs` journaling ensures that the file system remains consistent over a system crash. The backing file systems are never accessed directly from the client, so client crashes are not relevant for server file system consistency.

The only time it is REQUIRED that `e2fsck` be run on a device is when an event causes problems that ldiskfs journaling is unable to handle, such as a hardware device failure or I/O error. If the ldiskfs kernel code detects corruption on the disk, it mounts the file system as read-only to prevent further corruption, but still allows read access to the device. This appears as error "-30" (`EROFS`) in the syslogs on the server, e.g.:

```
Dec 29 14:11:32 mookie kernel: LDISKFS-fs error (device sdz):
            ldiskfs_lookup: unlinked inode 5384166 in dir #145170469
Dec 29 14:11:32 mookie kernel: Remounting filesystem read-only
```

In such a situation, it is normally required that e2fsck only be run on the bad device before placing the device back into service.

In the vast majority of cases, the Lustre software can cope with any inconsistencies found on the disk and between other devices in the file system.

Note

The offline LFSCK tool included with e2fsprogs is rarely required for Lustre file system operation.

For problem analysis, it is strongly recommended that `e2fsck` be run under a logger, like script, to record all of the output and changes that are made to the file system in case this information is needed later.

If time permits, it is also a good idea to first run `e2fsck` in non-fixing mode (-n option) to assess the type and extent of damage to the file system. The drawback is that in this mode, `e2fsck` does not recover the file system journal, so there may appear to be file system corruption when none really exists.

To address concern about whether corruption is real or only due to the journal not being replayed, you can briefly mount and unmount the `ldiskfs` file system directly on the node with the Lustre file system stopped, using a command similar to:

```
mount -t ldiskfs /dev/{ostdev} /mnt/ost; umount /mnt/ost
```

This causes the journal to be recovered.

The `e2fsck` utility works well when fixing file system corruption (better than similar file system recovery tools and a primary reason why `ldiskfs` was chosen over other file systems). However, it is often useful to identify the type of damage that has occurred so an `ldiskfs` expert can make intelligent decisions about what needs fixing, in place of `e2fsck`.

```
root# {stop lustre services for this device, if running}
root# script /tmp/e2fsck.sda
Script started, file is /tmp/e2fsck.sda
root# mount -t ldiskfs /dev/sda /mnt/ost
root# umount /mnt/ost
root# e2fsck -fn /dev/sda   # don't fix file system, just check for corruption
:
[e2fsck output]
:
root# e2fsck -fp /dev/sda   # fix errors with prudent answers (usually yes)
```

28.2. Recovering from Corruption in the Lustre File System

In cases where an ldiskfs MDT or OST becomes corrupt, you need to run e2fsck to correct the local filesystem consistency, then use `LFSCK` to run a distributed check on the file system to resolve any inconsistencies between the MDTs and OSTs.

1. Stop the Lustre file system.
2. Run `e2fsck -f` on the individual MDS / OST that had problems to fix any local file system damage.

   We recommend running `e2fsck` under script, to create a log of changes made to the file system in case it is needed later. After `e2fsck` is run, bring up the file system, if necessary, to reduce the outage window.

28.2.1. Working with Orphaned Objects

The easiest problem to resolve is that of orphaned objects. When the LFSCK layout check is run, these objects are linked to new files and put into `.lustre/lost+found` in the Lustre file system, where they can be examined and saved or deleted as necessary.

28.3. Recovering from an Unavailable OST

One problem encountered in a Lustre file system environment is when an OST becomes unavailable due to a network partition, OSS node crash, etc. When this happens, the OST's clients pause and wait for the OST to become available again, either on the primary OSS or a failover OSS. When the OST comes back online, the Lustre file system starts a recovery process to enable clients to reconnect to the OST. Lustre servers put a limit on the time they will wait in recovery for clients to reconnect.

During recovery, clients reconnect and replay their requests serially, in the same order they were done originally. Until a client receives a confirmation that a given transaction has been written to stable storage,

the client holds on to the transaction, in case it needs to be replayed. Periodically, a progress message prints to the log, stating how_many/expected clients have reconnected. If the recovery is aborted, this log shows how many clients managed to reconnect. When all clients have completed recovery, or if the recovery timeout is reached, the recovery period ends and the OST resumes normal request processing.

If some clients fail to replay their requests during the recovery period, this will not stop the recovery from completing. You may have a situation where the OST recovers, but some clients are not able to participate in recovery (e.g. network problems or client failure), so they are evicted and their requests are not replayed. This would result in any operations on the evicted clients failing, including in-progress writes, which would cause cached writes to be lost. This is a normal outcome; the recovery cannot wait indefinitely, or the file system would be hung any time a client failed. The lost transactions are an unfortunate result of the recovery process.

Note

The failure of client recovery does not indicate or lead to filesystem corruption. This is a normal event that is handled by the MDT and OST, and should not result in any inconsistencies between servers.

Note

The version-based recovery (VBR) feature enables a failed client to be "skipped", so remaining clients can replay their requests, resulting in a more successful recovery from a downed OST. For more information about the VBR feature, see Chapter 31, *Lustre File System Recovery*(Version-based Recovery).

Introduced in Lustre 2.3

LFSCK is an administrative tool introduced in Lustre software release 2.3 for checking and repair of the attributes specific to a mounted Lustre file system. It is similar in concept to an offline fsck repair tool for a local filesystem, but LFSCK is implemented to run as part of the Lustre file system while the file system is mounted and in use. This allows consistency of checking and repair by the Lustre software without unnecessary downtime, and can be run on the largest Lustre file systems.

In Lustre software release 2.3, LFSCK can verify and repair the Object Index (OI) table that is used internally to map Lustre File Identifiers (FIDs) to MDT internal inode numbers, through a process called OI Scrub. An OI Scrub is required after restoring from a file-level MDT backup (Section 17.2, “Backing Up and Restoring an MDS or OST (Device Level)”), or in case the OI table is otherwise corrupted. Later phases of LFSCK will add further checks to the Lustre distributed file system state.

Introduced in Lustre 2.4

In Lustre software release 2.4, LFSCK namespace scanning can verify and repair the directory FID-in-Dirent and LinkEA consistency.

Introduced in Lustre 2.6

In Lustre software release 2.6, LFSCK layout scanning can verify and repair MDT-OST file layout inconsistency. File layout inconsistencies between MDT-objects and OST-objects that are checked and corrected include dangling reference, unreferenced OST-objects, mismatched references and multiple references.

Control and monitoring of LFSCK is through LFSCK and the `/proc` file system interfaces. LFSCK supports three types of interface: switch interface, status interface and adjustment interface. These interfaces are detailed below.

28.4.1. LFSCK switch interface

28.4.1.1. Manually Starting LFSCK

28.4.1.1.1. Synopsis

```
lctl lfsck_start -M | --device [MDT,OST]_device \
                    [-e | --error error_handle] \
                    [-h | --help] \
                    [-n | --dryrun switch] \
                    [-r | --reset] \
                    [-s | --speed speed_limit] \
                    [-A | --all] \
                    [-t | --type lfsck_type[,lfsck_type...]] \
                    [-w | --windows win_size] \
                    [-o | --orphan]
```

28.4.1.1.2. Description

This command is used by LFSCK after the MDT is mounted.

28.4.1.1.3. Options

The various `lfsck_start` options are listed and described below. For a complete list of available options, type `lctl lfsck_start -h`.

Option	Description
`-M | --device`	The MDT or OST device to start LFSCK/scrub on.
`-e | --error`	Error handle, `continue` (default) or `abort`. Specify whether the LFSCK will stop or not if fail to repair something. If it is not specified, the saved value (when resuming from checkpoint) will be used if present. This option cannot be changed if LFSCK is running.
`-h | --help`	Operating help information.
`-n | --dryrun`	Perform a trial without making any changes. `off` (default) or `on`.
`-r | --reset`	Reset the start position for the object iteration to the beginning for the specified MDT. By default the iterator will resume scanning from the last checkpoint (saved periodically by LFSCK) provided it is available.
`-s | --speed`	Set the upper speed limit of LFSCK processing in objects per second. If it is not specified, the saved value (when resuming from checkpoint) or default value of 0 (0 = run as fast as possible) is used. Speed can be adjusted while LFSCK is running with the adjustment interface.
`-A | --all`	Introduced in Lustre 2.6 Start LFSCK on all devices via a single lctl command. It is not only used for layout consistency check/repair, but also for other LFSCK components, such as LFSCK for namespace consistency (LFSCK 1.5) and for DNE consistency check/repair in the future.
`-t | --type`	The type of checking/repairing that should be performed. The new LFSCK framework provides a single interface for a variety of system consistency checking/repairing operations including:

Option	Description
	Without a specified option, the LFSCK component(s) which ran last time and did not finish or the component(s) corresponding to some known system inconsistency, will be started. Anytime the LFSCK is triggered, the OI scrub will run automatically, so there is no need to specify OI_scrub. Introduced in Lustre 2.4 `namespace`: check and repair FID-in-Dirent and LinkEA consistency. Introduced in Lustre 2.6 `layout`: check and repair MDT-OST inconsistency.
`-w | --windows`	Introduced in Lustre 2.6 The windows size for async requests pipeline.
`-o | --orphan`	Introduced in Lustre 2.6 Handle orphan objects, such as orphan OST-objects for layout LFSCK.

28.4.1.2. Manually Stopping LFSCK

28.4.1.2.1. Synopsis

```
lctl lfsck_stop -M | --device [MDT,OST]_device \
                    [-A | --all] \
                    [-h | --help]
```

28.4.1.2.2. Description

This command is used by LFSCK after the MDT is mounted.

28.4.1.2.3. Options

The various `lfsck_stop` options are listed and described below. For a complete list of available options, type `lctl lfsck_stop -h`.

Option	Description
`-M | --device`	The MDT or OST device to stop LFSCK/scrub on.
`-A | --all`	Stop LFSCK on all devices.
`-h | --help`	Operating help information.

28.4.2. LFSCK status interface

28.4.2.1. LFSCK status of OI Scrub via `procfs`

28.4.2.1.1. Synopsis

```
lctl get_param -n osd-ldiskfs.FSNAME-MDT_device.oi_scrub
```

28.4.2.1.2. Description

For each LFSCK component there is a dedicated procfs interface to trace corresponding LFSCK component status. For OI Scrub, the interface is the OSD layer procfs interface, named `oi_scrub`. To display OI Scrub status, the standard `lctl get_param` command is used as described in the synopsis.

28.4.2.1.3. Output

Information	Detail
General Information	• Name: OI_scrub. • OI scrub magic id (an identifier unique to OI scrub). • OI files count. • Status: one of the status - `init`, `scanning`, `completed`, `failed`, `stopped`, `paused`, or `crashed`. • Flags: including - `recreated` (OI file(s) is/are removed/recreated), `inconsistent` (restored from file-level backup), `auto` (triggered by non-UI mechanism), and `upgrade` (from Lustre software release 1.8 IGIF format.) • Parameters: OI scrub parameters, like `failout`. • Time Since Last Completed. • Time Since Latest Start. • Time Since Last Checkpoint. • Latest Start Position: the position for the latest scrub started from. • Last Checkpoint Position. • First Failure Position: the position for the first object to be repaired. • Current Position.
Statistics	• `Checked` total number of objects scanned. • `Updated` total number of objects repaired. • `Failed` total number of objects that failed to be repaired. • `No Scrub` total number of objects marked `LDISKFS_STATE_LUSTRE_NOSCRUB and skipped`. • `IGIF` total number of objects IGIF scanned. • `Prior Updated` how many objects have been repaired which are triggered by parallel RPC. • `Success Count` total number of completed OI_scrub runs on the device.

Information	Detail
	• `Run Time` how long the scrub has run, tally from the time of scanning from the beginning of the specified MDT device, not include the paused/failure time among checkpoints. • `Average Speed` calculated by dividing `Checked` by `run_time`. • `Real-Time Speed` the speed since last checkpoint if the OI_scrub is running. • `Scanned` total number of objects under /lost+found that have been scanned. • `Repaired` total number of objects under /lost+found that have been recovered. • `Failed` total number of objects under /lost+found failed to be scanned or failed to be recovered.

Introduced in Lustre 2.4

28.4.2.2.1. Synopsis

```
lctl get_param -n mdd.FSNAME-MDT_device.lfsck_namespace
```

28.4.2.2.2. Description

The `namespace` component is responsible for checking and repairing FID-in-Dirent and LinkEA consistency. The `procfs` interface for this component is in the MDD layer, named `lfsck_namespace`. To show the status of this component `lctl get_param` should be used as described in the synopsis.

28.4.2.2.3. Output

Information	Detail
General Information	• Name: `lfsck_namespace` • LFSCK namespace magic. • LFSCK namespace version.. • Status: one of the status - `init`, `scanning-phase1`, `scanning-phase2`, `completed`, `failed`, `stopped`, `paused`, or `crashed`. • Flags: including - `scanned-once` (the first cycle scanning has been completed), `inconsistent` (one or more inconsistent FID-in-Dirent or LinkEA entries have been discovered), `upgrade` (from Lustre software release 1.8 IGIF format.) • Parameters: including `dryrun`, `all_targets` and `failout`. • Time Since Last Completed. • Time Since Latest Start. • Time Since Last Checkpoint.

Information	Detail
	• Latest Start Position: the position the checking began most recently. • Last Checkpoint Position. • First Failure Position: the position for the first object to be repaired. • Current Position.
Statistics	• `Checked Phase1` total number of objects scanned during `scanning-phase1`. • `Checked Phase2` total number of objects scanned during `scanning-phase2`. • `Updated Phase1` total number of objects repaired during `scanning-phase1`. • `Updated Phase2` total number of objects repaired during `scanning-phase2`. • `Failed Phase1` total number of objets that failed to be repaired during `scanning-phase1`. • `Failed Phase2` total number of objets that failed to be repaired during `scanning-phase2`. • `Dirs` total number of directories scanned. • `M-linked` total number of multiple-linked objects that have been scanned. • `Nlinks Repaired` total number of objects with nlink attributes that have been repaired. • `Lost_found` total number of objects that have had a name entry added back to the namespace. • `Success Count` the total number of completed LFSCK runs on the device. • `Run Time Phase1` the duration of the LFSCK run during `scanning-phase1`. Excluding the time spent paused between checkpoints. • `Run Time Phase2` the duration of the LFSCK run during `scanning-phase2`. Excluding the time spent paused between checkpoints. • `Average Speed Phase1` calculated by dividing `checked_phase1` by `run_time_phase1`. • `Average Speed Phase2` calculated by dividing `checked_phase2` by `run_time_phase1`. • `Real-Time Speed Phase1` the speed since the last checkpoint if the LFSCK is running `scanning-phase1`.

Information	Detail
	• `Real-Time Speed Phase2` the speed since the last checkpoint if the LFSCK is running `scanning-phase2`.

Introduced in Lustre 2.6

28.4.2.3.1. Synopsis

```
lctl get_param -n mdd.FSNAME-MDT_device.lfsck_layout
lctl get_param -n obdfilter.FSNAME-OST_device.lfsck_layout
```

28.4.2.3.2. Description

The `layout` component is responsible for checking and repairing MDT-OST inconsistency. The `procfs` interface for this component is in the MDD layer, named `lfsck_layout`, and in the OBD layer, named `lfsck_layout`. To show the status of this component `lctl get_param` should be used as described in the synopsis.

28.4.2.3.3. Output

Information	Detail
General Information	• Name: `lfsck_layout` • LFSCK namespace magic. • LFSCK namespace version.. • Status: one of the status - `init`, `scanning-phase1`, `scanning-phase2`, `completed`, `failed`, `stopped`, `paused`, `crashed`, `partial`, `co-failed`, `co-stopped`, or `co-paused`. • Flags: including - `scanned-once` (the first cycle scanning has been completed), `inconsistent` (one or more MDT-OST inconsistencies have been discovered), `incomplete` (some MDT or OST did not participate in the LFSCK or failed to finish the LFSCK) or `crashed_lastid` (the lastid files on the OST crashed and needs to be rebuilt). • Parameters: including `dryrun`, `all_targets` and `failout`. • Time Since Last Completed. • Time Since Latest Start. • Time Since Last Checkpoint. • Latest Start Position: the position the checking began most recently. • Last Checkpoint Position. • First Failure Position: the position for the first object to be repaired. • Current Position.

Information	Detail
Statistics	• `Success Count:` the total number of completed LFSCK runs on the device. • `Repaired Dangling:` total number of MDT-objects with dangling reference have been repaired in the scanning-phase1. • `Repaired Unmatched Pairs` total number of unmatched MDT and OST-object paris have been repaired in the scanning-phase1 • `Repaired Multiple Referenced` total number of OST-objects with multiple reference have been repaired in the scanning-phase1. • `Repaired Orphan` total number of orphan OST-objects have been repaired in the scanning-phase2. • `Repaired Inconsistent Owner` total number.of OST-objects with incorrect owner information have been repaired in the scanning-phase1. • `Repaired Others` total number of.other inconsistency repaired in the scanning phases. • `Skipped` Number of skipped objects. • `Failed Phase1` total number of objects that failed to be repaired during `scanning-phase1`. • `Failed Phase2` total number of objects that failed to be repaired during `scanning-phase2`. • `Checked Phase1` total number of objects scanned during `scanning-phase1`. • `Checked Phase2` total number of objects scanned during `scanning-phase2`. • `Run Time Phase1` the duration of the LFSCK run during `scanning-phase1`. Excluding the time spent paused between checkpoints. • `Run Time Phase2` the duration of the LFSCK run during `scanning-phase2`. Excluding the time spent paused between checkpoints. • `Average Speed Phase1` calculated by dividing `checked_phase1` by `run_time_phase1`. • `Average Speed Phase2` calculated by dividing `checked_phase2` by `run_time_phase1`. • `Real-Time Speed Phase1` the speed since the last checkpoint if the LFSCK is running `scanning-phase1`.

Information	Detail
	• `Real-Time Speed Phase2` the speed since the last checkpoint if the LFSCK is running `scanning-phase2`.

28.4.3. LFSCK adjustment interface

Introduced in Lustre 2.6

28.4.3.1.1. Synopsis

```
lctl set_param mdd.${FSNAME}-${MDT_device}.lfsck_speed_limit=N
lctl set_param obdfilter.${FSNAME}-${OST_device}.lfsck_speed_limit=N
```

28.4.3.1.2. Description

Change the LFSCK upper speed limit.

28.4.3.1.3. Values

0	No speed limit (run at maximum speed.)
positive integer	Maximum number of objects to scan per second.

28.4.3.2. Mount options

28.4.3.2.1. Synopsis

```
lctl set_param osd_ldiskfs.${FSNAME}-${MDT_device}.auto_scrub=N
```

28.4.3.2.2. Description

Typically, the MDT will detect restoration from a file-level backup during mount. For convenience an mount option `noscrub` is provided for MDTs. `noscrub` prevents the OI Scrub starting automatically when the MDT is mounted. The administrator can start LFSCK manually after the MDT is mounted with `lctl`. Manually starting LFSCK after mounting provides finer control over the starting conditions.

28.4.3.2.3. Values

0	Do not start OI Scrub automatically.
positive integer	Manually start OI Scrub if needed.

Chapter 29. Debugging a Lustre File System

This chapter describes tips and information to debug a Lustre file system, and includes the following sections:

- Section 29.1, " Diagnostic and Debugging Tools"
- Section 29.2, "Lustre Debugging Procedures"
- Section 29.3, "Lustre Debugging for Developers"

29.1. Diagnostic and Debugging Tools

A variety of diagnostic and analysis tools are available to debug issues with the Lustre software. Some of these are provided in Linux distributions, while others have been developed and are made available by the Lustre project.

29.1.1. Lustre Debugging Tools

The following in-kernel debug mechanisms are incorporated into the Lustre software:

- **Debug logs** - A circular debug buffer to which Lustre internal debug messages are written (in contrast to error messages, which are printed to the syslog or console). Entries to the Lustre debug log are controlled by the mask set by `/proc/sys/lnet/debug`. The log size defaults to 5 MB per CPU but can be increased as a busy system will quickly overwrite 5 MB. When the buffer fills, the oldest information is discarded.
- **Debug daemon** - The debug daemon controls logging of debug messages.
- **`/proc/sys/lnet/debug`** - This file contains a mask that can be used to delimit the debugging information written out to the kernel debug logs.

The following tools are also provided with the Lustre software:

- **`lctl`** - This tool is used with the debug_kernel option to manually dump the Lustre debugging log or post-process debugging logs that are dumped automatically. For more information about the lctl tool, see Section 29.2.2, "Using the lctl Tool to View Debug Messages" and Section 37.3, " lctl".
- Lustre subsystem asserts - A panic-style assertion (LBUG) in the kernel causes the Lustre file system to dump the debug log to the file `/tmp/lustre-log.`*`timestamp`* where it can be retrieved after a reboot. For more information, see Section 27.1.2, "Viewing Error Messages".
- *`lfs`* - This utility provides access to the extended attributes (EAs) of a Lustre file (along with other information). For more information about lfs, see Section 33.1, " `lfs` ".

29.1.2. External Debugging Tools

The tools described in this section are provided in the Linux kernel or are available at an external website. For information about using some of these tools for Lustre debugging, see Section 29.2, "Lustre Debugging Procedures" and Section 29.3, "Lustre Debugging for Developers".

29.1.2.1. Tools for Administrators and Developers

Some general debugging tools provided as a part of the standard Linux distribution are:

- **`strace`** . This tool allows a system call to be traced.
- **`/var/log/messages`** . `syslogd` prints fatal or serious messages at this log.
- **Crash dumps** . On crash-dump enabled kernels, sysrq c produces a crash dump. The Lustre software enhances this crash dump with a log dump (the last 64 KB of the log) to the console.
- **`debugfs`** . Interactive file system debugger.

The following logging and data collection tools can be used to collect information for debugging Lustre kernel issues:

- **`kdump`** . A Linux kernel crash utility useful for debugging a system running Red Hat Enterprise Linux. For more information about `kdump`, see the Red Hat knowledge base article How do I configure kexec/kdump on Red Hat Enterprise Linux 5? [http://kbase.redhat.com/faq/docs/DOC-6039]. To download `kdump`, go to the Fedora Project Download [http://fedoraproject.org/wiki/SystemConfig/kdump#Download] site.
- **`netconsole`** . Enables kernel-level network logging over UDP. A system requires (SysRq) allows users to collect relevant data through `netconsole`.
- **`netdump`** . A crash dump utility from Red Hat that allows memory images to be dumped over a network to a central server for analysis. The `netdump` utility was replaced by `kdump` in Red Hat Enterprise Linux 5. For more information about `netdump`, see Red Hat, Inc.'s Network Console and Crash Dump Facility [http://www.redhat.com/support/wpapers/redhat/netdump/].
- *`wireshark`* . A network packet inspection tool that allows debugging of information that was sent between the various Lustre nodes. This tool is built on top of `tcpdump` and can read packet dumps generated by it. There are plug-ins available to dissassemble the LNET and Lustre protocols. They are located within the Lustre git repository [http://git.hpdd.intel.com/] under `lustre/contrib/wireshark/`. Installation instruction are included in that directory. See also Wireshark Website [http://www.wireshark.org/] for more details.

29.1.2.2. Tools for Developers

The tools described in this section may be useful for debugging a Lustre file system in a development environment.

Of general interest is:

- *`leak_finder.pl`* . This program provided with the Lustre software is useful for finding memory leaks in the code.

A virtual machine is often used to create an isolated development and test environment. Some commonly-used virtual machines are:

- **VirtualBox Open Source Edition** . Provides enterprise-class virtualization capability for all major platforms and is available free at Get Sun VirtualBox [http://www.sun.com/software/products/virtualbox/get.jsp?intcmp=2945].
- **VMware Server** . Virtualization platform available as free introductory software at Download VMware Server [http://downloads.vmware.com/d/info/datacenter_downloads/vmware_server/2_0].

- **Xen** . A para-virtualized environment with virtualization capabilities similar to VMware Server and Virtual Box. However, Xen allows the use of modified kernels to provide near-native performance and the ability to emulate shared storage. For more information, go to xen.org [http://xen.org/].

A variety of debuggers and analysis tools are available including:

- **kgdb** . The Linux Kernel Source Level Debugger kgdb is used in conjunction with the GNU Debugger `gdb` for debugging the Linux kernel. For more information about using `kgdb` with `gdb`, see Chapter 6. Running Programs Under gdb [http://www.linuxtopia.org/online_books/redhat_linux_debugging_with_gdb/running.html] in the *Red Hat Linux 4 Debugging with GDB* guide.

- **crash** . Used to analyze saved crash dump data when a system had panicked or locked up or appears unresponsive. For more information about using crash to analyze a crash dump, see:

  - Red Hat Magazine article: A quick overview of Linux kernel crash dump analysis [http://magazine.redhat.com/2007/08/15/a-quick-overview-of-linux-kernel-crash-dump-analysis/]

  - Crash Usage: A Case Study [http://people.redhat.com/anderson/crash_whitepaper/#EXAMPLES] from the white paper *Red Hat Crash Utility* by David Anderson

  - Kernel Trap forum entry: Linux: Kernel Crash Dumps [http://kerneltrap.org/node/5758]

  - White paper: A Quick Overview of Linux Kernel Crash Dump Analysis [http://www.google.com/url?sa=t&source=web&ct=res&cd=8&ved=0CCUQFjAH&url=http%3A%2F%2Fwww.kernel.sg%2Fpapers%2Fcrash-dump-analysis.pdf&rct=j&q=redhat+crash+dump&ei=6aQBS-ifK4T8tAPcjdiHCw&usg=AFQjCNEk03E3GDtAsawG3gfpwc1gGNELAg]

29.2. Lustre Debugging Procedures

The procedures below may be useful to administrators or developers debugging a Lustre files system.

29.2.1. Understanding the Lustre Debug Messaging Format

Lustre debug messages are categorized by originating subsystem, message type, and location in the source code. For a list of subsystems and message types, see Section 29.2.1.1, “Lustre Debug Messages”.

Note

For a current list of subsystems and debug message types, see `libcfs/include/libcfs/libcfs_debug.h` in the Lustre software tree

The elements of a Lustre debug message are described in Section 29.2.1.2, “Format of Lustre Debug Messages” Format of Lustre Debug Messages.

29.2.1.1. Lustre Debug Messages

Each Lustre debug message has the tag of the subsystem it originated in, the message type, and the location in the source code. The subsystems and debug types used are as follows:

- Standard Subsystems:

mdc, mds, osc, ost, obdclass, obdfilter, llite, ptlrpc, portals, lnd, ldlm, lov

- Debug Types:

Types	Description
trace	Entry/Exit markers
dlmtrace	Locking-related information
inode	
super	
ext2	Anything from the ext2_debug
malloc	Print malloc or free information
cache	Cache-related information
info	General information
ioctl	IOCTL-related information
blocks	Ext2 block allocation information
net	Networking
warning	
buffs	
other	
dentry	
portals	Entry/Exit markers
page	Bulk page handling
error	Error messages
emerg	
rpctrace	For distributed debugging
ha	Failover and recovery-related information

29.2.1.2. Format of Lustre Debug Messages

The Lustre software uses the `CDEBUG()` and `CERROR()` macros to print the debug or error messages. To print the message, the `CDEBUG()` macro uses the function `libcfs_debug_msg()` (`libcfs/libcfs/tracefile.c`). The message format is described below, along with an example.

Description	Parameter
subsystem	800000
debug mask	000010
smp_processor_id	0
seconds.microseconds	1081880847.677302
stack size	1204
pid	2973
host pid (UML only) or zero	31070

Description	Parameter
(file:line #:function_name())	(obd_mount.c:2089:lustre_fill_super())
debug message	kmalloced '*obj': 24 at a375571c (tot 17447717)

29.2.1.3. Lustre Debug Messages Buffer

Lustre debug messages are maintained in a buffer, with the maximum buffer size specified (in MBs) by the `debug_mb` parameter (`lctl get_param debug_mb`). The buffer is circular, so debug messages are kept until the allocated buffer limit is reached, and then the first messages are overwritten.

29.2.2. Using the lctl Tool to View Debug Messages

The `lctl` tool allows debug messages to be filtered based on subsystems and message types to extract information useful for troubleshooting from a kernel debug log. For a command reference, see Section 37.3, “ lctl”.

You can use `lctl` to:

- Obtain a list of all the types and subsystems:

  ```
  lctl > debug_list subsystems|types
  ```

- Filter the debug log:

  ```
  lctl > filter subsystem_name|debug_type
  ```

  Note

  When `lctl` filters, it removes unwanted lines from the displayed output. This does not affect the contents of the debug log in the kernel's memory. As a result, you can print the log many times with different filtering levels without worrying about losing data.

- Show debug messages belonging to certain subsystem or type:

  ```
  lctl > show subsystem_name|debug_type
  ```

  `debug_kernel` pulls the data from the kernel logs, filters it appropriately, and displays or saves it as per the specified options

  ```
  lctl > debug_kernel [output filename]
  ```

  If the debugging is being done on User Mode Linux (UML), it might be useful to save the logs on the host machine so that they can be used at a later time.

- Filter a log on disk, if you already have a debug log saved to disk (likely from a crash):

  ```
  lctl > debug_file input_file [output_file]
  ```

  During the debug session, you can add markers or breaks to the log for any reason:

  ```
  lctl > mark [marker text]
  ```

  The marker text defaults to the current date and time in the debug log (similar to the example shown below):

```
DEBUG MARKER: Tue Mar 5 16:06:44 EST 2002
```

- Completely flush the kernel debug buffer:

```
lctl > clear
```

Note

Debug messages displayed with `lctl` are also subject to the kernel debug masks; the filters are additive.

29.2.2.1. Sample `lctl` Run

Below is a sample run using the `lctl` command.

```
bash-2.04# ./lctl
lctl > debug_kernel /tmp/lustre_logs/log_all
Debug log: 324 lines, 324 kept, 0 dropped.
lctl > filter trace
Disabling output of type "trace"
lctl > debug_kernel /tmp/lustre_logs/log_notrace
Debug log: 324 lines, 282 kept, 42 dropped.
lctl > show trace
Enabling output of type "trace"
lctl > filter portals
Disabling output from subsystem "portals"
lctl > debug_kernel /tmp/lustre_logs/log_noportals
Debug log: 324 lines, 258 kept, 66 dropped.
```

29.2.3. Dumping the Buffer to a File (debug_daemon)

The `lctl debug_daemon` command is used to continuously dump the `debug_kernel` buffer to a user-specified file. This functionality uses a kernel thread to continuously dump the messages from the kernel debug log, so that much larger debug logs can be saved over a longer time than would fit in the kernel ringbuffer.

The `debug_daemon` is highly dependent on file system write speed. File system write operations may not be fast enough to flush out all of the `debug_buffer` if the Lustre file system is under heavy system load and continues to log debug messages to the `debug_buffer`. The `debug_daemon` will write the message `DEBUG MARKER: Trace buffer full` into the `debug_buffer` to indicate the `debug_buffer` contents are overlapping before the `debug_daemon` flushes data to a file.

Users can use the `lctl debug_daemon` command to start or stop the Lustre daemon from dumping the `debug_buffer` to a file.

29.2.3.1. `lctl debug_daemon` Commands

To initiate the `debug_daemon` to start dumping the `debug_buffer` into a file, run as the root user:

```
lctl debug_daemon start filename [megabytes]
```

The debug log will be written to the specified filename from the kernel. The file will be limited to the optionally specified number of megabytes.

The daemon wraps around and dumps data to the beginning of the file when the output file size is over the limit of the user-specified file size. To decode the dumped file to ASCII and sort the log entries by time, run:

```
lctl debug_file filename > newfile
```

The output is internally sorted by the `lctl` command.

To stop the `debug_daemon` operation and flush the file output, run:

```
lctl debug_daemon stop
```

Otherwise, `debug_daemon` is shut down as part of the Lustre file system shutdown process. Users can restart `debug_daemon` by using start command after each stop command issued.

This is an example using `debug_daemon` with the interactive mode of `lctl` to dump debug logs to a 40 MB file.

```
lctl

lctl > debug_daemon start /var/log/lustre.40.bin 40

run filesystem operations to debug

lctl > debug_daemon stop

lctl > debug_file /var/log/lustre.bin /var/log/lustre.log
```

To start another daemon with an unlimited file size, run:

```
lctl > debug_daemon start /var/log/lustre.bin
```

The text message `*** End of debug_daemon trace log ***` appears at the end of each output file.

29.2.4. Controlling Information Written to the Kernel Debug Log

The `lctl set_param subsystem_debug=subsystem_mask` and `lctl set_param debug=debug_mask` are used to determine which information is written to the debug log. The subsystem_debug mask determines the information written to the log based on the functional area of the code (such as lnet, osc, or ldlm). The debug mask controls information based on the message type (such as info, error, trace, or malloc). For a complete list of possible debug masks use the `lctl debug_list types` command.

To turn off Lustre debugging completely:

```
lctl set_param debug=0
```

To turn on full Lustre debugging:

```
lctl set_param debug=-1
```

To list all possible debug masks:

```
lctl debug_list types
```

To log only messages related to network communications:

```
lctl set_param debug=net
```

To turn on logging of messages related to network communications and existing debug flags:

```
lctl set_param debug=+net
```

To turn off network logging with changing existing flags:

```
lctl set_param debug=-net
```

The various options available to print to kernel debug logs are listed in `libcfs/include/libcfs/libcfs.h`

29.2.5. Troubleshooting with `strace`

The `strace` utility provided with the Linux distribution enables system calls to be traced by intercepting all the system calls made by a process and recording the system call name, arguments, and return values.

To invoke `strace` on a program, enter:

```
$ strace program [arguments]
```

Sometimes, a system call may fork child processes. In this situation, use the `-f` option of `strace` to trace the child processes:

```
$ strace -f program [arguments]
```

To redirect the `strace` output to a file, enter:

```
$ strace -o filename program [arguments]
```

Use the `-ff` option, along with `-o`, to save the trace output in `filename.pid`, where `pid` is the process ID of the process being traced. Use the `-ttt` option to timestamp all lines in the strace output, so they can be correlated to operations in the lustre kernel debug log.

29.2.6. Looking at Disk Content

In a Lustre file system, the inodes on the metadata server contain extended attributes (EAs) that store information about file striping. EAs contain a list of all object IDs and their locations (that is, the OST that stores them). The `lfs` tool can be used to obtain this information for a given file using the `getstripe` subcommand. Use a corresponding `lfs setstripe` command to specify striping attributes for a new file or directory.

The `lfs getstripe` command takes a Lustre filename as input and lists all the objects that form a part of this file. To obtain this information for the file `/mnt/lustre/frog` in a Lustre file system, run:

```
$ lfs getstripe /mnt/lustre/frog
lmm_stripe_count:   2
lmm_stripe_size:    1048576
lmm_stripe_offset:  2
        obdidx           objid          objid           group
```

```
2          818855        0xc7ea7              0
0          873123        0xd52a3              0
```

The `debugfs` tool is provided in the `e2fsprogs` package. It can be used for interactive debugging of an `ldiskfs` file system. The `debugfs` tool can either be used to check status or modify information in the file system. In a Lustre file system, all objects that belong to a file are stored in an underlying `ldiskfs` file system on the OSTs. The file system uses the object IDs as the file names. Once the object IDs are known, use the `debugfs` tool to obtain the attributes of all objects from different OSTs.

A sample run for the `/mnt/lustre/frog` file used in the above example is shown here:

```
$ debugfs -c -R "stat O/0/d$((818855 % 32))/818855" /dev/vgmyth/lvmythost2

debugfs 1.41.90.wc3 (28-May-2011)
/dev/vgmyth/lvmythost2: catastrophic mode - not reading inode or group bitmaps
Inode: 227649   Type: regular    Mode:  0666   Flags: 0x80000
Generation: 1375019198    Version: 0x0000002f:0000728f
User:  1000   Group:  1000   Size: 2800
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x4e177fe5:00000000 -- Fri Jul  8 16:08:37 2011
 atime: 0x4d2e2397:00000000 -- Wed Jan 12 14:56:39 2011
 mtime: 0x4e177fe5:00000000 -- Fri Jul  8 16:08:37 2011
crtime: 0x4c3b5820:a364117c -- Mon Jul 12 12:00:00 2010
Size of extra inode fields: 28
Extended attributes stored in inode body:
  fid = "08 80 24 00 00 00 00 00 28 8a e7 fc 00 00 00 00 a7 7e 0c 00 00 00 00 00
 00 00 00 00 00 00 00 00 " (32)
  fid: objid=818855 seq=0 parent=[0x248008:0xfce78a28:0x0] stripe=0
EXTENTS:
(0):63331288
```

29.2.7. Finding the Lustre UUID of an OST

To determine the Lustre UUID of an OST disk (for example, if you mix up the cables on your OST devices or the SCSI bus numbering suddenly changes and the SCSI devices get new names), it is possible to extract this from the last_rcvd file using debugfs:

```
debugfs -c -R "dump last_rcvd /tmp/last_rcvd" /dev/sdc
strings /tmp/last_rcvd | head -1
myth-OST0004_UUID
```

It is also possible (and easier) to extract this from the file system label using the `dumpe2fs` command:

```
dumpe2fs -h /dev/sdc | grep volume
dumpe2fs 1.41.90.wc3 (28-May-2011)
Filesystem volume name:   myth-OST0004
```

The debugfs and dumpe2fs commands are well documented in the `debugfs(8)` and `dumpe2fs(8)` manual pages.

29.2.8. Printing Debug Messages to the Console

To dump debug messages to the console (`/var/log/messages`), set the corresponding debug mask in the `printk` flag:

```
lctl set_param printk=-1
```

This slows down the system dramatically. It is also possible to selectively enable or disable this capability for particular flags using:`lctl set_param printk=+vfstrace` and `lctl set_param printk=-vfstrace` .

It is possible to disable warning, error, and console messages, though it is strongly recommended to have something like `lctl debug_daemon` running to capture this data to a local file system for failure detection purposes.

29.2.9. Tracing Lock Traffic

The Lustre software provides a specific debug type category for tracing lock traffic. Use:

```
lctl> filter all_types
lctl> show dlmtrace
lctl> debug_kernel [filename]
```

29.2.10. Controlling Console Message Rate Limiting

Some console messages which are printed by Lustre are rate limited. When such messages are printed, they may be followed by a message saying "Skipped N previous similar message(s)," where N is the number of messages skipped. This rate limiting can be completely disabled by a libcfs module parameter called `libcfs_console_ratelimit`. To disable console message rate limiting, add this line to `/etc/modprobe.d/lustre.conf` and then reload Lustre modules.

```
options libcfs libcfs_console_ratelimit=0
```

It is also possible to set the minimum and maximum delays between rate-limited console messages using the module parameters `libcfs_console_max_delay` and `libcfs_console_min_delay`. Set these in `/etc/modprobe.d/lustre.conf` and then reload Lustre modules. Additional information on libcfs module parameters is available via `modinfo`:

```
modinfo libcfs
```

29.3. Lustre Debugging for Developers

The procedures in this section may be useful to developers debugging Lustre source code.

29.3.1. Adding Debugging to the Lustre Source Code

The debugging infrastructure provides a number of macros that can be used in Lustre source code to aid in debugging or reporting serious errors.

To use these macros, you will need to set the `DEBUG_SUBSYSTEM` variable at the top of the file as shown below:

```
#define DEBUG_SUBSYSTEM S_PORTALS
```

A list of available macros with descriptions is provided in the table below.

Macro	Description
LBUG()	A panic-style assertion in the kernel which causes the Lustre file system to dump its circular log to the `/tmp/lustre-log` file. This file can be retrieved after a reboot. `LBUG()` freezes the thread to allow capture of the panic stack. A system reboot is needed to clear the thread.
LASSERT()	Validates a given expression as true, otherwise calls LBUG(). The failed expression is printed on the console, although the values that make up the expression are not printed.
LASSERTF()	Similar to `LASSERT()` but allows a free-format message to be printed, like `printf/printk`.
CDEBUG()	The basic, most commonly used debug macro that takes just one more argument than standard `printf()` - the debug type. This message adds to the debug log with the debug mask set accordingly. Later, when a user retrieves the log for troubleshooting, they can filter based on this type. `CDEBUG(D_INFO, "debug message: rc= %d\n", number);`
CDEBUG_LIMIT()	Behaves similarly to `CDEBUG()`, but rate limits this message when printing to the console (for `D_WARN`, `D_ERROR`, and `D_CONSOLE` message types. This is useful for messages that use a variable debug mask: `CDEBUG(mask, "maybe bad: rc=%d\n", rc);`
CERROR()	Internally using `CDEBUG_LIMIT(D_ERROR, ...)`, which unconditionally prints the message in the debug log and to the console. This is appropriate for serious errors or fatal conditions. Messages printed to the console are prefixed with `LustreError:`, and are rate-limited, to avoid flooding the console with duplicates. `CERROR("Something bad happened: rc= %d\n", rc);`
CWARN()	Behaves similarly to `CERROR()`, but prefixes the messages with `Lustre:`. This is appropriate for important, but not fatal conditions. Messages printed to the console are rate-limited.
CNETERR()	Behaves similarly to `CERROR()`, but prints error messages for LNET if `D_NETERR` is set in the `debug` mask. This is appropriate for serious networking errors. Messages printed to the console are rate-limited.

Macro	Description
DEBUG_REQ()	Prints information about the given ptlrpc_request structure. DEBUG_REQ(D_RPCTRACE, req, ""Handled RPC: rc=%d\n", rc);
ENTRY	Add messages to the entry of a function to aid in call tracing (takes no arguments). When using these macros, cover all exit conditions with a single EXIT, GOTO(), or RETURN() macro to avoid confusion when the debug log reports that a function was entered, but never exited.
EXIT	Mark the exit of a function, to match ENTRY (takes no arguments).
GOTO()	Mark when code jumps via goto to the end of a function, to match ENTRY, and prints out the goto label and function return code in signed and unsigned decimal, and hexadecimal format.
RETURN()	Mark the exit of a function, to match ENTRY, and prints out the function return code in signed and unsigned decimal, and hexadecimal format.
LDLM_DEBUG() **LDLM_DEBUG_NOLOCK()**	Used when tracing LDLM locking operations. These macros build a thin trace that shows the locking requests on a node, and can also be linked across the client and server node using the printed lock handles.
OBD_FAIL_CHECK()	Allows insertion of failure points into the Lustre source code. This is useful to generate regression tests that can hit a very specific sequence of events. This works in conjunction with "lctl set_param fail_loc=*fail_loc*" to set a specific failure point for which a given OBD_FAIL_CHECK() will test.
OBD_FAIL_TIMEOUT()	Similar to OBD_FAIL_CHECK(). Useful to simulate hung, blocked or busy processes or network devices. If the given fail_loc is hit, OBD_FAIL_TIMEOUT() waits for the specified number of seconds.
OBD_RACE()	Similar to OBD_FAIL_CHECK(). Useful to have multiple processes execute the same code concurrently to provoke locking races. The first process to hit OBD_RACE() sleeps until a second process hits OBD_RACE(), then both processes continue.
OBD_FAIL_ONCE	A flag set on a fail_loc breakpoint to cause the OBD_FAIL_CHECK() condition to be hit only one time. Otherwise, a fail_loc is permanent until it is cleared with "lctl set_param fail_loc=0".

Macro	Description
OBD_FAIL_RAND	A flag set on a fail_loc breakpoint to cause OBD_FAIL_CHECK() to fail randomly; on average every (1 / fail_val) times.
OBD_FAIL_SKIP	A flag set on a fail_loc breakpoint to cause OBD_FAIL_CHECK() to succeed fail_val times, and then fail permanently or once with OBD_FAIL_ONCE.
OBD_FAIL_SOME	A flag set on fail_loc breakpoint to cause OBD_FAIL_CHECK to fail fail_val times, and then succeed.

29.3.2. Accessing the ptlrpc Request History

Each service maintains a request history, which can be useful for first occurrence troubleshooting.

ptlrpc is an RPC protocol layered on LNET that deals with stateful servers and has semantics and built-in support for recovery.

The ptlrpc request history works as follows:

1. request_in_callback() adds the new request to the service's request history.
2. When a request buffer becomes idle, it is added to the service's request buffer history list.
3. Buffers are culled from the service request buffer history if it has grown above req_buffer_history_max and its reqs are removed from the service request history.

Request history is accessed and controlled using the following /proc files under the service directory:

- req_buffer_history_len

  Number of request buffers currently in the history

- req_buffer_history_max

  Maximum number of request buffers to keep

- req_history

  The request history

Requests in the history include "live" requests that are currently being handled. Each line in req_history looks like:

sequence:*target_NID*:*client_NID*:*cliet_xid*:*request_length*:*rpc_phase service_specific*

Parameter	Description
seq	Request sequence number
target NID	Destination NID of the incoming request
client ID	Client PID and NID
xid	rq_xid
length	Size of the request message

Parameter	Description
phase	• New (waiting to be handled or could not be unpacked) • Interpret (unpacked or being handled) • Complete (handled)
svc specific	Service-specific request printout. Currently, the only service that does this is the OST (which prints the opcode if the message has been unpacked successfully

29.3.3. Finding Memory Leaks Using `leak_finder.pl`

Memory leaks can occur in code when memory has been allocated and then not freed once it is no longer required. The `leak_finder.pl` program provides a way to find memory leaks.

Before running this program, you must turn on debugging to collect all `malloc` and free entries. Run:

```
lctl set_param debug=+malloc
```

Then complete the following steps:

1. Dump the log into a user-specified log file using lctl (see Section 29.2.2, “Using the lctl Tool to View Debug Messages”).

2. Run the leak finder on the newly-created log dump:

   ```
   perl leak_finder.pl ascii-logname
   ```

The output is:

```
malloced 8bytes at a3116744 (called pathcopy)
(lprocfs_status.c:lprocfs_add_vars:80)
freed 8bytes at a3116744 (called pathcopy)
(lprocfs_status.c:lprocfs_add_vars:80)
```

The tool displays the following output to show the leaks found:

```
Leak:32bytes allocated at a23a8fc(service.c:ptlrpc_init_svc:144,debug file line 24
```

Part VI. Reference

Part VI includes reference information about Lustre file system user utilities, configuration files and module parameters, programming interfaces, system configuration utilities, and system limits. You will find information in this section about:

- Installing a Lustre File System from Source Code
- Lustre File System Recovery
- LustreProc
- User Utilities
- Programming Interfaces
- Setting Lustre Properties in a C Program (`llapi`)
- Configuration Files and Module Parameters
- System Configuration Utilities

Chapter 30. Installing a Lustre File System from Source Code

This chapter describes how to create a customized Lustre server kernel from source code. Sections included are:

- Section 30.1, “Overview and Prerequisites”
- Section 30.2, “Patching the Kernel”
- Section 30.3, “ Building the Lustre RPMs”
- Section 30.4, “Installing and Testing a Lustre File System”

Note

It is recommended that you install from prebuild RPMs of the Lustre software unless you need to customize the Lustre server kernel or will be using an Linux kernel that has not been tested with the Lustre software. Instructions on installing pre-built RPMs are in Chapter 8, *Installing the Lustre Software*

For a list of supported Linux distributions and architectures, see Table 8.1, “Lustre Test Matrix”. Prebuild RPMs are available in the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository. For information about installing Lustre RPMs, see Chapter 8, *Installing the Lustre Software*.

30.1. Overview and Prerequisites

To install the Lustre software from source code, the following are required:

- A x86_64 machine with a fresh installation of a Linux operating system.

  ### Note

  This section of the Manual only describes the specific procedure against building on Red Hat Enterprise Linux 6 and derivatives. For a complete set of supported platforms please visit Table 8.1, “Lustre Test Matrix”.The server kernel should be one of the supported kernel versions but it is not necessary to run a vendor distribution to build the Lustre software.

- Access to the Lustre software git repository at http://git.hpdd.intel.com/.
- (Recommended) Access to a recent version of EPEL [http://fedoraproject.org/wiki/EPEL] containing the `quilt` utility used for managing a series of patches.

  ### Note

  The use of `quilt` is optional for building the Lustre software, and is detected at configure time if it is installed. Otherwise the `patch` utility is used to apply `ldiskfs` patches to the kernel source. Quilt is useful if you intend to modify the Lustre kernel patches.

- *(Recommended)* At least 1 GB memory on the machine used for the build.
- *(Recommended)* At least 20 GB hard disk space on the machine used for the build.

- Security-Enhanced Linux (SELinux) *disabled* on all Lustre servers and clients. The Lustre software does not support SELinux.

The installation procedure includes several steps:

- Patching the core kernel.
- Building Lustre RPMs.
- Installing and testing the Lustre file system.

These steps are described in the following sections.

30.2. Patching the Kernel

This section first describes how to prepare your machine to serve as a development environment, including build tools, the Lustre source, and the Linux kernel source. It then describes how to apply Lustre patches to the Linux kernel.

Note

A patched Linux kernel is *NOT* required in order to build the Lustre client code. The steps below for building the Linux kernel are only required if a new kernel is needed for a Lustre server.

30.2.1. Provisioning the Build Machine and Installing Dependencies

Note

This example is specific for building a modified Red Hat Enteprise Linux 6 kernel for the Lustre file system server. It is assumed that the operating system has been freshly installed on a machine with the hostname `rhel6-master`.

To provision the build machine and install dependencies, complete these steps.

1. Log in as `root`.
2. Install the kernel development tools:

```
# yum -y groupinstall "Development Tools"
```

Note

If the Development Tools group is not available, you will need to satisfy the dependencies for the build manually.

3. Install additional dependencies:

```
# yum -y install xmlto asciidoc  elfutils-libelf-devel
     zlib-devel binutils-devel newt-devel python-devel  hmaccalc
     perl-ExtUtils-Embed
```

4. (Optional) Install EPEL:

```
# wget
```

```
      http://download.fedoraproject.org/pub/epel/6/x86_64/
      epel-release-6-8.noarch.rpm
# rpm -ivh ./epel-release-5-4.noarch.rpm
```

5. (Optional) Install quilt:

```
# yum -y install quilt
```

Note

newt-devel may not be available for Red Hat Enterprise Linux 6. One option is to download the newt-devel [http://mirror.centos.org/centos/6/os/x86_64/Packages/newt-devel-0.52.11-3.el6.x86_64.rpm], slang-devel [http://mirror.centos.org/centos/6/os/x86_64/Packages/slang-devel-2.2.1-1.el6.x86_64.rpm], and asciidoc [http://mirror.centos.org/centos/6/os/i386/Packages/asciidoc-8.4.5-4.1.el6.noarch.rpm] RPMs from the CentOS Enterprise Linux Distribution and install using:

```
yum --nogpgcheck localinstall
   ./newt-devel-0.52.11-3.el6.x86_64.rpm
   ./slang-devel-2.2.1-1.el6.x86_64.rpm
   ./asciidoc-8.4.5-4.1.el6.noarch.rpm
```

30.2.2. Preparing the Lustre Source

To prepare the Lustre source, complete these steps.

1. Create a user called `build` with a home directory `/home/build`:

```
# useradd -m build
```

2. Switch to the user called `build` and change the directory to the users home directory:

```
# su build
$ cd $HOME
```

3. Get the MASTER branch of the Lustre software from the Lustre software git repository:

```
# git clone git://git.hpdd.intel.com/fs/lustre-release.git
# cd lustre-release
```

4. Run:

```
sh ./autogen.sh
```

5. Resolve any outstanding dependencies.

```
# sh ./autogen.sh
```

When autogen.sh completes successfully, a response similar to the following is displayed:

```
Checking for a complete tree...
checking forautomake-1.9>= 1.9... found 1.9.6
...
...
configure.ac:10: installing `./config.sub'
configure.ac:12: installing `./install-sh'
```

```
configure.ac:12: installing `./missing'
Running autoconf
```

30.2.3. Preparing the Kernel Source

To build the kernel using `rpmbuild` (a tool specific to RPM-based distributions), complete these steps.

1. Get the kernel source:

   ```
   # cd $HOME
   # mkdir -p kernel/rpmbuild/{BUILD,RPMS,SOURCES,SPECS,SRPMS}
   # cd kernel
   # echo '%_topdir %(echo $HOME)/kernel/rpmbuild'> ~/.rpmmacros
   ```

2. Install the kernel source (enter on one line):

   ```
   # rpm -ivh http://ftp.redhat.com/pub/redhat/linux/enterprise/
       6Server/en/os/SRPMS/kernel-2.6.32-358.14.1.el6.src.rpm
       2>&1 | grep -v mockb
   ```

   Note

   It is intended that the Lustre software master branch be kept up-to-date with the most recent kernel distributions. However, a delay may occur between a periodic update to a kernel distribution and a corresponding update of the master branch. The most recent supported kernel in the master branch can be found in the source directory in `lustre/kernel_patches/which_patch`. If the master branch is not current with the latest distribution, download the most recent kernel RPMs from the vendor's download site.

3. Prepare the source using `rpmbuild`:

   ```
   # cd ~/kernel/rpmbuild
   # rpmbuild -bp --target=`uname -m` ./SPECS/kernel.spec
   ```

   The text displayed ends with the following:

   ```
   ...
   gpg: Total number processed: 1
   gpg:               imported: 1
   + gpg --homedir . --export --keyring ./kernel.pub Red
   gpg: WARNING: unsafe permissions on homedir `.'
   + gcc -o scripts/bin2c scripts/bin2c.c
   + scripts/bin2c ksign_def_public_key __initdata
   + cd ..
   + exit 0
   ```

The kernel source with the Red Hat Enterprise Linux patches applied is now residing in the directory `/home/build/kernel/rpmbuild/BUILD/kernel-2.6.32-358.14.1.el6.x86_64/`

30.2.4. Patching the Kernel Source with the Lustre Code

To patch the kernel source with the Lustre code, complete these steps.

1. Go to the directory containing the kernel source:

   ```
   #cd ~/kernel/rpmbuild/BUILD/kernel-2.6.32-358.14.1.el6/
   ```

```
linux-2.6.32-358.14.1.el6.x86_64/
```

2. Edit the `Makefile` file in this directory to add a unique build id to be able to ascertain that the kernel is booted. Modify line 4 as shown below:

   ```
   EXTRAVERSION = -358.14.1.el6_lustre.x86_64
   ```

3. Overwrite the `.config` file in this directory with the Lustre `.config` file:

   ```
   # cp ~/lustre-release/lustre/kernel_patches/kernel_configs/
        kernel-2.6.32-2.6-rhel6-x86_64.config ./.config
   ```

4. Link the Lustre series and patches:

   ```
   # ln -s ~/lustre-release/lustre/kernel_patches/series/
        2.6-rhel6.series series
   # ln -s ~/lustre-release/lustre/kernel_patches/patches patches
   ```

5. Apply the patches to the kernel source using quilt:

   ```
   # quilt push -av
   ```

   The following is displayed:

   ```
   ...
   ...
   patching file fs/jbd2/transaction.c
   Hunk #3succeeded at 1222(offset 3lines).
   Hunk #4succeeded at 1357(offset 3lines).
   Now at patch  patches/jbd2-jcberr-2.6-rhel6.patch
   ```

   Note

   If quilt is not installed, the following steps would be equivalent:

   ```
   for PATCH in $(cat series); do patch -p1 < patches/$PATCH; done
   ```

30.3. Building the Lustre RPMs

This section describes how to configure the patched kernel to work with the Lustre software and how to create and install the Lustre packages (RPMs).

30.3.1. Building a New Kernel

To build a new kernel as an RPM, complete these steps.

1. In the kernel source directory, build a kernel rpm:

   ```
   # cd ~/kernel/rpmbuild/BUILD/kernel-2.6.32-131.2.1.el6/
        linux-2.6.32-131.2.1.el6.x86_64/
   # make oldconfig || make menuconfig
   # make include/asm
   # make include/linux/version.h
   # make SUBDIRS=scripts
   # make include/linux/utsrelease.h
   ```

```
# make rpm
```

A successful build returns text similar to the following:

```
...
...
Wrote: /home/build/kernel/rpmbuild/SRPMS/
     kernel-2.6.32-358.14.1.el6_lustre.x86_64-1.src.rpm
Wrote: /home/build/kernel/rpmbuild/RPMS/x86_64/
     kernel-2.6.32-358.14.1.el6_lustre.x86_64-1.x86_64.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.f73m1V
+ umask 022+ cd /home/build/kernel/rpmbuild/BUILD
+ cd kernel-2.6.32-358.14.1.el6_lustre.x86_64
+ rm -rf /home/build/kernel/rpmbuild/BUILDROOT/
     kernel-2.6.32-358.14.1.el6_lustre.x86_64-1.x86_64
+ exit 0
rm ../kernel-2.6.3-358.14.1.el6_lustre.x86_64.tar.gz
```

Note

If a request to generate more entropy appears, some disk or keyboard I/O needs to be generated. You can generate entropy by entering the following on another terminal:

```
# grep -Ri 'any_text' /usr
```

A fresh kernel RPM can now be found at `~/kernel/rpmbuild/RPMS/x86_64/kernel-2.6.32-358.14.1.el6_lustre.x86_64-1.x86_64.rpm`.

30.3.2. Configuring and Building Lustre RPMs

To configure and build a set of Lustre RPMs, complete these steps.

1. Configure the Lustre source:

   ```
   # cd ~/lustre-release/
   # ./configure --with-linux=/home/build/kernel/rpmbuild/BUILD/
        kernel-2.6.32-358.14.1.el6_lustre.x86_64/
   ```

   Text similar to the following is displayed:

   ```
   ...
   ...
   LLCPPFLAGS:    -D__arch_lib__ -D_LARGEFILE64_SOURCE=1
   CFLAGS:        -g -O2 -Werror
   EXTRA_KCFLAGS:  -include /home/build/lustre-release/config.h  -g
        -I/home/build/lustre-release
   /libcfs/include  -I/home/build/lustre-release/lnet/include
        -I/home/build/lustre-release/
   lustre/include
   LLCFLAGS:      -g -Wall -fPIC -D_GNU_SOURCE
   Type 'make' to build Lustre.
   ```

2. Create the RPMs:

   ```
   # make rpms
   ```

Text similar to the following is displayed:

```
...
...
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.TsLWpD
+ umask 022
+ cd /home/build/kernel/rpmbuild/BUILD
+ cd lustre-2.0.61
+ rm -rf /home/build/kernel/rpmbuild/BUILDROOT/
     lustre-2.0.61-2.6.32-358.14.1.el6_lustre.x86_64_g0533e7b.x86_64
+ exit 0
make[1]: Leaving directory `/home/build/lustre-release'
```

The resulting RPMs are in ~build/kernel/rpmbuild/RPMS/x86_64/.

```
kernel-2.6.3-358.14.1.el6_lustre.x86_64-1.x86_64.rpm
lustre-2.0.61-2.6.32-358.14.1.el6_lustre.x86_64_g0533e7b.x86_64.rpm
lustre-debuginfo-2.0.61-2.6.32-358.14.1.el6_lustre.x86_64_g0533e7b.x86_64.rpm
lustre-ldiskfs-3.3.0-2.6.32-358.14.1.el6_lustre.x86_64_g0533e7b.x86_64.rpm
lustre-ldiskfs-debuginfo-3.3.0-2.6.32-358.14.1.el6_lustre.x86_64_g0533e7b.x86_64.r
lustre-modules-2.0.61-2.6.32-358.14.1.el6_lustre.x86_64_g0533e7b.x86_64.rpm
lustre-source-2.0.61-2.6.32-358.14.1.el6_lustre.x86_64_g0533e7b.x86_64.rpm
lustre-tests-2.0.61-2.6.32-358.14.1.el6_lustre.x86_64_g0533e7b.x86_64.rpm
```

30.3.3. Installing the Lustre Kernel

To install the Lustre kernel, complete these steps.

1. As root, install the kernel:

   ```
   # rpm -ivh ~build/kernel/rpmbuild/RPMS/x86_64/
        kernel-2.6.32-358.14.1.el6_lustre.x86_64-1.x86_64.rpm
   ```

2. Create initrd using dracut:

   ```
   # /sbin/new-kernel-pkg --package kernel --mkinitrd --dracut
        --depmod --install 2.6.32-358.14.1.el6_lustre.x86_64
   ```

3. Reboot the system:

   ```
   reboot
   ```

   A login prompt such as that shown below indicates success:

   ```
   Red Hat Enterprise Linux Server release 6.0(Santiago)
   Kernel 2.6.3-358.14.1.el6_lustre.x86_64 on an x86_64

   client-10login:
   ```

30.3.4. Building a Lustre File System with a Third-party Network Stack on OFED (Optional)

When using third-party network hardware, you must follow a specific process to install and recompile the Lustre software. This section provides an installation example that shows how to build Lustre software

release 2.4.x when using the OpenFabrics Enterprise Distribution (OFED*) version 3.5-1 open-source software. The same process can be used for other third-party network stacks by replacing the OFED installation procedure and using the appropriate `--with` option when configuring the Lustre source code.

1. Install the Lustre kernel (see Section 30.3.3, “Installing the Lustre Kernel”).

2. Compile and install the OFED stack. As root, enter:

   ```
   # cd /usr/src
   # tar –zxvf OFED-3.5-1.tgz
   # cd OFED-3.5-1
   # ./install.pl
   ```

   A directory `/usr/src/compat-rdma` is created.

3. Configure and build Lustre source code:

   a. Install the Lustre source. This can be done via RPM or git (see Section 30.2.2, “Preparing the Lustre Source”).

   b. Configure and build the Lustre source code

      ```
      $ cd ~/lustre-release
      $ sh ./autogen.sh
      ```

The `./configure --help` command shows a list of all the `--with` options. All third-party network stacks are built in this manner.

```
$ ./configure --with-linux=/usr/src/kernels/$(uname –r) --with-o2ib=/usr/src/compa
$ make
$ make rpms
```

The `make rpms` command output shows the location of the generated RPMs.

30.4. Installing and Testing a Lustre File System

This section describes how to install the Lustre RPMs and run the Lustre test suite.

30.4.1. Installing e2fsprogs

The e2fsprogs package is required on all Lustre file system server nodes. To download and install `e2fsprogs`, complete these steps.

1. Download `e2fsprogs` from the Lustre Releases [https://wiki.hpdd.intel.com/display/PUB/Lustre+Releases] repository.

2. Install the `e2fsprogs` package:

   ```
   # rpm -Uvh
        ./e2fsprogs-1.42.7.wc2-7.el6.x86_64.rpm
        ./e2fsprogs-libs-1.42.7.wc2-7.el6.x86_64.rpm
   ```

30.4.2. Installing the Lustre RPMs

To install the Lustre RPMs, complete these steps as `root`:

1. Make the directory containing the Lustre RPMs your current directory:

   ```
   # cd /home/build/kernel/rpmbuild/RPMS/x86_64/
   ```

2. Install the RPMs:

   ```
   # rpm -ivh lustre-ldiskfs-3.3.0-2.6.32-358.14.1.el6_lustre.x86_64*
   # rpm -ivh lustre-modules-2.0.61-2.6.32-358.14.1.el6_lustre.x86_64*
   # rpm -ivh lustre-2.0.61-2.6.32-358.14.1.el6_lustre.x86_64_*
   # rpm -ivh lustre-tests-*
   ```

30.4.3. Running the Test Suite

To test a single node Lustre file system installation, complete these steps.

1. Mount the Lustre file system.

   ```
   # /usr/lib64/lustre/tests/llmount.sh
   ```

   Text similar to the following will be displayed:

   ```
   Loading modules from /usr/lib64/lustre/tests/..
   debug=0x33f0404
   subsystem_debug=0xffb7e3ff
   gss/krb5 is not supported
   Formatting mgs, mds, osts
   Format mds1: /tmp/lustre-mdt1
   Format ost1: /tmp/lustre-ost1
   Format ost2: /tmp/lustre-ost2
   Checking servers environments
   Checking clients rhel6-master environments
   Loading modules from /usr/lib64/lustre/tests/..
   debug=0x33f0404
   subsystem_debug=0xffb7e3ff
   gss/krb5 is not supported
   Setup mgs, mdt, osts
   Starting mds1: -o loop,user_xattr,acl
   /tmp/lustre-mdt1 /mnt/mds1
   debug=0x33f0404
   subsystem_debug=0xffb7e3ff
   debug_mb=10
   Started lustre-MDT0000
   Starting ost1: -o loop /tmp/lustre-ost1 /mnt/ost1
   debug=0x33f0404
   subsystem_debug=0xffb7e3ff
   debug_mb=10
   Started lustre-OST0000
   Starting ost2: -o loop /tmp/lustre-ost2 /mnt/ost2
   debug=0x33f0404
   subsystem_debug=0xffb7e3ff
   ```

```
debug_mb=10
Started lustre-OST0001
Starting client: rhel5-build: -o user_xattr,acl,flock
rhel6-master@tcp:/lustre /mnt/lustre
debug=0x33f0404
subsystem_debug=0xffb7e3ff
debug_mb=10
Using TIMEOUT=20
disable quota as required
```

The Lustre file system is now available at `/mnt/lustre`.

Warning

The Lustre file system available at `/mnt/lustre` is a test file system built from small loopback devices in `/tmp` and not for production use.

Note

If you see the error below, associate the IP address of a non-loopback interface with the name of your machine in the file `/etc/hosts`.

```
mkfs.lustre: Can't parse NID 'rhel6-master@tcp'
```

Chapter 31. Lustre File System Recovery

This chapter describes how recovery is implemented in a Lustre file system and includes the following sections:

- Section 31.1, " Recovery Overview"
- Section 31.2, "Metadata Replay"
- Section 31.3, "Reply Reconstruction"
- Section 31.4, "Version-based Recovery"
- Section 31.5, "Commit on Share"
- Section 31.6, "Imperative Recovery"

31.1. Recovery Overview

The recovery feature provided in the Lustre software is responsible for dealing with node or network failure and returning the cluster to a consistent, performant state. Because the Lustre software allows servers to perform asynchronous update operations to the on-disk file system (i.e., the server can reply without waiting for the update to synchronously commit to disk), the clients may have state in memory that is newer than what the server can recover from disk after a crash.

A handful of different types of failures can cause recovery to occur:

- Client (compute node) failure
- MDS failure (and failover)
- OST failure (and failover)
- Transient network partition

For Lustre software release 2.1.x and all earlier releases, all Lustre file system failure and recovery operations are based on the concept of connection failure; all imports or exports associated with a given connection are considered to fail if any of them fail. Lustre software release 2.2.x adds the Section 31.6, "Imperative Recovery" feature which enables the MGS to actively inform clients when a target restarts after a failure, failover or other interruption.

For information on Lustre file system recovery, see Section 31.2, "Metadata Replay". For information on recovering from a corrupt file system, see Section 31.5, "Commit on Share". For information on resolving orphaned objects, a common issue after recovery, see Section 28.2.1, "Working with Orphaned Objects". For information on imperative recovery see Section 31.6, "Imperative Recovery"

31.1.1. Client Failure

Recovery from client failure in a Lustre file system is based on lock revocation and other resources, so surviving clients can continue their work uninterrupted. If a client fails to timely respond to a blocking lock callback from the Distributed Lock Manager (DLM) or fails to communicate with the server in a long period of time (i.e., no pings), the client is forcibly removed from the cluster (evicted). This enables other

clients to acquire locks blocked by the dead client's locks, and also frees resources (file handles, export data) associated with that client. Note that this scenario can be caused by a network partition, as well as an actual client node system failure. Section 31.1.5, “Network Partition” describes this case in more detail.

31.1.2. Client Eviction

If a client is not behaving properly from the server's point of view, it will be evicted. This ensures that the whole file system can continue to function in the presence of failed or misbehaving clients. An evicted client must invalidate all locks, which in turn, results in all cached inodes becoming invalidated and all cached data being flushed.

Reasons why a client might be evicted:

- Failure to respond to a server request in a timely manner
  - Blocking lock callback (i.e., client holds lock that another client/server wants)
  - Lock completion callback (i.e., client is granted lock previously held by another client)
  - Lock glimpse callback (i.e., client is asked for size of object by another client)
  - Server shutdown notification (with simplified interoperability)
- Failure to ping the server in a timely manner, unless the server is receiving no RPC traffic at all (which may indicate a network partition).

31.1.3. MDS Failure (Failover)

Highly-available (HA) Lustre file system operation requires that the metadata server have a peer configured for failover, including the use of a shared storage device for the MDT backing file system. The actual mechanism for detecting peer failure, power off (STONITH) of the failed peer (to prevent it from continuing to modify the shared disk), and takeover of the Lustre MDS service on the backup node depends on external HA software such as Heartbeat. It is also possible to have MDS recovery with a single MDS node. In this case, recovery will take as long as is needed for the single MDS to be restarted.

When Section 31.6, “Imperative Recovery” is enabled, clients are notified of an MDS restart (either the backup or a restored primary). Clients always may detect an MDS failure either by timeouts of in-flight requests or idle-time ping messages. In either case the clients then connect to the new backup MDS and use the Metadata Replay protocol. Metadata Replay is responsible for ensuring that the backup MDS re-acquires state resulting from transactions whose effects were made visible to clients, but which were not committed to the disk.

The reconnection to a new (or restarted) MDS is managed by the file system configuration loaded by the client when the file system is first mounted. If a failover MDS has been configured (using the `--failnode=` option to `mkfs.lustre` or `tunefs.lustre`), the client tries to reconnect to both the primary and backup MDS until one of them responds that the failed MDT is again available. At that point, the client begins recovery. For more information, see Section 31.2, “Metadata Replay”.

Transaction numbers are used to ensure that operations are replayed in the order they were originally performed, so that they are guaranteed to succeed and present the same file system state as before the failure. In addition, clients inform the new server of their existing lock state (including locks that have not yet been granted). All metadata and lock replay must complete before new, non-recovery operations are permitted. In addition, only clients that were connected at the time of MDS failure are permitted to reconnect during the recovery window, to avoid the introduction of state changes that might conflict with what is being replayed by previously-connected clients.

Introduced in Lustre 2.4

Lustre software release 2.4 introduces multiple metadata targets. If multiple metadata targets are in use, active-active failover is possible. See Section 3.2.2, “MDT Failover Configuration (Active/Active)” for more information.

31.1.4. OST Failure (Failover)

When an OST fails or has communication problems with the client, the default action is that the corresponding OSC enters recovery, and I/O requests going to that OST are blocked waiting for OST recovery or failover. It is possible to administratively mark the OSC as *inactive* on the client, in which case file operations that involve the failed OST will return an IO error (`-EIO`). Otherwise, the application waits until the OST has recovered or the client process is interrupted (e.g. ,with *CTRL-C*).

The MDS (via the LOV) detects that an OST is unavailable and skips it when assigning objects to new files. When the OST is restarted or re-establishes communication with the MDS, the MDS and OST automatically perform orphan recovery to destroy any objects that belong to files that were deleted while the OST was unavailable. For more information, see Chapter 28, *Troubleshooting Recovery* (Working with Orphaned Objects).

While the OSC to OST operation recovery protocol is the same as that between the MDC and MDT using the Metadata Replay protocol, typically the OST commits bulk write operations to disk synchronously and each reply indicates that the request is already committed and the data does not need to be saved for recovery. In some cases, the OST replies to the client before the operation is committed to disk (e.g. truncate, destroy, setattr, and I/O operations in newer releases of the Lustre software), and normal replay and resend handling is done, including resending of the bulk writes. In this case, the client keeps a copy of the data available in memory until the server indicates that the write has committed to disk.

To force an OST recovery, unmount the OST and then mount it again. If the OST was connected to clients before it failed, then a recovery process starts after the remount, enabling clients to reconnect to the OST and replay transactions in their queue. When the OST is in recovery mode, all new client connections are refused until the recovery finishes. The recovery is complete when either all previously-connected clients reconnect and their transactions are replayed or a client connection attempt times out. If a connection attempt times out, then all clients waiting to reconnect (and their transactions) are lost.

Note

If you know an OST will not recover a previously-connected client (if, for example, the client has crashed), you can manually abort the recovery using this command:

```
oss# lctl --device lustre_device_number abort_recovery
```

To determine an OST's device number and device name, run the `lctl dl` command. Sample `lctl dl` command output is shown below:

```
7 UP obdfilter ddn_data-OST0009 ddn_data-OST0009_UUID 1159
```

In this example, 7 is the OST device number. The device name is `ddn_data-OST0009`. In most instances, the device name can be used in place of the device number.

31.1.5. Network Partition

Network failures may be transient. To avoid invoking recovery, the client tries, initially, to re-send any timed out request to the server. If the resend also fails, the client tries to re-establish a connection to the

server. Clients can detect harmless partition upon reconnect if the server has not had any reason to evict the client.

If a request was processed by the server, but the reply was dropped (i.e., did not arrive back at the client), the server must reconstruct the reply when the client resends the request, rather than performing the same request twice.

31.1.6. Failed Recovery

In the case of failed recovery, a client is evicted by the server and must reconnect after having flushed its saved state related to that server, as described in Section 31.1.2, “Client Eviction”, above. Failed recovery might occur for a number of reasons, including:

- Failure of recovery
  - Recovery fails if the operations of one client directly depend on the operations of another client that failed to participate in recovery. Otherwise, Version Based Recovery (VBR) allows recovery to proceed for all of the connected clients, and only missing clients are evicted.
  - Manual abort of recovery
- Manual eviction by the administrator

31.2. Metadata Replay

Highly available Lustre file system operation requires that the MDS have a peer configured for failover, including the use of a shared storage device for the MDS backing file system. When a client detects an MDS failure, it connects to the new MDS and uses the metadata replay protocol to replay its requests.

Metadata replay ensures that the failover MDS re-accumulates state resulting from transactions whose effects were made visible to clients, but which were not committed to the disk.

31.2.1. XID Numbers

Each request sent by the client contains an XID number, which is a client-unique, monotonically increasing 64-bit integer. The initial value of the XID is chosen so that it is highly unlikely that the same client node reconnecting to the same server after a reboot would have the same XID sequence. The XID is used by the client to order all of the requests that it sends, until such a time that the request is assigned a transaction number. The XID is also used in Reply Reconstruction to uniquely identify per-client requests at the server.

31.2.2. Transaction Numbers

Each client request processed by the server that involves any state change (metadata update, file open, write, etc., depending on server type) is assigned a transaction number by the server that is a target-unique, monotonically increasing, server-wide 64-bit integer. The transaction number for each file system-modifying request is sent back to the client along with the reply to that client request. The transaction numbers allow the client and server to unambiguously order every modification to the file system in case recovery is needed.

Each reply sent to a client (regardless of request type) also contains the last committed transaction number that indicates the highest transaction number committed to the file system. The `ldiskfs` backing file system that the Lustre software uses enforces the requirement that any earlier disk operation will always be committed to disk before a later disk operation, so the last committed transaction number also reports that any requests with a lower transaction number have been committed to disk.

31.2.3. Replay and Resend

Lustre file system recovery can be separated into two distinct types of operations: *replay* and *resend*.

Replay operations are those for which the client received a reply from the server that the operation had been successfully completed. These operations need to be redone in exactly the same manner after a server restart as had been reported before the server failed. Replay can only happen if the server failed; otherwise it will not have lost any state in memory.

Resend operations are those for which the client never received a reply, so their final state is unknown to the client. The client sends unanswered requests to the server again in XID order, and again awaits a reply for each one. In some cases, resent requests have been handled and committed to disk by the server (possibly also having dependent operations committed), in which case, the server performs reply reconstruction for the lost reply. In other cases, the server did not receive the lost request at all and processing proceeds as with any normal request. These are what happen in the case of a network interruption. It is also possible that the server received the request, but was unable to reply or commit it to disk before failure.

31.2.4. Client Replay List

All file system-modifying requests have the potential to be required for server state recovery (replay) in case of a server failure. Replies that have an assigned transaction number that is higher than the last committed transaction number received in any reply from each server are preserved for later replay in a per-server replay list. As each reply is received from the server, it is checked to see if it has a higher last committed transaction number than the previous highest last committed number. Most requests that now have a lower transaction number can safely be removed from the replay list. One exception to this rule is for open requests, which need to be saved for replay until the file is closed so that the MDS can properly reference count open-unlinked files.

31.2.5. Server Recovery

A server enters recovery if it was not shut down cleanly. If, upon startup, if any client entries are in the `last_rcvd` file for any previously connected clients, the server enters recovery mode and waits for these previously-connected clients to reconnect and begin replaying or resending their requests. This allows the server to recreate state that was exposed to clients (a request that completed successfully) but was not committed to disk before failure.

In the absence of any client connection attempts, the server waits indefinitely for the clients to reconnect. This is intended to handle the case where the server has a network problem and clients are unable to reconnect and/or if the server needs to be restarted repeatedly to resolve some problem with hardware or software. Once the server detects client connection attempts - either new clients or previously-connected clients - a recovery timer starts and forces recovery to finish in a finite time regardless of whether the previously-connected clients are available or not.

If no client entries are present in the `last_rcvd` file, or if the administrator manually aborts recovery, the server does not wait for client reconnection and proceeds to allow all clients to connect.

As clients connect, the server gathers information from each one to determine how long the recovery needs to take. Each client reports its connection UUID, and the server does a lookup for this UUID in the `last_rcvd` file to determine if this client was previously connected. If not, the client is refused connection and it will retry until recovery is completed. Each client reports its last seen transaction, so the server knows when all transactions have been replayed. The client also reports the amount of time that it was previously waiting for request completion so that the server can estimate how long some clients might need to detect the server failure and reconnect.

If the client times out during replay, it attempts to reconnect. If the client is unable to reconnect, REPLAY fails and it returns to DISCON state. It is possible that clients will timeout frequently during REPLAY, so reconnection should not delay an already slow process more than necessary. We can mitigate this by increasing the timeout during replay.

31.2.6. Request Replay

If a client was previously connected, it gets a response from the server telling it that the server is in recovery and what the last committed transaction number on disk is. The client can then iterate through its replay list and use this last committed transaction number to prune any previously-committed requests. It replays any newer requests to the server in transaction number order, one at a time, waiting for a reply from the server before replaying the next request.

Open requests that are on the replay list may have a transaction number lower than the server's last committed transaction number. The server processes those open requests immediately. The server then processes replayed requests from all of the clients in transaction number order, starting at the last committed transaction number to ensure that the state is updated on disk in exactly the same manner as it was before the crash. As each replayed request is processed, the last committed transaction is incremented. If the server receives a replay request from a client that is higher than the current last committed transaction, that request is put aside until other clients provide the intervening transactions. In this manner, the server replays requests in the same sequence as they were previously executed on the server until either all clients are out of requests to replay or there is a gap in a sequence.

31.2.7. Gaps in the Replay Sequence

In some cases, a gap may occur in the reply sequence. This might be caused by lost replies, where the request was processed and committed to disk but the reply was not received by the client. It can also be caused by clients missing from recovery due to partial network failure or client death.

In the case where all clients have reconnected, but there is a gap in the replay sequence the only possibility is that some requests were processed by the server but the reply was lost. Since the client must still have these requests in its resend list, they are processed after recovery is finished.

In the case where all clients have not reconnected, it is likely that the failed clients had requests that will no longer be replayed. The VBR feature is used to determine if a request following a transaction gap is safe to be replayed. Each item in the file system (MDS inode or OST object) stores on disk the number of the last transaction in which it was modified. Each reply from the server contains the previous version number of the objects that it affects. During VBR replay, the server matches the previous version numbers in the resend request against the current version number. If the versions match, the request is the next one that affects the object and can be safely replayed. For more information, see Section 31.4, “Version-based Recovery”.

31.2.8. Lock Recovery

If all requests were replayed successfully and all clients reconnected, clients then do lock replay locks -- that is, every client sends information about every lock it holds from this server and its state (whenever it was granted or not, what mode, what properties and so on), and then recovery completes successfully. Currently, the Lustre software does not do lock verification and just trusts clients to present an accurate lock state. This does not impart any security concerns since Lustre software release 1.x clients are trusted for other information (e.g. user ID) during normal operation also.

After all of the saved requests and locks have been replayed, the client sends an MDS_GETSTATUS request with last-replay flag set. The reply to that request is held back until all clients have completed replay (sent

the same flagged getstatus request), so that clients don't send non-recovery requests before recovery is complete.

31.2.9. Request Resend

Once all of the previously-shared state has been recovered on the server (the target file system is up-to-date with client cache and the server has recreated locks representing the locks held by the client), the client can resend any requests that did not receive an earlier reply. This processing is done like normal request processing, and, in some cases, the server may do reply reconstruction.

31.3. Reply Reconstruction

When a reply is dropped, the MDS needs to be able to reconstruct the reply when the original request is re-sent. This must be done without repeating any non-idempotent operations, while preserving the integrity of the locking system. In the event of MDS failover, the information used to reconstruct the reply must be serialized on the disk in transactions that are joined or nested with those operating on the disk.

31.3.1. Required State

For the majority of requests, it is sufficient for the server to store three pieces of data in the `last_rcvd` file:

- XID of the request
- Resulting transno (if any)
- Result code (`req->rq_status`)

For open requests, the "disposition" of the open must also be stored.

31.3.2. Reconstruction of Open Replies

An open reply consists of up to three pieces of information (in addition to the contents of the "request log"):

- File handle
- Lock handle
- `mds_body` with information about the file created (for `O_CREAT`)

The disposition, status and request data (re-sent intact by the client) are sufficient to determine which type of lock handle was granted, whether an open file handle was created, and which resource should be described in the `mds_body`.

31.3.2.1. Finding the File Handle

The file handle can be found in the XID of the request and the list of per-export open file handles. The file handle contains the resource/FID.

31.3.2.2. Finding the Resource/fid

The file handle contains the resource/fid.

31.3.2.3. Finding the Lock Handle

The lock handle can be found by walking the list of granted locks for the resource looking for one with the appropriate remote file handle (present in the re-sent request). Verify that the lock has the right mode (determined by performing the disposition/request/status analysis above) and is granted to the proper client.

31.4. Version-based Recovery

The Version-based Recovery (VBR) feature improves Lustre file system reliability in cases where client requests (RPCs) fail to replay during recovery [1].

In pre-VBR releases of the Lustre software, if the MGS or an OST went down and then recovered, a recovery process was triggered in which clients attempted to replay their requests. Clients were only allowed to replay RPCs in serial order. If a particular client could not replay its requests, then those requests were lost as well as the requests of clients later in the sequence. The "downstream" clients never got to replay their requests because of the wait on the earlier client's RPCs. Eventually, the recovery period would time out (so the component could accept new requests), leaving some number of clients evicted and their requests and data lost.

With VBR, the recovery mechanism does not result in the loss of clients or their data, because changes in inode versions are tracked, and more clients are able to reintegrate into the cluster. With VBR, inode tracking looks like this:

- Each inode[2] stores a version, that is, the number of the last transaction (transno) in which the inode was changed.
- When an inode is about to be changed, a pre-operation version of the inode is saved in the client's data.
- The client keeps the pre-operation inode version and the post-operation version (transaction number) for replay, and sends them in the event of a server failure.
- If the pre-operation version matches, then the request is replayed. The post-operation version is assigned on all inodes modified in the request.

> **Note**
>
> An RPC can contain up to four pre-operation versions, because several inodes can be involved in an operation. In the case of a "rename" operation, four different inodes can be modified.

During normal operation, the server:

- Updates the versions of all inodes involved in a given operation
- Returns the old and new inode versions to the client with the reply

When the recovery mechanism is underway, VBR follows these steps:

1. VBR only allows clients to replay transactions if the affected inodes have the same version as during the original execution of the transactions, even if there is gap in transactions due to a missed client.

[1] There are two scenarios under which client RPCs are not replayed: (1) Non-functioning or isolated clients do not reconnect, and they cannot replay their RPCs, causing a gap in the replay sequence. These clients get errors and are evicted. (2) Functioning clients connect, but they cannot replay some or all of their RPCs that occurred after the gap caused by the non-functioning/isolated clients. These clients get errors (caused by the failed clients). With VBR, these requests have a better chance to replay because the "gaps" are only related to specific files that the missing client(s) changed.
[2] Usually, there are two inodes, a parent and a child.

2. The server attempts to execute every transaction that the client offers, even if it encounters a re-integration failure.

3. When the replay is complete, the client and server check if a replay failed on any transaction because of inode version mismatch. If the versions match, the client gets a successful re-integration message. If the versions do not match, then the client is evicted.

VBR recovery is fully transparent to users. It may lead to slightly longer recovery times if the cluster loses several clients during server recovery.

31.4.1. VBR Messages

The VBR feature is built into the Lustre file system recovery functionality. It cannot be disabled. These are some VBR messages that may be displayed:

```
DEBUG_REQ(D_WARNING, req, "Version mismatch during replay\n");
```

This message indicates why the client was evicted. No action is needed.

```
CWARN("%s: version recovery fails, reconnecting\n");
```

This message indicates why the recovery failed. No action is needed.

31.4.2. Tips for Using VBR

VBR will be successful for clients which do not share data with other client. Therefore, the strategy for reliable use of VBR is to store a client's data in its own directory, where possible. VBR can recover these clients, even if other clients are lost.

31.5. Commit on Share

The commit-on-share (COS) feature makes Lustre file system recovery more reliable by preventing missing clients from causing cascading evictions of other clients. With COS enabled, if some Lustre clients miss the recovery window after a reboot or a server failure, the remaining clients are not evicted.

Note

The commit-on-share feature is enabled, by default.

31.5.1. Working with Commit on Share

To illustrate how COS works, let's first look at the old recovery scenario. After a service restart, the MDS would boot and enter recovery mode. Clients began reconnecting and replaying their uncommitted transactions. Clients could replay transactions independently as long as their transactions did not depend on each other (one client's transactions did not depend on a different client's transactions). The MDS is able to determine whether one transaction is dependent on another transaction via the Section 31.4, “Version-based Recovery” feature.

If there was a dependency between client transactions (for example, creating and deleting the same file), and one or more clients did not reconnect in time, then some clients may have been evicted because their transactions depended on transactions from the missing clients. Evictions of those clients caused more clients to be evicted and so on, resulting in "cascading" client evictions.

COS addresses the problem of cascading evictions by eliminating dependent transactions between clients. It ensures that one transaction is committed to disk if another client performs a transaction dependent on the first one. With no dependent, uncommitted transactions to apply, the clients replay their requests independently without the risk of being evicted.

31.5.2. Tuning Commit On Share

Commit on Share can be enabled or disabled using the `mdt.commit_on_sharing` tunable (0/1). This tunable can be set when the MDS is created (`mkfs.lustre`) or when the Lustre file system is active, using the `lctl set/get_param` or `lctl conf_param` commands.

To set a default value for COS (disable/enable) when the file system is created, use:

```
--param mdt.commit_on_sharing=0/1
```

To disable or enable COS when the file system is running, use:

```
lctl set_param mdt.*.commit_on_sharing=0/1
```

Note

Enabling COS may cause the MDS to do a large number of synchronous disk operations, hurting performance. Placing the `ldiskfs` journal on a low-latency external device may improve file system performance.

31.6. Imperative Recovery

Imperative Recovery (IR) was first introduced in Lustre software release 2.2.0.

Large-scale Lustre file system implementations have historically experienced problems recovering in a timely manner after a server failure. This is due to the way that clients detect the server failure and how the servers perform their recovery. Many of the processes are driven by the RPC timeout, which must be scaled with system size to prevent false diagnosis of server death. The purpose of imperative recovery is to reduce the recovery window by actively informing clients of server failure. The resulting reduction in the recovery window will minimize target downtime and therefore increase overall system availability. Imperative Recovery does not remove previous recovery mechanisms, and client timeout-based recovery actions can occur in a cluster when IR is enabled as each client can still independently disconnect and reconnect from a target. In case of a mix of IR and non-IR clients connecting to an OST or MDT, the server cannot reduce its recovery timeout window, because it cannot be sure that all clients have been notified of the server restart in a timely manner. Even in such mixed environments the time to complete recovery may be reduced, since IR-enabled clients will still be notified to reconnect to the server promptly and allow recovery to complete as soon as the last non-IR client detects the server failure.

31.6.1. MGS role

The MGS now holds additional information about Lustre targets, in the form of a Target Status Table. Whenever a target registers with the MGS, there is a corresponding entry in this table identifying the target. This entry includes NID information, and state/version information for the target. When a client mounts the file system, it caches a locked copy of this table, in the form of a Lustre configuration log. When a target restart occurs, the MGS revokes the client lock, forcing all clients to reload the table. Any new targets will have an updated version number, the client detects this and reconnects to the restarted target. Since successful IR notification of server restart depends on all clients being registered with the MGS, and

there is no other node to notify clients in case of MGS restart, the MGS will disable IR for a period when it first starts. This interval is configurable, as shown in Section 31.6.2, “Tuning Imperative Recovery”

Because of the increased importance of the MGS in recovery, it is strongly recommended that the MGS node be separate from the MDS. If the MGS is co-located on the MDS node, then in case of MDS/MGS failure there will be no IR notification for the MDS restart, and clients will always use timeout-based recovery for the MDS. IR notification would still be used in the case of OSS failure and recovery.

Unfortunately, it’s impossible for the MGS to know how many clients have been successfully notified or whether a specific client has received the restarting target information. The only thing the MGS can do is tell the target that, for example, all clients are imperative recovery-capable, so it is not necessary to wait as long for all clients to reconnect. For this reason, we still require a timeout policy on the target side, but this timeout value can be much shorter than normal recovery.

31.6.2. Tuning Imperative Recovery

Imperative recovery has a default parameter set which means it can work without any extra configuration. However, the default parameter set only fits a generic configuration. The following sections discuss the configuration items for imperative recovery.

31.6.2.1. ir_factor

Ir_factor is used to control targets’ recovery window. If imperative recovery is enabled, the recovery timeout window on the restarting target is calculated by: *new timeout = recovery_time * ir_factor / 10* Ir_factor must be a value in range of [1, 10]. The default value of ir_factor is 5. The following example will set imperative recovery timeout to 80% of normal recovery timeout on the target testfs-OST0000:

```
lctl conf_param obdfilter.testfs-OST0000.ir_factor=8
```

Note

If this value is too small for the system, clients may be unnecessarily evicted

You can read the current value of the parameter in the standard manner with *lctl get_param*:

```
# lctl get_param obdfilter.testfs-OST0000.ir_factor
# obdfilter.testfs-OST0000.ir_factor=8
```

31.6.2.2. Disabling Imperative Recovery

Imperative recovery can be disabled manually by a mount option. For example, imperative recovery can be disabled on an OST by:

```
# mount -t lustre -onoir /dev/sda /mnt/ost1
```

Imperative recovery can also be disabled on the client side with the same mount option:

```
# mount -t lustre -onoir mymgsnid@tcp:/testfs /mnt/testfs
```

Note

When a single client is deactivated in this manner, the MGS will deactivate imperative recovery for the whole cluster. IR-enabled clients will still get notification of target restart, but targets will not be allowed to shorten the recovery window.

You can also disable imperative recovery globally on the MGS by writing `state=disabled' to the controlling procfs entry

```
# lctl set_param mgs.MGS.live.testfs="state=disabled"
```

The above command will disable imperative recovery for file system named *testfs*

31.6.2.3. Checking Imperative Recovery State - MGS

You can get the imperative recovery state from the MGS. Let's take an example and explain states of imperative recovery:

```
[mgs]$ lctl get_param mgs.MGS.live.testfs
...
imperative_recovery_state:
    state: full
    nonir_clients: 0
    nidtbl_version: 242
    notify_duration_total: 0.470000
    notify_duation_max: 0.041000
    notify_count: 38
```

Item	Meaning
state	• **full:** IR is working, all clients are connected and can be notified. • **partial:** some clients are not IR capable. • **disabled:** IR is disabled, no client notification. • **startup:** the MGS was just restarted, so not all clients may reconnect to the MGS.
nonir_clients	Number of non-IR capable clients in the system.
nidtbl_version	Version number of the target status table. Client version must match MGS.
notify_duration_total	[Seconds.microseconds] Total time spent by MGS notifying clients
notify_duration_max	[Seconds.microseconds] Maximum notification time for the MGS to notify a single IR client.
notify_count	Number of MGS restarts - to obtain average notification time, divide `notify_duration_total` by `notify_count`

31.6.2.4. Checking Imperative Recovery State - client

A `client' in IR means a Lustre client or a MDT. You can get the IR state on any node which running client or MDT, those nodes will always have an MGC running. An example from a client:

```
[client]$ lctl get_param mgc.*.ir_state
```

```
mgc.MGC192.168.127.6@tcp.ir_state=
imperative_recovery: ON
client_state:
    - { client: testfs-client, nidtbl_version: 242 }
```

An example from a MDT:

```
mgc.MGC192.168.127.6@tcp.ir_state=
imperative_recovery: ON
client_state:
    - { client: testfs-MDT0000, nidtbl_version: 242 }
```

Item	Meaning
imperative_recovery	imperative_recoverycan be ON or OFF. If it's OFF state, then IR is disabled by administrator at mount time. Normally this should be ON state.
client_state: client:	The name of the client
client_state: nidtbl_version	Version number of the target status table. Client version must match MGS.

31.6.2.5. Target Instance Number

The Target Instance number is used to determine if a client is connecting to the latest instance of a target. We use the lowest 32 bit of mount count as target instance number. For an OST you can get the target instance number of testfs-OST0001 in this way (the command is run from an OSS login prompt):

```
$ lctl get_param obdfilter.testfs-OST0001*.instance
obdfilter.testfs-OST0001.instance=5
```

From a client, query the relevant OSC:

```
$ lctl get_param osc.testfs-OST0001-osc-*.import |grep instance
    instance: 5
```

31.6.3. Configuration Suggestions for Imperative Recovery

We used to build the MGS and MDT0 on the same target to save a server node. However, to make IR work efficiently, we strongly recommend running the MGS node on a separate node for any significant Lustre file system installation. There are three main advantages of doing this:

1. Be able to notify clients if the MDT0 is dead
2. Load balance. The load on the MDS may be very high which may make the MGS unable to notify the clients in time
3. Safety. The MGS code is simpler and much smaller compared to the code of MDT. This means the chance of MGS down time due to a software bug is very low.

31.7. Suppressing Pings

On clusters with large numbers of clients and OSTs, OBD_PING messages may impose significant performance overheads. As an intermediate solution before a more self-contained one is built, Lustre software release 2.4 introduces an option to suppress pings, allowing ping overheads to be considerably reduced. Before turning on this option, administrators should consider the following requirements and understand the trade-offs involved:

- When suppressing pings, a target can not detect client deaths, since clients do not send pings that are only to keep their connections alive. Therefore, a mechanism external to the Lustre file system shall be set up to notify Lustre targets of client deaths in a timely manner, so that stale connections do not exist for too long and lock callbacks to dead clients do not always have to wait for timeouts.
- Without pings, a client has to rely on Imperative Recovery to notify it of target failures, in order to join recoveries in time. This dictates that the client shall eargerly keep its MGS connection alive. Thus, a highly available standalone MGS is recommended and, on the other hand, MGS pings are always sent regardless of how the option is set.
- If a client has uncommitted requests to a target and it is not sending any new requests on the connection, it will still ping that target even when pings should be suppressed. This is because the client needs to query the target's last committed transaction numbers in order to free up local uncommitted requests (and possibly other resources associated). However, these pings shall stop as soon as all the uncommitted requests have been freed or new requests need to be sent, rendering the pings unnecessary.

31.7.1. "suppress_pings" Kernel Module Parameter

The new option that controls whether pings are suppressed is implemented as the ptlrpc kernel module parameter "suppress_pings". Setting it to "1" on a server turns on ping suppressing for all targets on that server, while leaving it with the default value "0" gives previous pinging behavior. The parameter is ignored on clients and the MGS. While the parameter is recommended to be set persistently via the modprobe.conf(5) mechanism, it also accept online changes through sysfs. Note that an online change only affects connections established later; existing connections' pinging behaviors stay the same.

31.7.2. Client Death Notification

The required external client death notification shall write UUIDs of dead clients into targets' "evict_client" procfs entries like

```
/proc/fs/lustre/obdfilter/testfs-OST0000/evict_client
/proc/fs/lustre/obdfilter/testfs-OST0001/evict_client
/proc/fs/lustre/mdt/testfs-MDT0000/evict_client
```

Clients' UUIDs can be obtained from their "uuid" procfs entries like

```
/proc/fs/lustre/llite/testfs-ffff8800612bf800/uuid
```

Chapter 32. LustreProc

The /proc file system acts as an interface to internal data structures in the kernel. This chapter describes entries in /proc that are useful for tuning and monitoring aspects of a Lustre file system. It includes these sections:

- Section 32.10, “Enabling and Interpreting Debugging Logs”

.

32.1. Introduction to /proc

The /proc directory provides an interface to internal data structures in the kernel that enables monitoring and tuning of many aspects of Lustre file system and application performance These data structures include settings and metrics for components such as memory, networking, file systems, and kernel housekeeping routines, which are available throughout the hierarchical file layout in /proc.

Typically, metrics are accessed by reading from /proc files and settings are changed by writing to /proc files. Some data is server-only, some data is client-only, and some data is exported from the client to the server and is thus duplicated in both locations.

Note

In the examples in this chapter, # indicates a command is entered as root. Servers are named according to the convention *fsname-MDT|OSTnumber*. The standard UNIX wildcard designation (*) is used.

In most cases, information is accessed using the lctl get_param command and settings are changed using the lctl set_param command. Some examples are shown below:

- To obtain data from a Lustre client:

```
# lctl list_param osc.*
osc.testfs-OST0000-osc-ffff881071d5cc00
osc.testfs-OST0001-osc-ffff881071d5cc00
osc.testfs-OST0002-osc-ffff881071d5cc00
osc.testfs-OST0003-osc-ffff881071d5cc00
osc.testfs-OST0004-osc-ffff881071d5cc00
osc.testfs-OST0005-osc-ffff881071d5cc00
osc.testfs-OST0006-osc-ffff881071d5cc00
osc.testfs-OST0007-osc-ffff881071d5cc00
osc.testfs-OST0008-osc-ffff881071d5cc00
```

  In this example, information about OST connections available on a client is displayed (indicated by "osc").

- To see multiple levels of parameters, use multiple wildcards:

```
# lctl list_param osc.*.*
osc.testfs-OST0000-osc-ffff881071d5cc00.active
osc.testfs-OST0000-osc-ffff881071d5cc00.blocksize
osc.testfs-OST0000-osc-ffff881071d5cc00.checksum_type
```

```
osc.testfs-OST0000-osc-ffff881071d5cc00.checksums
osc.testfs-OST0000-osc-ffff881071d5cc00.connect_flags
osc.testfs-OST0000-osc-ffff881071d5cc00.contention_seconds
osc.testfs-OST0000-osc-ffff881071d5cc00.cur_dirty_bytes
...
osc.testfs-OST0000-osc-ffff881071d5cc00.rpc_stats
```

- To view a specific file, use `lctl get_param` :

```
# lctl get_param osc.lustre-OST0000-osc-ffff881071d5cc00.rpc_stats
```

For more information about using `lctl`, see Section 13.9.3, “Setting Parameters with `lctl`”.

Data can also be viewed using the `cat` command with the full path to the file. The form of the `cat` command is similar to that of the `lctl get_param` command with these differences. In the `cat` command:

- Replace the dots in the path with slashes.
- Prepend the path with the following as appropriate:

```
/proc/{fs,sys}/{lustre,lnet}
```

For example, an `lctl get_param` command may look like this:

```
# lctl get_param osc.*.uuid
osc.testfs-OST0000-osc-ffff881071d5cc00.uuid=594db456-0685-bd16-f59b-e72ee90e9819
osc.testfs-OST0001-osc-ffff881071d5cc00.uuid=594db456-0685-bd16-f59b-e72ee90e9819
...
```

The equivalent `cat` command looks like this:

```
# cat /proc/fs/lustre/osc/*/uuid
594db456-0685-bd16-f59b-e72ee90e9819
594db456-0685-bd16-f59b-e72ee90e9819
...
```

The `llstat` utility can be used to monitor some Lustre file system I/O activity over a specified time period. For more details, see Section 37.8, “ llstat”

Some data is imported from attached clients and is available in a directory called `exports` located in the corresponding per-service directory on a Lustre server. For example:

```
# ls /proc/fs/lustre/obdfilter/testfs-OST0000/exports/192.168.124.9\@o2ib1/
# hash ldlm_stats stats uuid
```

32.1.1. Identifying Lustre File Systems and Servers

Several `/proc` files on the MGS list existing Lustre file systems and file system servers. The examples below are for a Lustre file system called `testfs` with one MDT and three OSTs.

- To view all known Lustre file systems, enter:

```
mgs# lctl get_param mgs.*.filesystems
testfs
```

- To view the names of the servers in a file system in which least one server is running, enter:

  ```
  lctl get_param mgs.*.live.<filesystem name>
  ```

  For example:

  ```
  mgs# lctl get_param mgs.*.live.testfs
  fsname: testfs
  flags: 0x20     gen: 45
  testfs-MDT0000
  testfs-OST0000
  testfs-OST0001
  testfs-OST0002

  Secure RPC Config Rules:

  imperative_recovery_state:
      state: startup
      nonir_clients: 0
      nidtbl_version: 6
      notify_duration_total: 0.001000
      notify_duation_max:  0.001000
      notify_count: 4
  ```

- To view the names of all live servers in the file system as listed in `/proc/fs/lustre/devices`, enter:

  ```
  # lctl device_list
  0 UP mgs MGS MGS 11
  1 UP mgc MGC192.168.10.34@tcp 1f45bb57-d9be-2ddb-c0b0-5431a49226705
  2 UP mdt MDS MDS_uuid 3
  3 UP lov testfs-mdtlov testfs-mdtlov_UUID 4
  4 UP mds testfs-MDT0000 testfs-MDT0000_UUID 7
  5 UP osc testfs-OST0000-osc testfs-mdtlov_UUID 5
  6 UP osc testfs-OST0001-osc testfs-mdtlov_UUID 5
  7 UP lov testfs-clilov-ce63ca00 08ac6584-6c4a-3536-2c6d-b36cf9cbdaa04
  8 UP mdc testfs-MDT0000-mdc-ce63ca00 08ac6584-6c4a-3536-2c6d-b36cf9cbdaa05
  9 UP osc testfs-OST0000-osc-ce63ca00 08ac6584-6c4a-3536-2c6d-b36cf9cbdaa05
  10 UP osc testfs-OST0001-osc-ce63ca00 08ac6584-6c4a-3536-2c6d-b36cf9cbdaa05
  ```

  The information provided on each line includes:

  - Device number

  - Device status (UP, INactive, or STopping)

  - Device name

  - Device UUID

  - Reference count (how many users this device has)

- To display the name of any server, view the device label:

  ```
  mds# e2label /dev/sda
  testfs-MDT0000
  ```

32.2. Tuning Multi-Block Allocation (mballoc)

Capabilities supported by `mballoc` include:

- Pre-allocation for single files to help to reduce fragmentation.
- Pre-allocation for a group of files to enable packing of small files into large, contiguous chunks.
- Stream allocation to help decrease the seek rate.

The following `mballoc` tunables are available:

Field	Description
`mb_max_to_scan`	Maximum number of free chunks that `mballoc` finds before a final decision to avoid a livelock situation.
`mb_min_to_scan`	Minimum number of free chunks that `mballoc` searches before picking the best chunk for allocation. This is useful for small requests to reduce fragmentation of big free chunks.
`mb_order2_req`	For requests equal to 2^N, where N >= `mb_order2_req`, a fast search is done using a base 2 buddy allocation service.
`mb_small_req`	`mb_small_req` - Defines (in MB) the upper bound of "small requests".
`mb_large_req`	`mb_large_req` - Defines (in MB) the lower bound of "large requests". Requests are handled differently based on size: • < `mb_small_req` - Requests are packed together to form large, aggregated requests. • > `mb_small_req` and < `mb_large_req` - Requests are primarily allocated linearly. • > `mb_large_req` - Requests are allocated since hard disk seek time is less of a concern in this case. In general, small requests are combined to create larger requests, which are then placed close to one another to minimize the number of seeks required to access the data.

`mb_prealloc_table`	A table of values used to preallocate space when a new request is received. By default, the table looks like this:
	
	`prealloc_table`
	`4 8 16 32 64 128 256 512 1024 2048`
	
	When a new request is received, space is preallocated at the next higher increment specified in the table. For example, for requests of less than 4 file system blocks, 4 blocks of space are preallocated; for requests between 4 and 8, 8 blocks are preallocated; and so forth
	
	Although customized values can be entered in the table, the performance of general usage file systems will not typically be improved by modifying the table (in fact, in ext4 systems, the table values are fixed). However, for some specialized workloads, tuning the `prealloc_table` values may result in smarter preallocation decisions.

Field	Description
mb_group_prealloc	The amount of space (in kilobytes) preallocated for groups of small requests.

Buddy group cache information found in /proc/fs/ldiskfs/*disk_device*/mb_groups may be useful for assessing on-disk fragmentation. For example:

```
cat /proc/fs/ldiskfs/loop0/mb_groups
#group: free free frags first pa [ 2^0 2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
     2^10 2^11 2^12 2^13]
#0    : 2936 2936 1     42    0  [ 0   0   0   1   1   1   1   2   0   1
     2    0    0    0   ]
```

In this example, the columns show:

- #group number
- Available blocks in the group
- Blocks free on a disk
- Number of free fragments
- First free block in the group
- Number of preallocated chunks (not blocks)
- A series of available chunks of different sizes

32.3. Monitoring Lustre File System I/O

A number of system utilities are provided to enable collection of data related to I/O activity in a Lustre file system. In general, the data collected describes:

- Data transfer rates and throughput of inputs and outputs external to the Lustre file system, such as network requests or disk I/O operations performed
- Data about the throughput or transfer rates of internal Lustre file system data, such as locks or allocations.

Note

It is highly recommended that you complete baseline testing for your Lustre file system to determine normal I/O activity for your hardware, network, and system workloads. Baseline data will allow you to easily determine when performance becomes degraded in your system. Two particularly useful baseline statistics are:

- brw_stats – Histogram data characterizing I/O requests to the OSTs. For more details, see Section 32.3.5, “Monitoring the OST Block I/O Stream”.
- rpc_stats – Histogram data showing information about RPCs made by clients. For more details, see Section 32.3.1, “Monitoring the Client RPC Stream”.

32.3.1. Monitoring the Client RPC Stream

The rpc_stats file contains histogram data showing information about remote procedure calls (RPCs) that have been made since this file was last cleared. The histogram data can be cleared by writing any value into the rpc_stats file.

Example:

```
# lctl get_param osc.testfs-OST0000-osc-ffff810058d2f800.rpc_stats
snapshot_time:            1372786692.389858 (secs.usecs)
read RPCs in flight:      0
write RPCs in flight:     1
dio read RPCs in flight:  0
dio write RPCs in flight: 0
pending write pages:      256
pending read pages:       0

                        read                    write
pages per rpc   rpcs   % cum % |       rpcs   % cum %
1:                 0   0   0   |          0   0   0
2:                 0   0   0   |          1   0   0
4:                 0   0   0   |          0   0   0
8:                 0   0   0   |          0   0   0
16:                0   0   0   |          0   0   0
32:                0   0   0   |          2   0   0
64:                0   0   0   |          2   0   0
128:               0   0   0   |          5   0   0
256:             850 100 100   |      18346  99 100

                        read                    write
rpcs in flight  rpcs   % cum % |       rpcs   % cum %
0:                691  81  81   |       1740   9   9
1:                 48   5  86   |        938   5  14
2:                 29   3  90   |       1059   5  20
3:                 17   2  92   |       1052   5  26
4:                 13   1  93   |        920   5  31
5:                 12   1  95   |        425   2  33
6:                 10   1  96   |        389   2  35
7:                 30   3 100   |      11373  61  97
8:                  0   0 100   |        460   2 100

                        read                    write
offset          rpcs   % cum % |       rpcs   % cum %
0:                850 100 100   |      18347  99  99
1:                  0   0 100   |          0   0  99
2:                  0   0 100   |          0   0  99
4:                  0   0 100   |          0   0  99
8:                  0   0 100   |          0   0  99
16:                 0   0 100   |          1   0  99
32:                 0   0 100   |          1   0  99
64:                 0   0 100   |          3   0  99
128:                0   0 100   |          4   0 100
```

The header information includes:

- `snapshot_time` - UNIX epoch instant the file was read.
- `read RPCs in flight` - Number of read RPCs issued by the OSC, but not complete at the time of the snapshot. This value should always be less than or equal to `max_rpcs_in_flight`.

- `write RPCs in flight` - Number of write RPCs issued by the OSC, but not complete at the time of the snapshot. This value should always be less than or equal to `max_rpcs_in_flight`.
- `dio read RPCs in flight` - Direct I/O (as opposed to block I/O) read RPCs issued but not completed at the time of the snapshot.
- `dio write RPCs in flight` - Direct I/O (as opposed to block I/O) write RPCs issued but not completed at the time of the snapshot.
- `pending write pages` - Number of pending write pages that have been queued for I/O in the OSC.
- `pending read pages` - Number of pending read pages that have been queued for I/O in the OSC.

The tabular data is described in the table below. Each row in the table shows the number of reads or writes (`ios`) occurring for the statistic, the relative percentage (`%`) of total reads or writes, and the cumulative percentage (`cum %`) to that point in the table for the statistic.

Field	Description
pages per RPC	Shows cumulative RPC reads and writes organized according to the number of pages in the RPC. A single page RPC increments the `0:` row.
RPCs in flight	Shows the number of RPCs that are pending when an RPC is sent. When the first RPC is sent, the `0:` row is incremented. If the first RPC is sent while another RPC is pending, the `1:` row is incremented and so on.
offset	The page index of the first page read from or written to the object by the RPC.

Analysis:

This table provides a way to visualize the concurrency of the RPC stream. Ideally, you will see a large clump around the `max_rpcs_in_flight value`, which shows that the network is being kept busy.

For information about optimizing the client I/O RPC stream, see Section 32.4.1, “Tuning the Client I/O RPC Stream”.

32.3.2. Monitoring Client Activity

The `stats` file maintains statistics accumulate during typical operation of a client across the VFS interface of the Lustre file system. Only non-zero parameters are displayed in the file.

Client statistics are enabled by default.

Note

Statistics for all mounted file systems can be discovered by entering:

```
lctl get_param llite.*.stats
```

Example:

```
client# lctl get_param llite.*.stats
```

```
snapshot_time           1308343279.169704 secs.usecs
dirty_pages_hits        14819716 samples [regs]
dirty_pages_misses      81473472 samples [regs]
read_bytes              36502963 samples [bytes] 1 26843582 55488794
write_bytes             22985001 samples [bytes] 0 125912 3379002
brw_read                2279 samples [pages] 1 1 2270
ioctl                   186749 samples [regs]
open                    3304805 samples [regs]
close                   3331323 samples [regs]
seek                    48222475 samples [regs]
fsync                   963 samples [regs]
truncate                9073 samples [regs]
setxattr                19059 samples [regs]
getxattr                61169 samples [regs]
```

The statistics can be cleared by echoing an empty string into the `stats` file or by using the command:

```
lctl set_param llite.*.stats=0
```

The statistics displayed are described in the table below.

Entry	Description
`snapshot_time`	UNIX epoch instant the stats file was read.
`dirty_page_hits`	The number of write operations that have been satisfied by the dirty page cache. See Section 32.4.1, “Tuning the Client I/O RPC Stream” for more information about dirty cache behavior in a Lustre file system.
`dirty_page_misses`	The number of write operations that were not satisfied by the dirty page cache.
`read_bytes`	The number of read operations that have occurred. Three additional parameters are displayed: min The minimum number of bytes read in a single request since the counter was reset. max The maximum number of bytes read in a single request since the counter was reset. sum The accumulated sum of bytes of all read requests since the counter was reset.
`write_bytes`	The number of write operations that have occurred. Three additional parameters are displayed: min The minimum number of bytes written in a single request since the counter was reset. max The maximum number of bytes written in a single request since the counter was reset. sum The accumulated sum of bytes of all write requests since the counter was reset.
`brw_read`	The number of pages that have been read. Three additional parameters are displayed:

Entry	Description
	min The minimum number of bytes read in a single block read/write (brw) read request since the counter was reset. max The maximum number of bytes read in a single brw read requests since the counter was reset. sum The accumulated sum of bytes of all brw read requests since the counter was reset.
ioctl	The number of combined file and directory ioctl operations.
open	The number of open operations that have succeeded.
close	The number of close operations that have succeeded.
seek	The number of times seek has been called.
fsync	The number of times fsync has been called.
truncate	The total number of calls to both locked and lockless truncate.
setxattr	The number of times extended attributes have been set.
getxattr	The number of times value(s) of extended attributes have been fetched.

Analysis:

Information is provided about the amount and type of I/O activity is taking place on the client.

32.3.3. Monitoring Client Read-Write Offset Statistics

When the offset_stats parameter is set, statistics are maintained for occurrences of a series of read or write calls from a process that did not access the next sequential location. The OFFSET field is reset to 0 (zero) whenever a different file is read or written.

Note

By default, statistics are not collected in the offset_stats, extents_stats, and extents_stats_per_process files to reduce monitoring overhead when this information is not needed. The collection of statistics in all three of these files is activated by writing anything into any one of the files.

Example:

```
# lctl get_param llite.testfs-f57dee0.offset_stats
snapshot_time: 1155748884.591028 (secs.usecs)
                RANGE   RANGE    SMALLEST   LARGEST
R/W   PID       START   END      EXTENT     EXTENT    OFFSET
R     8385      0       128      128        128       0
R     8385      0       224      224        224       -128
W     8385      0       250      50         100       0
W     8385      100     1110     10         500       -150
W     8384      0       5233     5233       5233      0
R     8385      500     600      100        100       -610
```

In this example, snapshot_time is the UNIX epoch instant the file was read. The tabular data is described in the table below.

The `offset_stats` file can be cleared by entering:

```
lctl set_param llite.*.offset_stats=0
```

Field	Description
R/W	Indicates if the non-sequential call was a read or write
PID	Process ID of the process that made the read/write call.
RANGE START/RANGE END	Range in which the read/write calls were sequential.
SMALLEST EXTENT	Smallest single read/write in the corresponding range (in bytes).
LARGEST EXTENT	Largest single read/write in the corresponding range (in bytes).
OFFSET	Difference between the previous range end and the current range start.

Analysis:

This data provides an indication of how contiguous or fragmented the data is. For example, the fourth entry in the example above shows the writes for this RPC were sequential in the range 100 to 1110 with the minimum write 10 bytes and the maximum write 500 bytes. The range started with an offset of -150 from the `RANGE END` of the previous entry in the example.

32.3.4. Monitoring Client Read-Write Extent Statistics

For in-depth troubleshooting, client read-write extent statistics can be accessed to obtain more detail about read/write I/O extents for the file system or for a particular process.

Note

By default, statistics are not collected in the `offset_stats`, `extents_stats`, and `extents_stats_per_process` files to reduce monitoring overhead when this information is not needed. The collection of statistics in all three of these files is activated by writing anything into any one of the files.

32.3.4.1. Client-Based I/O Extent Size Survey

The `extent_stats` histogram in the `llite` directory shows the statistics for the sizes of the read/write I/O extents. This file does not maintain the per-process statistics.

Example:

```
# lctl get_param llite.testfs-*.extents_stats
snapshot_time:                    1213828728.348516 (secs.usecs)
                        read            |            write
extents          calls  %      cum%   |     calls  %     cum%

0K - 4K :        0      0      0      |     2      2     2
4K - 8K :        0      0      0      |     0      0     2
```

```
8K - 16K :       0      0      0      |     0      0     2
16K - 32K :      0      0      0      |     20     23    26
32K - 64K :      0      0      0      |     0      0     26
64K - 128K :     0      0      0      |     51     60    86
128K - 256K :    0      0      0      |     0      0     86
256K - 512K :    0      0      0      |     0      0     86
512K - 1024K :   0      0      0      |     0      0     86
1M - 2M :        0      0      0      |     11     13    100
```

In this example, `snapshot_time` is the UNIX epoch instant the file was read. The table shows cumulative extents organized according to size with statistics provided separately for reads and writes. Each row in the table shows the number of RPCs for reads and writes respectively (`calls`), the relative percentage of total calls (%), and the cumulative percentage to that point in the table of calls (`cum %`).

The file can be cleared by issuing the following command:

```
# lctl set_param llite.testfs-*.extents_stats=0
```

32.3.4.2. Per-Process Client I/O Statistics

The `extents_stats_per_process` file maintains the I/O extent size statistics on a per-process basis.

Example:

```
# lctl get_param llite.testfs-*.extents_stats_per_process
snapshot_time:                    1213828762.204440 (secs.usecs)
                        read            |             write
extents            calls   %      cum%    |      calls   %       cum%

PID: 11488
   0K - 4K :       0       0       0      |      0       0       0
   4K - 8K :       0       0       0      |      0       0       0
   8K - 16K :      0       0       0      |      0       0       0
   16K - 32K :     0       0       0      |      0       0       0
   32K - 64K :     0       0       0      |      0       0       0
   64K - 128K :    0       0       0      |      0       0       0
   128K - 256K :   0       0       0      |      0       0       0
   256K - 512K :   0       0       0      |      0       0       0
   512K - 1024K :  0       0       0      |      0       0       0
   1M - 2M :       0       0       0      |      10      100     100

PID: 11491
   0K - 4K :       0       0       0      |      0       0       0
   4K - 8K :       0       0       0      |      0       0       0
   8K - 16K :      0       0       0      |      0       0       0
   16K - 32K :     0       0       0      |      20      100     100

PID: 11424
   0K - 4K :       0       0       0      |      0       0       0
   4K - 8K :       0       0       0      |      0       0       0
   8K - 16K :      0       0       0      |      0       0       0
   16K - 32K :     0       0       0      |      0       0       0
   32K - 64K :     0       0       0      |      0       0       0
```

```
   64K - 128K :    0       0       0      |      16      100     100

PID: 11426
   0K - 4K :       0       0       0      |      1       100     100

PID: 11429
   0K - 4K :       0       0       0      |      1       100     100
```

This table shows cumulative extents organized according to size for each process ID (PID) with statistics provided separately for reads and writes. Each row in the table shows the number of RPCs for reads and writes respectively (`calls`), the relative percentage of total calls (%), and the cumulative percentage to that point in the table of calls (`cum %`).

32.3.5. Monitoring the OST Block I/O Stream

The `brw_stats` file in the `obdfilter` directory contains histogram data showing statistics for number of I/O requests sent to the disk, their size, and whether they are contiguous on the disk or not.

Example:

Enter on the OSS:

```
# lctl get_param obdfilter.testfs-OST0000.brw_stats
snapshot_time:         1372775039.769045 (secs.usecs)
                           read      |      write
pages per bulk r/w     rpcs  % cum % |  rpcs   % cum %
1:                     108 100 100   |    39   0   0
2:                       0   0 100   |     6   0   0
4:                       0   0 100   |     1   0   0
8:                       0   0 100   |     0   0   0
16:                      0   0 100   |     4   0   0
32:                      0   0 100   |    17   0   0
64:                      0   0 100   |    12   0   0
128:                     0   0 100   |    24   0   0
256:                     0   0 100   | 23142  99 100

                           read      |      write
discontiguous pages    rpcs  % cum % |  rpcs   % cum %
0:                     108 100 100   | 23245 100 100

                           read      |      write
discontiguous blocks   rpcs  % cum % |  rpcs   % cum %
0:                     108 100 100   | 23243  99  99
1:                       0   0 100   |     2   0 100

                           read      |      write
disk fragmented I/Os   ios   % cum % |   ios   % cum %
0:                      94  87  87   |     0   0   0
1:                      14  12 100   | 23243  99  99
2:                       0   0 100   |     2   0 100

                           read      |      write
disk I/Os in flight    ios   % cum % |   ios   % cum %
```

```
1:                      14 100 100   | 20896  89  89
2:                       0   0 100   |  1071   4  94
3:                       0   0 100   |   573   2  96
4:                       0   0 100   |   300   1  98
5:                       0   0 100   |   166   0  98
6:                       0   0 100   |   108   0  99
7:                       0   0 100   |    81   0  99
8:                       0   0 100   |    47   0  99
9:                       0   0 100   |     5   0 100

                         read        |       write
I/O time (1/1000s)     ios   % cum % |   ios   % cum %
1:                      94  87  87   |     0   0   0
2:                       0   0  87   |     7   0   0
4:                      14  12 100   |    27   0   0
8:                       0   0 100   |    14   0   0
16:                      0   0 100   |    31   0   0
32:                      0   0 100   |    38   0   0
64:                      0   0 100   | 18979  81  82
128:                     0   0 100   |   943   4  86
256:                     0   0 100   |  1233   5  91
512:                     0   0 100   |  1825   7  99
1K:                      0   0 100   |    99   0  99
2K:                      0   0 100   |     0   0  99
4K:                      0   0 100   |     0   0  99
8K:                      0   0 100   |    49   0 100

                         read        |       write
disk I/O size          ios   % cum % |   ios   % cum %
4K:                     14 100 100   |    41   0   0
8K:                      0   0 100   |     6   0   0
16K:                     0   0 100   |     1   0   0
32K:                     0   0 100   |     0   0   0
64K:                     0   0 100   |     4   0   0
128K:                    0   0 100   |    17   0   0
256K:                    0   0 100   |    12   0   0
512K:                    0   0 100   |    24   0   0
1M:                      0   0 100   | 23142  99 100
```

The tabular data is described in the table below. Each row in the table shows the number of reads and writes occurring for the statistic (`ios`), the relative percentage of total reads or writes (%), and the cumulative percentage to that point in the table for the statistic (`cum %`).

Field	Description
`pages per bulk r/w`	Number of pages per RPC request, which should match aggregate client `rpc_stats` (see Section 32.3.1, “Monitoring the Client RPC Stream”).
`discontiguous pages`	Number of discontinuities in the logical file offset of each page in a single RPC.
`discontiguous blocks`	Number of discontinuities in the physical block allocation in the file system for a single RPC.
`disk fragmented I/Os`	Number of I/Os that were not written entirely sequentially.

Field	Description
`disk I/Os in flight`	Number of disk I/Os currently pending.
`I/O time (1/1000s)`	Amount of time for each I/O operation to complete.
`disk I/O size`	Size of each I/O operation.

Analysis:

This data provides an indication of extent size and distribution in the file system.

32.4. Tuning Lustre File System I/O

Each OSC has its own tree of tunables. For example:

```
$ ls -d /proc/fs/testfs/osc/OSC_client_ost1_MNT_client_2 /localhost
/proc/fs/testfs/osc/OSC_uml0_ost1_MNT_localhost
/proc/fs/testfs/osc/OSC_uml0_ost2_MNT_localhost
/proc/fs/testfs/osc/OSC_uml0_ost3_MNT_localhost

$ ls /proc/fs/testfs/osc/OSC_uml0_ost1_MNT_localhost
blocksizefilesfree max_dirty_mb ost_server_uuid stats

...
```

The following sections describe some of the parameters that can be tuned in a Lustre file system.

32.4.1. Tuning the Client I/O RPC Stream

Ideally, an optimal amount of data is packed into each I/O RPC and a consistent number of issued RPCs are in progress at any time. To help optimize the client I/O RPC stream, several tuning variables are provided to adjust behavior according to network conditions and cluster size. For information about monitoring the client I/O RPC stream, see Section 32.3.1, “Monitoring the Client RPC Stream”.

RPC stream tunables include:

- `osc.`*`osc_instance`*`.max_dirty_mb` - Controls how many MBs of dirty data can be written and queued up in the OSC. POSIX file writes that are cached contribute to this count. When the limit is reached, additional writes stall until previously-cached writes are written to the server. This may be changed by writing a single ASCII integer to the file. Only values between 0 and 2048 or 1/4 of RAM are allowable. If 0 is specified, no writes are cached. Performance suffers noticeably unless you use large writes (1 MB or more).

  To maximize performance, the value for `max_dirty_mb` is recommended to be 4 * `max_pages_per_rpc` * `max_rpcs_in_flight`.

- `osc.`*`osc_instance`*`.cur_dirty_bytes` - A read-only value that returns the current number of bytes written and cached on this OSC.

- `osc.`*`osc_instance`*`.max_pages_per_rpc` - The maximum number of pages that will undergo I/O in a single RPC to the OST. The minimum setting is a single page and the maximum setting is 1024 (for systems with a `PAGE_SIZE` of 4 KB), with the default maximum of 1 MB in the RPC. It is also possible to specify a units suffix (e.g. `4M`), so that the RPC size can be specified independently of the client `PAGE_SIZE`.

- `osc.`*`osc_instance`*`.max_rpcs_in_flight` - The maximum number of concurrent RPCs in flight from an OSC to its OST. If the OSC tries to initiate an RPC but finds that it already has the same number of RPCs outstanding, it will wait to issue further RPCs until some complete. The minimum setting is 1 and maximum setting is 256.

  To improve small file I/O performance, increase the `max_rpcs_in_flight` value.

- `llite.`*`fsname-instance`*`/max_cache_mb` - Maximum amount of inactive data cached by the client (default is 3/4 of RAM). For example:

```
# lctl get_param llite.testfs-ce63ca00.max_cached_mb
128
```

Note

The value for *`osc_instance`* is typically *`fsname`*`-OST`*`ost_index`*`-osc-`*`mountpoint_instance`*, where the value for *`mountpoint_instance`* is unique to each mount point to allow associating osc, mdc, lov, lmv, and llite parameters with the same mount point. For example:

```
lctl get_param osc.testfs-OST0000-osc-ffff88107412f400.rpc_stats
osc.testfs-OST0000-osc-ffff88107412f400.rpc_stats=
snapshot_time:         1375743284.337839 (secs.usecs)
read RPCs in flight:  0
write RPCs in flight: 0
```

32.4.2. Tuning File Readahead and Directory Statahead

File readahead and directory statahead enable reading of data into memory before a process requests the data. File readahead reads file content data into memory and directory statahead reads metadata into memory. When readahead and statahead work well, a process that accesses data finds that the information it needs is available immediately when requested in memory without the delay of network I/O.

Introduced in Lustre 2.2

In Lustre software release 2.2.0, the directory statahead feature was improved to enhance directory traversal performance. The improvements primarily addressed two issues:

1. A race condition existed between the statahead thread and other VFS operations while processing asynchronous `getattr` RPC replies, causing duplicate entries in dcache. This issue was resolved by using statahead local dcache.

2. File size/block attributes pre-fetching was not supported, so the traversing thread had to send synchronous glimpse size RPCs to OST(s). This issue was resolved by using asynchronous glimpse lock (AGL) RPCs to pre-fetch file size/block attributes from OST(s).

32.4.2.1. Tuning File Readahead

File readahead is triggered when two or more sequential reads by an application fail to be satisfied by data in the Linux buffer cache. The size of the initial readahead is 1 MB. Additional readaheads grow linearly and increment until the readahead cache on the client is full at 40 MB.

Readahead tunables include:

- `llite.fsname-instance.max_read_ahead_mb` - Controls the maximum amount of data readahead on a file. Files are read ahead in RPC-sized chunks (1 MB or the size of the `read()` call, if larger) after the second sequential read on a file descriptor. Random reads are done at the size of the `read()` call only (no readahead). Reads to non-contiguous regions of the file reset the readahead algorithm, and readahead is not triggered again until sequential reads take place again.

  To disable readahead, set this tunable to 0. The default value is 40 MB.

- `llite.fsname-instance.max_read_ahead_whole_mb` - Controls the maximum size of a file that is read in its entirety, regardless of the size of the `read()`.

32.4.2.2. Tuning Directory Statahead and AGL

Many system commands, such as `ls –l`, `du`, and `find`, traverse a directory sequentially. To make these commands run efficiently, the directory statahead and asynchronous glimpse lock (AGL) can be enabled to improve the performance of traversing.

The statahead tunables are:

- `statahead_max` - Controls whether directory statahead is enabled and the maximum statahead window size (i.e., how many files can be pre-fetched by the statahead thread). By default, statahead is enabled and the value of `statahead_max` is 32.

  To disable statahead, run:

  ```
  lctl set_param llite.*.statahead_max=0
  ```

  To set the maximum statahead window size (*n*), run:

  ```
  lctl set_param llite.*.statahead_max=n
  ```

  The maximum value of *n* is 8192.

  The AGL can be controlled by entering:

  ```
  lctl set_param llite.*.statahead_agl=n
  ```

  The default value for *n* is 1, which enables the AGL. If *n* is 0, the AGL is disabled.

- `statahead_stats` - A read-only interface that indicates the current statahead and AGL statistics, such as how many times statahead/AGL has been triggered since the last mount, how many statahead/AGL failures have occurred due to an incorrect prediction or other causes.

  Note

  The AGL is affected by statahead because the inodes processed by AGL are built by the statahead thread, which means the statahead thread is the input of the AGL pipeline. So if statahead is disabled, then the AGL is disabled by force.

32.4.3. Tuning OSS Read Cache

The OSS read cache feature provides read-only caching of data on an OSS. This functionality uses the Linux page cache to store the data and uses as much physical memory as is allocated.

OSS read cache improves Lustre file system performance in these situations:

- Many clients are accessing the same data set (as in HPC applications or when diskless clients boot from the Lustre file system).
- One client is storing data while another client is reading it (i.e., clients are exchanging data via the OST).
- A client has very limited caching of its own.

OSS read cache offers these benefits:

- Allows OSTs to cache read data more frequently.
- Improves repeated reads to match network speeds instead of disk speeds.
- Provides the building blocks for OST write cache (small-write aggregation).

32.4.3.1. Using OSS Read Cache

OSS read cache is implemented on the OSS, and does not require any special support on the client side. Since OSS read cache uses the memory available in the Linux page cache, the appropriate amount of memory for the cache should be determined based on I/O patterns; if the data is mostly reads, then more cache is required than would be needed for mostly writes.

OSS read cache is managed using the following tunables:

- `read_cache_enable` - Controls whether data read from disk during a read request is kept in memory and available for later read requests for the same data, without having to re-read it from disk. By default, read cache is enabled (`read_cache_enable=1`).

  When the OSS receives a read request from a client, it reads data from disk into its memory and sends the data as a reply to the request. If read cache is enabled, this data stays in memory after the request from the client has been fulfilled. When subsequent read requests for the same data are received, the OSS skips reading data from disk and the request is fulfilled from the cached data. The read cache is managed by the Linux kernel globally across all OSTs on that OSS so that the least recently used cache pages are dropped from memory when the amount of free memory is running low.

  If read cache is disabled (`read_cache_enable=0`), the OSS discards the data after a read request from the client is serviced and, for subsequent read requests, the OSS again reads the data from disk.

  To disable read cache on all the OSTs of an OSS, run:

  ```
  root@oss1# lctl set_param obdfilter.*.read_cache_enable=0
  ```

  To re-enable read cache on one OST, run:

  ```
  root@oss1# lctl set_param obdfilter.{OST_name}.read_cache_enable=1
  ```

  To check if read cache is enabled on all OSTs on an OSS, run:

  ```
  root@oss1# lctl get_param obdfilter.*.read_cache_enable
  ```

- `writethrough_cache_enable` - Controls whether data sent to the OSS as a write request is kept in the read cache and available for later reads, or if it is discarded from cache when the write is completed. By default, the writethrough cache is enabled (`writethrough_cache_enable=1`).

  When the OSS receives write requests from a client, it receives data from the client into its memory and writes the data to disk. If the writethrough cache is enabled, this data stays in memory after the write

request is completed, allowing the OSS to skip reading this data from disk if a later read request, or partial-page write request, for the same data is received.

If the writethrough cache is disabled (`writethrough_cache_enabled=0`), the OSS discards the data after the write request from the client is completed. For subsequent read requests, or partial-page write requests, the OSS must re-read the data from disk.

Enabling writethrough cache is advisable if clients are doing small or unaligned writes that would cause partial-page updates, or if the files written by one node are immediately being accessed by other nodes. Some examples where enabling writethrough cache might be useful include producer-consumer I/O models or shared-file writes with a different node doing I/O not aligned on 4096-byte boundaries.

Disabling the writethrough cache is advisable when files are mostly written to the file system but are not re-read within a short time period, or files are only written and re-read by the same node, regardless of whether the I/O is aligned or not.

To disable the writethrough cache on all OSTs of an OSS, run:

```
root@oss1# lctl set_param obdfilter.*.writethrough_cache_enable=0
```

To re-enable the writethrough cache on one OST, run:

```
root@oss1# lctl set_param obdfilter.{OST_name}.writethrough_cache_enable=1
```

To check if the writethrough cache is enabled, run:

```
root@oss1# lctl set_param obdfilter.*.writethrough_cache_enable=1
```

- `readcache_max_filesize` - Controls the maximum size of a file that both the read cache and writethrough cache will try to keep in memory. Files larger than `readcache_max_filesize` will not be kept in cache for either reads or writes.

  Setting this tunable can be useful for workloads where relatively small files are repeatedly accessed by many clients, such as job startup files, executables, log files, etc., but large files are read or written only once. By not putting the larger files into the cache, it is much more likely that more of the smaller files will remain in cache for a longer time.

  When setting `readcache_max_filesize`, the input value can be specified in bytes, or can have a suffix to indicate other binary units such as `K` (kilobytes), `M` (megabytes), `G` (gigabytes), `T` (terabytes), or `P` (petabytes).

  To limit the maximum cached file size to 32 MB on all OSTs of an OSS, run:

  ```
  root@oss1# lctl set_param obdfilter.*.readcache_max_filesize=32M
  ```

  To disable the maximum cached file size on an OST, run:

  ```
  root@oss1# lctl set_param obdfilter.{OST_name}.readcache_max_filesize=-1
  ```

  To check the current maximum cached file size on all OSTs of an OSS, run:

  ```
  root@oss1# lctl get_param obdfilter.*.readcache_max_filesize
  ```

32.4.4. Enabling OSS Asynchronous Journal Commit

The OSS asynchronous journal commit feature asynchronously writes data to disk without forcing a journal flush. This reduces the number of seeks and significantly improves performance on some hardware.

Note

Asynchronous journal commit cannot work with direct I/O-originated writes (`O_DIRECT` flag set). In this case, a journal flush is forced.

When the asynchronous journal commit feature is enabled, client nodes keep data in the page cache (a page reference). Lustre clients monitor the last committed transaction number (`transno`) in messages sent from the OSS to the clients. When a client sees that the last committed `transno` reported by the OSS is at least equal to the bulk write `transno`, it releases the reference on the corresponding pages. To avoid page references being held for too long on clients after a bulk write, a 7 second ping request is scheduled (the default OSS file system commit time interval is 5 seconds) after the bulk write reply is received, so the OSS has an opportunity to report the last committed `transno`.

If the OSS crashes before the journal commit occurs, then intermediate data is lost. However, OSS recovery functionality incorporated into the asynchronous journal commit feature causes clients to replay their write requests and compensate for the missing disk updates by restoring the state of the file system.

By default, `sync_journal` is enabled (`sync_journal=1`), so that journal entries are committed synchronously. To enable asynchronous journal commit, set the `sync_journal` parameter to `0` by entering:

```
$ lctl set_param obdfilter.*.sync_journal=0
obdfilter.lol-OST0001.sync_journal=0
```

An associated `sync-on-lock-cancel` feature (enabled by default) addresses a data consistency issue that can result if an OSS crashes after multiple clients have written data into intersecting regions of an object, and then one of the clients also crashes. A condition is created in which the POSIX requirement for continuous writes is violated along with a potential for corrupted data. With `sync-on-lock-cancel` enabled, if a cancelled lock has any volatile writes attached to it, the OSS synchronously writes the journal to disk on lock cancellation. Disabling the `sync-on-lock-cancel` feature may enhance performance for concurrent write workloads, but it is recommended that you not disable this feature.

The `sync_on_lock_cancel` parameter can be set to the following values:

- `always` - Always force a journal flush on lock cancellation (default when `async_journal` is enabled).
- `blocking` - Force a journal flush only when the local cancellation is due to a blocking callback.
- `never` - Do not force any journal flush (default when `async_journal` is disabled).

For example, to set `sync_on_lock_cancel` to not to force a journal flush, use a command similar to:

```
$ lctl get_param obdfilter.*.sync_on_lock_cancel
obdfilter.lol-OST0001.sync_on_lock_cancel=never
```

32.5. Configuring Timeouts in a Lustre File System

In a Lustre file system, RPC timeouts are set using an adaptive timeouts mechanism, which is enabled by default. Servers track RPC completion times and then report back to clients estimates for completion times for future RPCs. Clients use these estimates to set RPC timeout values. If the processing of server

requests slows down for any reason, the server estimates for RPC completion increase, and clients then revise RPC timeout values to allow more time for RPC completion.

If the RPCs queued on the server approach the RPC timeout specified by the client, to avoid RPC timeouts and disconnect/reconnect cycles, the server sends an "early reply" to the client, telling the client to allow more time. Conversely, as server processing speeds up, RPC timeout values decrease, resulting in faster detection if the server becomes non-responsive and quicker connection to the failover partner of the server.

32.5.1. Configuring Adaptive Timeouts

The adaptive timeout parameters in the table below can be set persistently system-wide using `lctl conf_param` on the MGS. For example, the following command sets the `at_max` value for all servers and clients associated with the file system `testfs`:

```
lctl conf_param testfs.sys.at_max=1500
```

Note

Clients that access multiple Lustre file systems must use the same parameter values for all file systems.

Parameter	Description
`at_min`	Minimum adaptive timeout (in seconds). The default value is 0. The `at_min` parameter is the minimum processing time that a server will report. Ideally, `at_min` should be set to its default value. Clients base their timeouts on this value, but they do not use this value directly. If, for unknown reasons (usually due to temporary network outages), the adaptive timeout value is too short and clients time out their RPCs, you can increase the `at_min` value to compensate for this.
`at_max`	Maximum adaptive timeout (in seconds). The `at_max` parameter is an upper-limit on the service time estimate. If `at_max` is reached, an RPC request times out. Setting `at_max` to 0 causes adaptive timeouts to be disabled and a fixed timeout method to be used instead (see Section 32.5.2, “Setting Static Timeouts” **Note** If slow hardware causes the service estimate to increase beyond the default value of `at_max`, increase `at_max` to the maximum time you are willing to wait for an RPC completion.
`at_history`	Time period (in seconds) within which adaptive timeouts remember the slowest event that occurred. The default is 600.
`at_early_margin`	Amount of time before the Lustre server sends an early reply (in seconds). Default is 5.
`at_extra`	Incremental amount of time that a server requests with each early reply (in seconds). The server does not know how much time the RPC will take, so it asks for a fixed value. The default is 30, which provides a balance between sending too many early replies for the same RPC and overestimating the actual completion time.

Parameter	Description
	When a server finds a queued request about to time out and needs to send an early reply out, the server adds the `at_extra` value. If the time expires, the Lustre server drops the request, and the client enters recovery status and reconnects to restore the connection to normal status. If you see multiple early replies for the same RPC asking for 30-second increases, change the `at_extra` value to a larger number to cut down on early replies sent and, therefore, network load.
`ldlm_enqueue_min`	Minimum lock enqueue time (in seconds). The default is 100. The time it takes to enqueue a lock, `ldlm_enqueue`, is the maximum of the measured enqueue estimate (influenced by `at_min` and `at_max` parameters), multiplied by a weighting factor and the value of `ldlm_enqueue_min`. Lustre Distributed Lock Manager (LDLM) lock enqueues have a dedicated minimum value for `ldlm_enqueue_min`. Lock enqueue timeouts increase as the measured enqueue times increase (similar to adaptive timeouts).

32.5.1.1. Interpreting Adaptive Timeout Information

Adaptive timeout information can be obtained from the `timeouts` files in `/proc/fs/lustre/*/` on each server and client using the `lctl` command. To read information from a `timeouts` file, enter a command similar to:

```
# lctl get_param -n ost.*.ost_io.timeouts
service : cur 33  worst 34 (at 1193427052, 0d0h26m40s ago) 1 1 33 2
```

In this example, the `ost_io` service on this node is currently reporting an estimated RPC service time of 33 seconds. The worst RPC service time was 34 seconds, which occurred 26 minutes ago.

The output also provides a history of service times. Four "bins" of adaptive timeout history are shown, with the maximum RPC time in each bin reported. In both the 0-150s bin and the 150-300s bin, the maximum RPC time was 1. The 300-450s bin shows the worst (maximum) RPC time at 33 seconds, and the 450-600s bin shows a maximum of RPC time of 2 seconds. The estimated service time is the maximum value across the four bins (33 seconds in this example).

Service times (as reported by the servers) are also tracked in the client OBDs, as shown in this example:

```
# lctl get_param osc.*.timeouts
last reply : 1193428639, 0d0h00m00s ago
network    : cur  1 worst  2 (at 1193427053, 0d0h26m26s ago)  1  1  1  1
portal 6   : cur 33 worst 34 (at 1193427052, 0d0h26m27s ago) 33 33 33  2
portal 28  : cur  1 worst  1 (at 1193426141, 0d0h41m38s ago)  1  1  1  1
portal 7   : cur  1 worst  1 (at 1193426141, 0d0h41m38s ago)  1  0  1  1
portal 17  : cur  1 worst  1 (at 1193426177, 0d0h41m02s ago)  1  0  0  1
```

In this example, portal 6, the `ost_io` service portal, shows the history of service estimates reported by the portal.

Server statistic files also show the range of estimates including min, max, sum, and sumsq. For example:

```
# lctl get_param mdt.*.mdt.stats
...
```

```
req_timeout               6 samples [sec] 1 10 15 105
...
```

32.5.2. Setting Static Timeouts

The Lustre software provides two sets of static (fixed) timeouts, LND timeouts and Lustre timeouts, which are used when adaptive timeouts are not enabled.

- ***LND timeouts*** - LND timeouts ensure that point-to-point communications across a network complete in a finite time in the presence of failures, such as packages lost or broken connections. LND timeout parameters are set for each individual LND.

  LND timeouts are logged with the `S_LND` flag set. They are not printed as console messages, so check the Lustre log for `D_NETERROR` messages or enable printing of `D_NETERROR` messages to the console using:

  ```
  lctl set_param printk=+neterror
  ```

  Congested routers can be a source of spurious LND timeouts. To avoid this situation, increase the number of LNET router buffers to reduce back-pressure and/or increase LND timeouts on all nodes on all connected networks. Also consider increasing the total number of LNET router nodes in the system so that the aggregate router bandwidth matches the aggregate server bandwidth.

- ***Lustre timeouts*** - Lustre timeouts ensure that Lustre RPCs complete in a finite time in the presence of failures when adaptive timeouts are not enabled. Adaptive timeouts are enabled by default. To disable adaptive timeouts at run time, set `at_max` to 0 by running on the MGS:

  ```
  # lctl conf_param fsname.sys.at_max=0
  ```

  > **Note**
  >
  > Changing the status of adaptive timeouts at runtime may cause a transient client timeout, recovery, and reconnection.

  Lustre timeouts are always printed as console messages.

  If Lustre timeouts are not accompanied by LND timeouts, increase the Lustre timeout on both servers and clients. Lustre timeouts are set using a command such as the following:

  ```
  # lctl set_param timeout=30
  ```

  Lustre timeout parameters are described in the table below.

Parameter	Description
`timeout`	The time that a client waits for a server to complete an RPC (default 100s). Servers wait half this time for a normal client RPC to complete and a quarter of this time for a single bulk request (read or write of up to 4 MB) to complete. The client pings recoverable targets (MDS and OSTs) at one quarter of the timeout, and the server waits one and a half times the timeout before evicting a client for being "stale." Lustre client sends periodic 'ping' messages to servers with which it has had no communication for the specified period of time. Any network activity between a client and a server in the file system also serves as a ping.

Parameter	Description
ldlm_timeout	The time that a server waits for a client to reply to an initial AST (lock cancellation request). The default is 20s for an OST and 6s for an MDS. If the client replies to the AST, the server will give it a normal timeout (half the client timeout) to flush any dirty data and release the lock.
fail_loc	An internal debugging failure hook. The default value of 0 means that no failure will be triggered or injected.
dump_on_timeout	Triggers a dump of the Lustre debug log when a timeout occurs. The default value of 0 (zero) means a dump of the Lustre debug log will not be triggered.
dump_on_eviction	Triggers a dump of the Lustre debug log when an eviction occurs. The default value of 0 (zero) means a dump of the Lustre debug log will not be triggered.

32.6. Monitoring LNET

LNET information is located in /proc/sys/lnet in these files:

- peers - Shows all NIDs known to this node and provides information on the queue state.

  Example:

```
# lctl get_param peers
nid                refs   state  max  rtr  min   tx    min   queue
0@lo               1      ~rtr   0    0    0     0     0     0
192.168.10.35@tcp  1      ~rtr   8    8    8     8     6     0
192.168.10.36@tcp  1      ~rtr   8    8    8     8     6     0
192.168.10.37@tcp  1      ~rtr   8    8    8     8     6     0
```

  The fields are explained in the table below:

Field	Description
refs	A reference count.
state	If the node is a router, indicates the state of the router. Possible values are: • NA - Indicates the node is not a router. • up/down- Indicates if the node (router) is up or down.
max	Maximum number of concurrent sends from this peer.
rtr	Number of routing buffer credits.
min	Minimum number of routing buffer credits seen.
tx	Number of send credits.
min	Minimum number of send credits seen.
queue	Total bytes in active/queued sends.

  Credits are initialized to allow a certain number of operations (in the example above the table, eight as shown in the max column. LNET keeps track of the minimum number of credits ever seen over time showing the peak congestion that has occurred during the time monitored. Fewer available credits indicates a more congested resource.

The number of credits currently in flight (number of transmit credits) is shown in the `tx` column. The maximum number of send credits available is shown in the `max` column and never changes. The number of router buffers available for consumption by a peer is shown in the `rtr` column.

Therefore, `rtr – tx` is the number of transmits in flight. Typically, `rtr == max`, although a configuration can be set such that `max >= rtr`. The ratio of routing buffer credits to send credits (`rtr/tx`) that is less than `max` indicates operations are in progress. If the ratio `rtr/tx` is greater than `max`, operations are blocking.

LNET also limits concurrent sends and number of router buffers allocated to a single peer so that no peer can occupy all these resources.

- `nis` - Shows the current queue health on this node.

  Example:

```
# lctl get_param nis
nid                     refs    peer    max    tx    min
0@lo                    3       0       0      0     0
192.168.10.34@tcp       4       8       256    256   252
```

  The fields are explained in the table below.

Field	Description
`nid`	Network interface.
`refs`	Internal reference counter.
`peer`	Number of peer-to-peer send credits on this NID. Credits are used to size buffer pools.
`max`	Total number of send credits on this NID.
`tx`	Current number of send credits available on this NID.
`min`	Lowest number of send credits available on this NID.
`queue`	Total bytes in active/queued sends.

  Analysis:

  Subtracting `max` from `tx` (`max - tx`) yields the number of sends currently active. A large or increasing number of active sends may indicate a problem.

32.7. Allocating Free Space on OSTs

Free space is allocated using either a round-robin or a weighted algorithm. The allocation method is determined by the maximum amount of free-space imbalance between the OSTs. When free space is relatively balanced across OSTs, the faster round-robin allocator is used, which maximizes network balancing. The weighted allocator is used when any two OSTs are out of balance by more than a specified threshold.

Free space distribution can be tuned using these two `/proc` tunables:

- `qos_threshold_rr` - The threshold at which the allocation method switches from round-robin to weighted is set in this file. The default is to switch to the weighted algorithm when any two OSTs are out of balance by more than 17 percent.

- `qos_prio_free` - The weighting priority used by the weighted allocator can be adjusted in this file. Increasing the value of `qos_prio_free` puts more weighting on the amount of free space available on each OST and less on how stripes are distributed across OSTs. The default value is 91 percent. When the free space priority is set to 100, weighting is based entirely on free space and location is no longer used by the striping algorthm.

For more information about monitoring and managing free space, see Section 18.5, “Managing Free Space”.

32.8. Configuring Locking

The `lru_size` parameter is used to control the number of client-side locks in an LRU cached locks queue. LRU size is dynamic, based on load to optimize the number of locks available to nodes that have different workloads (e.g., login/build nodes vs. compute nodes vs. backup nodes).

The total number of locks available is a function of the server RAM. The default limit is 50 locks/1 MB of RAM. If memory pressure is too high, the LRU size is shrunk. The number of locks on the server is limited to *the number of OSTs per server * the number of clients * the value of the* `lru_size` *setting on the client* as follows:

- To enable automatic LRU sizing, set the `lru_size` parameter to 0. In this case, the `lru_size` parameter shows the current number of locks being used on the export. LRU sizing is enabled by default.
- To specify a maximum number of locks, set the `lru_size` parameter to a value other than zero but, normally, less than 100 * *number of CPUs in client.* It is recommended that you only increase the LRU size on a few login nodes where users access the file system interactively.

To clear the LRU on a single client, and, as a result, flush client cache without changing the `lru_size` value, run:

```
$ lctl set_param ldlm.namespaces.osc_name|mdc_name.lru_size=clear
```

If the LRU size is set to be less than the number of existing unused locks, the unused locks are canceled immediately. Use `echo clear` to cancel all locks without changing the value.

Note

The `lru_size` parameter can only be set temporarily using `lctl set_param`; it cannot be set permanently.

To disable LRU sizing, on the Lustre clients, run:

```
$ lctl set_param ldlm.namespaces.*osc*.lru_size=$((NR_CPU*100))
```

Replace *`NR_CPU`* with the number of CPUs on the node.

To determine the number of locks being granted, run:

```
$ lctl get_param ldlm.namespaces.*.pool.limit
```

32.9. Setting MDS and OSS Thread Counts

MDS and OSS thread counts tunable can be used to set the minimum and maximum thread counts or get the current number of running threads for the services listed in the table below.

Service	Description
mds.MDS.mdt	Main metadata operations service
mds.MDS.mdt_readpage	Metadata readdir service
mds.MDS.mdt_setattr	Metadata setattr/close operations service
ost.OSS.ost	Main data operations service
ost.OSS.ost_io	Bulk data I/O services
ost.OSS.ost_create	OST object pre-creation service
ldlm.services.ldlm_canceld	DLM lock cancel service
ldlm.services.ldlm_cbd	DLM lock grant service

For each service, an entry as shown below is created:

/proc/fs/lustre/*service*/*/threads_*min*|*max*|*started*

- To temporarily set this tunable, run:

  # lctl *get*|*set*_param *service*.threads_*min*|*max*|*started*

- To permanently set this tunable, run:

  # lctl conf_param *obdname*|*fsname.obdtype*.threads_*min*|*max*|*started*

Introduced in Lustre 2.5

For version 2.5 or later, run:

lctl set_param -P *service*.threads_*min*|*max*|*started*

The following examples show how to set thread counts and get the number of running threads for the service ost_io using the tunable *service*.threads_*min*|*max*|*started*.

- To get the number of running threads, run:

  ```
  # lctl get_param ost.OSS.ost_io.threads_started
  ost.OSS.ost_io.threads_started=128
  ```

- To set the number of threads to the maximum value (512), run:

  ```
  # lctl get_param ost.OSS.ost_io.threads_max
  ost.OSS.ost_io.threads_max=512
  ```

- To set the maximum thread count to 256 instead of 512 (to avoid overloading the storage or for an array with requests), run:

  ```
  # lctl set_param ost.OSS.ost_io.threads_max=256
  ost.OSS.ost_io.threads_max=256
  ```

- To set the maximum thread count to 256 instead of 512 permanently, run:

  ```
  # lctl conf_param testfs.ost.ost_io.threads_max=256
  ```

Introduced in Lustre 2.5

For version 2.5 or later, run:

```
# lctl set_param -P ost.OSS.ost_io.threads_max=256
ost.OSS.ost_io.threads_max=256
```

- To check if the `threads_max` setting is active, run:

```
# lctl get_param ost.OSS.ost_io.threads_max
ost.OSS.ost_io.threads_max=256
```

Note

If the number of service threads is changed while the file system is running, the change may not take effect until the file system is stopped and rest. If the number of service threads in use exceeds the new `threads_max` value setting, service threads that are already running will not be stopped.

See also Chapter 26, *Tuning a Lustre File System*

32.10. Enabling and Interpreting Debugging Logs

By default, a detailed log of all operations is generated to aid in debugging. Flags that control debugging are found in `/proc/sys/lnet/debug`.

The overhead of debugging can affect the performance of Lustre file system. Therefore, to minimize the impact on performance, the debug level can be lowered, which affects the amount of debugging information kept in the internal log buffer but does not alter the amount of information to goes into syslog. You can raise the debug level when you need to collect logs to debug problems.

The debugging mask can be set using "symbolic names". The symbolic format is shown in the examples below.

- To verify the debug level used, examine the `sysctl` that controls debugging by running:

```
# sysctl lnet.debug
lnet.debug = ioctl neterror warning error emerg ha config console
```

- To turn off debugging (except for network error debugging), run the following command on all nodes concerned:

```
# sysctl -w lnet.debug="neterror"
lnet.debug = neterror
```

- To turn off debugging completely, run the following command on all nodes concerned:

```
# sysctl -w lnet.debug=0
lnet.debug = 0
```

- To set an appropriate debug level for a production environment, run:

```
# sysctl -w lnet.debug="warning dlmtrace error emerg ha rpctrace vfstrace"
lnet.debug = warning dlmtrace error emerg ha rpctrace vfstrace
```

The flags shown in this example collect enough high-level information to aid debugging, but they do not cause any serious performance impact.

- To clear all flags and set new flags, run:

```
# sysctl -w lnet.debug="warning"
lnet.debug = warning
```

- To add new flags to flags that have already been set, precede each one with a "+":

```
# sysctl -w lnet.debug="+neterror +ha"
lnet.debug = +neterror +ha
# sysctl lnet.debug
lnet.debug = neterror warning ha
```

- To remove individual flags, precede them with a "-":

```
# sysctl -w lnet.debug="-ha"
lnet.debug = -ha
# sysctl lnet.debug
lnet.debug = neterror warning
```

- To verify or change the debug level, run commands such as the following: :

```
# lctl get_param debug
debug=
neterror warning
# lctl set_param debug=+ha
# lctl get_param debug
debug=
neterror warning ha
# lctl set_param debug=-warning
# lctl get_param debug
debug=
neterror ha
```

Debugging parameters include:

- `subsystem_debug` - Controls the debug logs for subsystems.
- `debug_path` - Indicates the location where the debug log is dumped when triggered automatically or manually. The default path is `/tmp/lustre-log`.

These parameters are also set using:

```
sysctl -w lnet.debug={value}
```

Additional useful parameters:

- `panic_on_lbug` - Causes "panic" to be called when the Lustre software detects an internal problem (an `LBUG` log entry); panic crashes the node. This is particularly useful when a kernel crash dump utility is configured. The crash dump is triggered when the internal inconsistency is detected by the Lustre software.
- `upcall` - Allows you to specify the path to the binary which will be invoked when an `LBUG` log entry is encountered. This binary is called with four parameters:

- The string "LBUG".

- The file where the LBUG occurred.

- The function name.

- The line number in the file

32.10.1. Interpreting OST Statistics

Note

See also Section 37.6, " llobdstat" (llobdstat) and Section 12.4, " CollectL " (collectl).

OST stats files can be used to provide statistics showing activity for each OST. For example:

```
# lctl get_param osc.testfs-OST0000-osc.stats
snapshot_time                     1189732762.835363
ost_create                 1
ost_get_info               1
ost_connect                1
ost_set_info               1
obd_ping                   212
```

Use the llstat utility to monitor statistics over time.

To clear the statistics, use the -c option to llstat. To specify how frequently the statistics should be reported (in seconds), use the -i option. In the example below, the -c option clears the statistics and -i10 option reports statistics every 10 seconds:

```
$ llstat -c -i10 /proc/fs/lustre/ost/OSS/ost_io/stats

/usr/bin/llstat: STATS on 06/06/07
        /proc/fs/lustre/ost/OSS/ost_io/ stats on 192.168.16.35@tcp
snapshot_time                               1181074093.276072

/proc/fs/lustre/ost/OSS/ost_io/stats @ 1181074103.284895
Name        Cur.  Cur. #
            Count Rate Events Unit  last   min    avg       max    stddev
req_waittime 8    0    8    [usec]  2078   34     259.75    868    317.49
req_qdepth   8    0    8    [reqs]  1      0      0.12      1      0.35
req_active   8    0    8    [reqs]  11     1      1.38      2      0.52
reqbuf_avail 8    0    8    [bufs]  511    63     63.88     64     0.35
ost_write    8    0    8    [bytes] 169767 72914  212209.62 387579 91874.29

/proc/fs/lustre/ost/OSS/ost_io/stats @ 1181074113.290180
Name        Cur.  Cur. #
            Count Rate Events Unit  last    min   avg       max    stddev
req_waittime 31   3    39   [usec]  30011   34    822.79    12245  2047.71
req_qdepth   31   3    39   [reqs]  0       0     0.03      1      0.16
req_active   31   3    39   [reqs]  58      1     1.77      3      0.74
reqbuf_avail 31   3    39   [bufs]  1977    63    63.79     64     0.41
ost_write    30   3    38   [bytes] 1028467 15019 315325.16 910694 197776.51
```

```
/proc/fs/lustre/ost/OSS/ost_io/stats @ 1181074123.325560
Name        Cur.  Cur. #
            Count Rate Events Unit  last    min    avg       max    stddev
req_waittime 21   2    60   [usec]  14970   34     784.32    12245  1878.66
req_qdepth   21   2    60   [reqs]  0       0      0.02      1      0.13
req_active   21   2    60   [reqs]  33      1      1.70      3      0.70
reqbuf_avail 21   2    60   [bufs]  1341    63     63.82     64     0.39
ost_write    21   2    59   [bytes] 7648424 15019  332725.08 910694 180397.87
```

The columns in this example are described in the table below.

Parameter	Description
Name	Name of the service event. See the tables below for descriptions of service events that are tracked.
Cur. Count	Number of events of each type sent in the last interval.
Cur. Rate	Number of events per second in the last interval.
# Events	Total number of such events since the events have been cleared.
Unit	Unit of measurement for that statistic (microseconds, requests, buffers).
last	Average rate of these events (in units/event) for the last interval during which they arrived. For instance, in the above mentioned case of `ost_destroy` it took an average of 736 microseconds per destroy for the 400 object destroys in the previous 10 seconds.
min	Minimum rate (in units/events) since the service started.
avg	Average rate.
max	Maximum rate.
stddev	Standard deviation (not measured in some cases)

Events common to all services are shown in the table below.

Parameter	Description
req_waittime	Amount of time a request waited in the queue before being handled by an available server thread.
req_qdepth	Number of requests waiting to be handled in the queue for this service.
req_active	Number of requests currently being handled.
reqbuf_avail	Number of unsolicited lnet request buffers for this service.

Some service-specific events of interest are described in the table below.

Parameter	Description
ldlm_enqueue	Time it takes to enqueue a lock (this includes file open on the MDS)

Parameter	Description
mds_reint	Time it takes to process an MDS modification record (includes create, mkdir, unlink, rename and setattr)

32.10.2. Interpreting MDT Statistics

Note

See also Section 37.6, " llobdstat" (llobdstat) and Section 12.4, " CollectL " (collectl).

MDT stats files can be used to track MDT statistics for the MDS. The example below shows sample output from an MDT stats file.

```
# lctl get_param mds.*-MDT0000.stats
snapshot_time                   1244832003.676892 secs.usecs
open                            2 samples [reqs]
close                           1 samples [reqs]
getxattr                        3 samples [reqs]
process_config                  1 samples [reqs]
connect                         2 samples [reqs]
disconnect                      2 samples [reqs]
statfs                          3 samples [reqs]
setattr                         1 samples [reqs]
getattr                         3 samples [reqs]
llog_init                       6 samples [reqs]
notify                          16 samples [reqs]
```

Chapter 33. User Utilities

This chapter describes user utilities and includes the following sections:

- Section 33.1, “ `lfs` ”
- Section 33.2, “ `lfs_migrate` ”
- Section 33.3, “ `filefrag` ”
- Section 33.4, “ `mount` ”
- Section 33.5, “Handling Timeouts”

33.1. `lfs`

The `lfs` utility can be used for user configuration routines and monitoring.

33.1.1. Synopsis

```
lfs
lfs changelog [--follow] mdt_name [startrec [endrec]]
lfs changelog_clear mdt_name id endrec
lfs check mds|osts|servers
lfs df [-i] [-h] [--pool]-p fsname[.pool] [path] [--lazy]
lfs find [[!] --atime|-A [-+]N] [[!] --mtime|-M [-+]N]
         [[!] --ctime|-C [-+]N] [--maxdepth|-D N] [--name|-n pattern]
         [--print|-p] [--print0|-P] [[!] --obd|-O ost_name[,ost_name...]]
         [[!] --size|-S [+-]N[kMGTPE]] --type |-t {bcdflpsD}]
         [[!] --gid|-g|--group|-G gname|gid]
         [[!] --uid|-u|--user|-U uname|uid]
         dirname|filename
lfs getname [-h]|[path...]
lfs getstripe [--obd|-O ost_name] [--quiet|-q] [--verbose|-v]
              [--count|-c] [--index|-i | --offset|-o]
              [--size|-s] [--pool|-p] [--directory|-d]
              [--recursive|-r] [--raw|-R] [-M] dirname|filename ...
lfs setstripe [--size|-s stripe_size] [--count|-c stripe_count]
              [--index|-i|--offset|-o start_ost_index]
              [--pool|-p pool]
              dirname|filename
lfs setstripe -d dir
lfs osts [path]
lfs poollist filesystem[.pool]|pathname
lfs quota [-q] [-v] [-h] [-o obd_uuid|-I ost_idx|-i mdt_idx]
          [-u username|uid|-g group|gid]
          /mount_point
lfs quota -t -u|-g /mount_point
lfs quotacheck [-ug] /mount_point
lfs quotachown [-i] /mount_point
lfs quotainv [-ug] [-f] /mount_point
lfs quotaon [-ugf] /mount_point
lfs quotaoff [-ug] /mount_point
```

```
lfs setquota {-u|--user|-g|--group} uname|uid|gname|gid
             [--block-softlimit block_softlimit]
             [--block-hardlimit block_hardlimit]
             [--inode-softlimit inode_softlimit]
             [--inode-hardlimit inode_hardlimit]
             /mount_point
lfs setquota -u|--user|-g|--group uname|uid|gname|gid
             [-b block_softlimit] [-B block_hardlimit]
             [-i inode-softlimit] [-I inode_hardlimit]
             /mount_point
lfs setquota -t -u|-g
             [--block-grace block_grace]
             [--inode-grace inode_grace]
             /mount_point
lfs setquota -t -u|-g
             [-b block_grace] [-i inode_grace]
             /mount_point
lfs help
```

Note

In the above example, the */mount_point* parameter refers to the mount point of the Lustre file system.

Note

The old lfs quota output was very detailed and contained cluster-wide quota statistics (including cluster-wide limits for a user/group and cluster-wide usage for a user/group), as well as statistics for each MDS/OST. Now, `lfs quota` has been updated to provide only cluster-wide statistics, by default. To obtain the full report of cluster-wide limits, usage and statistics, use the `-v` option with `lfs quota`.

33.1.2. Description

The `lfs` utility is used to create a new file with a specific striping pattern, determine the default striping pattern, gather the extended attributes (object numbers and location) for a specific file, find files with specific attributes, list OST information or set quota limits. It can be invoked interactively without any arguments or in a non-interactive mode with one of the supported arguments.

33.1.3. Options

The various `lfs` options are listed and described below. For a complete list of available options, type help at the `lfs` prompt.

Option	Description
`changelog`	Shows the metadata changes on an MDT. Start and end points are optional. The `--follow` option blocks on new changes; this option is only valid when run directly on the MDT node.
`changelog_clear`	Indicates that changelog records previous to *`endrec`* are no longer of interest to a particular consumer *`id`*, potentially allowing the MDT to free up disk space. An *`endrec`* of 0 indicates the

Option		Description
		current last record. Changelog consumers must be registered on the MDT node using `lctl`.
`check`		Displays the status of MDS or OSTs (as specified in the command) or all servers (MDS and OSTs).
`df [-i] [-h] [--pool|-p` *`fsname`*`[.`*`pool`*`] [`*`path`*`] [--lazy]`		Use `-i` to report file system disk space usage or inode usage of each MDT or OST or, if a pool is specified with the `-p` option, a subset of OSTs. By default, the usage of all mounted Lustre file systems is reported. If the `path` option is included, only the usage for the specified file system is reported. If the `-h` option is included, the output is printed in human-readable format, using SI base-2 suffixes for **M**ega-, **G**iga-, **T**era-, **P**eta-, or **E**xabytes. If the `--lazy` option is specified, any OST that is currently disconnected from the client will be skipped. Using the `--lazy` option prevents the `df` output from being blocked when an OST is offline. Only the space on the OSTs that can currently be accessed are returned. The `llite.*.lazystatfs` tunable can be enabled to make this the default behaviour for all `statfs()` operations.
`find`		Searches the directory tree rooted at the given directory/filename for files that match the given parameters. Using `!` before an option negates its meaning (files NOT matching the parameter). Using `+` before a numeric value means files with the parameter OR MORE. Using `-` before a numeric value means files with the parameter OR LESS.
	`--atime`	File was last accessed N*24 hours ago. (There is no guarantee that `atime` is kept coherent across the cluster.) OSTs store a transient `atime` that is updated when clients do read requests. Permanent `atime` is written to the MDS when the file is closed. However, on-disk atime is only updated if it is more than 60 seconds old (`/proc/fs/lustre/mds/*/max_atime_diff`). The Lustre software considers the latest `atime` from all OSTs. If a `setattr` is set by user, then it is updated on both the MDS and OST, allowing the `atime` to go backward.
	`--ctime`	File status was last changed N*24 hours ago.
	`--mtime`	File data was last modified N*24 hours ago.

Option		Description
	`--obd`	File has an object on a specific OST(s).
	`--size`	File has a size in bytes, or kilo-, Mega-, Giga-, Tera-, Peta- or Exabytes if a suffix is given.
	`--type`	File has the type - block, character, directory, pipe, file, symlink, socket or door (used in Solaris operating system).
	`--uid`	File has a specific numeric user ID.
	`--user`	File owned by a specific user (numeric user ID allowed).
	`--gid`	File has a specific group ID.
	`--group`	File belongs to a specific group (numeric group ID allowed).
	`--maxdepth`	Limits find to descend at most N levels of the directory tree.
	`--print` / `--print0`	Prints the full filename, followed by a new line or NULL character correspondingly.
`osts [path]`		Lists all OSTs for the file system. If a path located on a mounted Lustre file system is specified, then only OSTs belonging to this file system are displayed.
`getname [path...]`		List each Lustre file system instance associated with each Lustre mount point. If no path is specified, all Lustre mount points are interrogated. If a list of paths is provided, the instance of each path is provided. If the path is not a Lustre instance 'No such device' is returned.
`getstripe`		Lists striping information for a given filename or directory. By default, the stripe count, stripe size and offset are returned. If you only want specific striping information, then the options of `--count`,`--size`,`--index` or `--offset` plus various combinations of these options can be used to retrieve specific information. If the `--raw` option is specified, the stripe information is printed without substituting the file system default values for unspecified fields. If the striping EA is not set, 0, 0, and -1 will be printed for the stripe count, size, and offset respectively. Introduced in Lustre 2.4 The `-M` prints the index of the MDT for a given directory. See Section 14.8.1, “Removing a MDT from the File System”.
	`--obd` *`ost_name`*	Lists files that have an object on a specific OST.
	`--quiet`	Lists details about the file's object ID information.
	`--verbose`	Prints additional striping information.

Option		Description
	`--count`	Lists the stripe count (how many OSTs to use).
	`--index`	Lists the index for each OST in the file system.
	`--offset`	Lists the OST index on which file striping starts.
	`--pool`	Lists the pools to which a file belongs.
	`--size`	Lists the stripe size (how much data to write to one OST before moving to the next OST).
	`--directory`	Lists entries about a specified directory instead of its contents (in the same manner as `ls -d`).
	`--recursive`	Recurses into all sub-directories.
`setstripe`		Create new files with a specific file layout (stripe pattern) configuration.[a]
	`--count stripe_cnt`	Number of OSTs over which to stripe a file. A `stripe_cnt` of 0 uses the file system-wide default stripe count (default is 1). A `stripe_cnt` of -1 stripes over all available OSTs.
	`--size stripe_size`[b]	Number of bytes to store on an OST before moving to the next OST. A stripe_size of 0 uses the file system's default stripe size, (default is 1 MB). Can be specified with **k** (KB), **m** (MB), or **g** (GB), respectively.
	`--index --offset start_ost_index`	The OST index (base 10, starting at 0) on which to start striping for this file. A start_ost_index value of -1 allows the MDS to choose the starting index. This is the default value, and it means that the MDS selects the starting OST as it wants. We strongly recommend selecting this default, as it allows space and load balancing to be done by the MDS as needed. The `start_ost_index` value has no relevance on whether the MDS will use round-robin or QoS weighted allocation for the remaining stripes in the file.
	`--pool` *`pool`*	Name of the pre-defined pool of OSTs (see Section 37.3, " lctl") that will be used for striping. The `stripe_cnt`, `stripe_size` and `start_ost` values are used as well. The start-ost value must be part of the pool or an error is returned.
`setstripe -d`		Deletes default striping on the specified directory.
`poollist {filesystem} [.poolname]| {pathname}`		Lists pools in the file system or pathname, or OSTs in the file system's pool.
`quota [-q] [-v] [-o` *`obd_uuid`*`|-i` *`mdt_idx`*`|-I` *`ost_idx`*`] [-u|-g` *`uname`*`|` *`uid|gname|gid] /mount_point`*		Displays disk usage and limits, either for the full file system or for objects on a specific OBD. A user or group name or an ID can be specified. If both user and group are omitted, quotas for the current UID/ GID are shown. The `-q` option disables printing of additional descriptions (including column titles). It fills in blank spaces in the "grace" column with zeros (when there is no grace period set), to ensure that

Option	Description
	the number of columns is consistent. The `-v` option provides more verbose (per-OBD statistics) output.
`quota -t -u|-g /mount_point`	Displays block and inode grace times for user (`-u`) or group (`-g`) quotas.
`quotachown`	Changes the file's owner and group on OSTs of the specified file system.
`quotacheck [-ugf] /mount_point`	Scans the specified file system for disk usage, and creates or updates quota files. Options specify quota for users (`-u`), groups (`-g`), and force (`-f`).
`quotaon [-ugf] /mount_point`	Turns on file system quotas. Options specify quota for users (`-u`), groups (`-g`), and force (`-f`).
`quotaoff [-ugf] /mount_point`	Turns off file system quotas. Options specify quota for users (`-u`), groups (`-g`), and force (`-f`).
`quotainv [-ug] [-f] /mount_point`	Clears quota files (administrative quota files if used without `-f`, operational quota files otherwise), all of their quota entries for users (`-u`) or groups (`-g`). After running `quotainv`, you must run `quotacheck` before using quotas. **Caution** Use extreme caution when using this command; its results cannot be undone.
`setquota -u|-g uname|uid|gname|gid} [--block-softlimit block_softlimit] [--block-hardlimit block_hardlimit] [--inode-softlimit inode_softlimit] [--inode-hardlimit inode_hardlimit] /mount_point`	Sets file system quotas for users or groups. Limits can be specified with `--{block|inode}-{softlimit|hardlimit}` or their short equivalents `-b`, `-B`, `-i`, `-I`. Users can set 1, 2, 3 or 4 limits.[c] Also, limits can be specified with special suffixes, -b, -k, -m, -g, -t, and -p to indicate units of 1, 2^10, 2^20, 2^30, 2^40 and 2^50, respectively. By default, the block limits unit is 1 kilobyte (1,024), and block limits are always kilobyte-grained (even if specified in bytes). See Section 33.1.4, “Examples”.
`setquota -t -u|-g [--block-grace block_grace] [--inode-grace inode_grace] /mount_point`	Sets the file system quota grace times for users or groups. Grace time is specified in 'XXwXXdXXhXXmXXs' format or as an integer seconds value. See Section 33.1.4, “Examples”.
`help`	Provides brief help on various `lfs` arguments.
`exit/quit`	Quits the interactive `lfs` session.

[a] The file cannot exist prior to using `setstripe`. A directory must exist prior to using `setstripe`.

[b] The default stripe-size is 0. The default start-ost is -1. Do NOT confuse them! If you set start-ost to 0, all new file creations occur on OST 0 (seldom a good idea).

[c] The old `setquota` interface is supported, but it may be removed in a future Lustre software release.

33.1.4. Examples

Creates a file striped on two OSTs with 128 KB on each stripe.

```
$ lfs setstripe -s 128k -c 2 /mnt/lustre/file1
```

Deletes a default stripe pattern on a given directory. New files use the default striping pattern.

```
$ lfs setstripe -d /mnt/lustre/dir
```

Lists the detailed object allocation of a given file.

```
$ lfs getstripe -v /mnt/lustre/file1
```

List all the mounted Lustre file systems and corresponding Lustre instances.

```
$ lfs getname
```

Efficiently lists all files in a given directory and its subdirectories.

```
$ lfs find /mnt/lustre
```

Recursively lists all regular files in a given directory more than 30 days old.

```
$ lfs find /mnt/lustre -mtime +30 -type f -print
```

Recursively lists all files in a given directory that have objects on OST2-UUID. The lfs check servers command checks the status of all servers (MDT and OSTs).

```
$ lfs find --obd OST2-UUID /mnt/lustre/
```

Lists all OSTs in the file system.

```
$ lfs osts
```

Lists space usage per OST and MDT in human-readable format.

```
$ lfs df -h
```

Lists inode usage per OST and MDT.

```
$ lfs df -i
```

List space or inode usage for a specific OST pool.

```
$ lfs df --pool filesystem[.pool] | pathname
```

List quotas of user 'bob'.

```
$ lfs quota -u bob /mnt/lustre
```

Show grace times for user quotas on `/mnt/lustre`.

```
$ lfs quota -t -u /mnt/lustre
```

Changes file owner and group.

```
$ lfs quotachown -i /mnt/lustre
```

Checks quotas for user and group. Turns on quotas after making the check.

```
$ lfs quotacheck -ug /mnt/lustre
```

Turns on quotas of user and group.

```
$ lfs quotaon -ug /mnt/lustre
```

Turns off quotas of user and group.

```
$ lfs quotaoff -ug /mnt/lustre
```

Sets quotas of user 'bob', with a 1 GB block quota hardlimit and a 2 GB block quota softlimit.

```
$ lfs setquota -u bob --block-softlimit 2000000 --block-hardlimit 1000000 /mnt/lus
```

Sets grace times for user quotas: 1000 seconds for block quotas, 1 week and 4 days for inode quotas.

```
$ lfs setquota -t -u --block-grace 1000 --inode-grace 1w4d /mnt/lustre
```

Checks the status of all servers (MDT, OST)

```
$ lfs check servers
```

Creates a file striped on two OSTs from the pool `my_pool`

```
$ lfs setstripe --pool my_pool -c 2 /mnt/lustre/file
```

Lists the pools defined for the mounted Lustre file system `/mnt/lustre`

```
$ lfs poollist /mnt/lustre/
```

Lists the OSTs which are members of the pool `my_pool` in file system `my_fs`

```
$ lfs poollist my_fs.my_pool
```

Finds all directories/files associated with `poolA`.

```
$ lfs find /mnt/lustre --pool poolA
```

Finds all directories/files not associated with a pool.

```
$ lfs find /mnt//lustre --pool ""
```

Finds all directories/files associated with pool.

```
$ lfs find /mnt/lustre ! --pool ""
```

Associates a directory with the pool `my_pool`, so all new files and directories are created in the pool.

```
$ lfs setstripe --pool my_pool /mnt/lustre/dir
```

33.1.5. See Also

Section 37.3, “ lctl”

33.2. `lfs_migrate`

The `lfs_migrate` utility is a simple tool to migrate files between Lustre OSTs.

33.2.1. Synopsis

```
lfs_migrate [-c stripecount] [-h] [-l] [-n] [-q] [-R] [-s] [-y] [file|directory ..
```

33.2.2. Description

The `lfs_migrate` utility is a simple tool to assist migration of files between Lustre OSTs. The utility copies each specified file to a new file, verifies the file contents have not changed, and then renames the new file to the original filename. This allows balanced space usage between OSTs, moving files of OSTs that are starting to show hardware problems (though are still functional) or OSTs that will be discontinued.

Because `lfs_migrate` is not closely integrated with the MDS, it cannot determine whether a file is currently open and/or in-use by other applications or nodes. This makes it UNSAFE for use on files that might be modified by other applications, since the migrated file is only a copy of the current file. This results in the old file becoming an open-unlinked file and any modifications to that file are lost.

Files to be migrated can be specified as command-line arguments. If a directory is specified on the command-line then all files within the directory are migrated. If no files are specified on the command-line, then a list of files is read from the standard input, making `lfs_migrate` suitable for use with `lfs` find to locate files on specific OSTs and/or matching other file attributes.

The current file allocation policies on the MDS dictate where the new files are placed, taking into account whether specific OSTs have been disabled on the MDS via `lctl` (preventing new files from being allocated there), whether some OSTs are overly full (reducing the number of files placed on those OSTs), or if there is a specific default file striping for the target directory (potentially changing the stripe count, stripe size, OST pool, or OST index of a new file).

33.2.3. Options

Options supporting `lfs_migrate` are described below.

Option	Description
`-c stripecount`	Restripe file using the specified stripe count. This option may not be specified at the same time as the `-R` option.
`-h`	Display help information.
`-l`	Migrate files with hard links (skips, by default). Files with multiple hard links are split into multiple separate files by `lfs_migrate`, so they are skipped, by default, to avoid breaking the hard links.
`-n`	Only print the names of files to be migrated.
`-q`	Run quietly (does not print filenames or status).
`-R`	Restripe file using default directory striping instead of keeping striping. This option may not be specified at the same time as the `-c` option.
`-s`	Skip file data comparison after migrate. Default is to compare migrated file against original to verify correctness.
`-y`	Answer '`y`' to usage warning without prompting (for scripts, use with caution).

33.2.4. Examples

Rebalances all files in `/mnt/lustre/dir`.

```
$ lfs_migrate /mnt/lustre/file
```

Migrates files in /test filesystem on OST004 larger than 4 GB in size.

```
$ lfs find /test -obd test-OST004 -size +4G | lfs_migrate -y
```

33.2.5. See Also

Section 33.1, " `lfs` "

33.3. `filefrag`

The `e2fsprogs` package contains the `filefrag` tool which reports the extent of file fragmentation.

33.3.1. Synopsis

```
filefrag [ -belsv ] [ files...  ]
```

33.3.2. Description

The `filefrag` utility reports the extent of fragmentation in a given file. Initially, `filefrag` attempts to obtain extent information using `FIEMAP ioctl`, which is efficient and fast. If `FIEMAP` is not supported, then `filefrag` uses `FIBMAP`.

Note

The Lustre software only supports `FIEMAP ioctl`. `FIBMAP ioctl` is not supported.

In default mode [1], `filefrag` returns the number of physically discontiguous extents in the file. In extent or verbose mode, each extent is printed with details. For a Lustre file system, the extents are printed in device offset order, not logical offset order.

33.3.3. Options

The options and descriptions for the `filefrag` utility are listed below.

Option	Description
`-b`	Uses the 1024-byte blocksize for the output. By default, this blocksize is used by the Lustre file system, since OSTs may use different block sizes.
`-e`	Uses the extent mode when printing the output.
`-l`	Displays extents in LUN offset order.
`-s`	Synchronizes the file before requesting the mapping.

[1] The default mode is faster than the verbose/extent mode.

Option	Description
-v	Uses the verbose mode when checking file fragmentation.

33.3.4. Examples

Lists default output.

```
$ filefrag /mnt/lustre/foo
/mnt/lustre/foo: 6 extents found
```

Lists verbose output in extent format.

```
$ filefrag  -ve /mnt/lustre/foo
Checking /mnt/lustre/foo
Filesystem type is: bd00bd0
Filesystem cylinder groups is approximately 5
File size of /mnt/lustre/foo is 157286400 (153600 blocks)
ext:       device_logical:         start..end              physical:
0: 0..                    49151:                  212992..                262144:
1: 49152..                73727:                  270336..                294912:
2: 73728..                76799:                  24576..                 27648:
3: 0..                    57343:                  196608..                253952:
4: 57344..                65535:                  139264..                147456:
5: 65536..                76799:                  163840..                175104:
/mnt/lustre/foo: 6 extents found
```

33.4. mount

The standard mount(8) Linux command is used to mount a Lustre file system. When mounting a Lustre file system, mount(8) executes the /sbin/mount.lustre command to complete the mount. The mount command supports these options specific to a Lustre file system:

Server options	Description
abort_recov	Aborts recovery when starting a target
nosvc	Starts only MGS/MGC servers
nomgs	Start a MDT with a co-located MGS without starting the MGS
exclude	Starts with a dead OST
md_stripe_cache_size	Sets the stripe cache size for server side disk with a striped raid configuration

Client options	Description
flock/noflock/localflock	Enables/disables global flock or local flock support
user_xattr/nouser_xattr	Enables/disables user-extended attributes
user_fid2path/nouser_fid2path	Introduced in Lustre 2.3 Enables/disables FID to path translation by regular users

Client options	Description
`retry=`	Number of times a client will retry to mount the file system

33.5. Handling Timeouts

Timeouts are the most common cause of hung applications. After a timeout involving an MDS or failover OST, applications attempting to access the disconnected resource wait until the connection gets established.

When a client performs any remote operation, it gives the server a reasonable amount of time to respond. If a server does not reply either due to a down network, hung server, or any other reason, a timeout occurs which requires a recovery.

If a timeout occurs, a message (similar to this one), appears on the console of the client, and in `/var/log/messages`:

```
LustreError: 26597:(client.c:810:ptlrpc_expire_one_request()) @@@ timeout

req@a2d45200 x5886/t0 o38->mds_svc_UUID@NID_mds_UUID:12 lens 168/64 ref 1 fl

RPC:/0/0 rc 0
```

Chapter 34. Programming Interfaces

This chapter describes public programming interfaces to that can be used to control various aspects of a Lustre file system from userspace. This chapter includes the following sections:

- Section 34.1, “User/Group Upcall”
- Section 34.2, “`l_getidentity` Utility”

Note

Lustre programming interface man pages are found in the `lustre/doc` folder.

34.1. User/Group Upcall

This section describes the supplementary user/group upcall, which allows the MDS to retrieve and verify the supplementary groups to which a particular user is assigned. This avoids the need to pass all the supplementary groups from the client to the MDS with every RPC.

Note

For information about universal UID/GID requirements in a Lustre file system ennvironment, see Section 8.1.2, “Environmental Requirements”.

34.1.1. Synopsis

The MDS uses the utility as specified by `lctl get_param mdt.${FSNAME}-MDT{xxxx}.identity_upcall` to look up the supplied UID in order to retrieve the user's supplementary group membership. The result is temporarily cached in the kernel (for five minutes, by default) to avoid the overhead of calling into userspace repeatedly.

34.1.2. Description

The identity upcall file contains the path to an executable that is invoked to resolve a numeric UID to a group membership list. This utility opens `/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_info` and fills in the related `identity_downcall_data` data structure (see Section 34.1.4, “Data Structures”). The data is persisted with `lctl set_param mdt.${FSNAME}-MDT{xxxx}.identity_info`.

For a sample upcall program, see `lustre/utils/l_getidentity.c` in the Lustre source distribution.

34.1.2.1. Primary and Secondary Groups

The mechanism for the primary/secondary group is as follows:

- The MDS issues an upcall (set per MDS) to map the numeric UID to the supplementary group(s).
- If there is no upcall or if there is an upcall and it fails, one supplementary group at most will be added as supplied by the client.
- The default upcall is `/usr/sbin/l_getidentity`, which can interact with the user/group database to obtain UID/GID/suppgid. The user/group database depends on how authentication is configured,

such as local `/etc/passwd`, Network Information Service (NIS), or Lightweight Directory Access Protocol (LDAP). If necessary, the administrator can use a parse utility to set `/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_upcall`. If the upcall interface is set to NONE, then upcall is disabled. The MDS uses the UID/GID/suppgid supplied by the client.

- The default group upcall is set by `mkfs.lustre`. Use `tunefs.lustre --param` or `lctl set_param mdt.${FSNAME}-MDT{xxxx}.identity_upcall={path}`

A Lustre file system administrator can specify permissions for a specific UID by configuring `/etc/lustre/perm.conf` on the MDS. The `/usr/sbin/l_getidentity` utility parses `/etc/lustre/perm.conf` to obtain the permission mask for a specified UID.

The permission file format is:

{*nid*} {*uid*} {*perms*}

An asterisk (*) in the *nid* column or *uid* column matches any NID or UID respectively. When '*' is specified as the NID, it is used for the default permissions for all NIDS, unless permissions are specified for a particular NID. In this case the specified permissions take precedence for that particular NID. Valid values for *perms* are:

- `setuid/setgid/setgrp/`*XXX* - enables the corresponding *perm*
- `nosetuid/nosetgid/nosetgrp/no`*XXX* - disables the corresponding *perm*

Permissions can be specified in a comma-separated list. When a *perm* and a *noperm* permission are listed in the same line, the *noperm* permission takes precedence. When they are listed on separate lines, the permission that appears later takes precedence.

- The `/usr/sbin/l_getidentity` utility can parse `/etc/lustre/perm.conf` to obtain the permission mask for the specified UID.
- To avoid repeated upcalls, the MDS caches supplemental group information. Use `lctl set_param mdt.*.identity_expire=<seconds>` to set the cache time. The default cache time is 600 seconds. The kernel waits for the upcall to complete (at most, 5 seconds) and takes the "failure" behavior as described.

  Set the wait time using `lctl set_param mdt.*.identity_acquire_expire=<seconds>` to change the length of time that the kernel waits for the upcall to finish. Note that the client process is blocked during this time. The default wait time is 15 seconds.

  Cached entries are flushed using `lctl set_param mdt.${FSNAME}-MDT{xxxx}.identity_flush=0`.

34.1.3. Parameters

- Name of the MDS service
- Numeric UID

34.1.4. Data Structures

```
struct perm_downcall_data{
     __u64 pdd_nid;
```

```
        __u32 pdd_perm;
        __u32 pdd_padding;
};

struct identity_downcall_data{
        __u32        idd_magic;
        :
        :
```

34.2. l_getidentity Utility

The l_getidentity utility handles the Lustre supplementary group upcall by default as described in the previous section.

34.2.1. Synopsis

```
l_getidentity ${FSNAME}-MDT{xxxx} {uid}
```

34.2.2. Description

The identity upcall file contains the path to an executable that is invoked to resolve a numeric UID to a group membership list. This utility opens /proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_info, completes the identity_downcall_data data structure (see Section 34.1.4, “Data Structures”) and writes the data to the /proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_info pseudo file. The data is persisted with lctl set_param mdt.${FSNAME}-MDT{xxxx}.identity_info.

l_getidentity is the reference implementation of the user/group cache upcall.

34.2.3. Files

```
/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_upcall

/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_info
```

Chapter 35. Setting Lustre Properties in a C Program (`llapi`)

This chapter describes the `llapi` library of commands used for setting Lustre file properties within a C program running in a cluster environment, such as a data processing or MPI application. The commands described in this chapter are:

- Section 35.1, " `llapi_file_create` "
- Section 35.2, "llapi_file_get_stripe"
- Section 35.3, " `llapi_file_open` "
- Section 35.4, " `llapi_quotactl` "
- Section 35.5, " `llapi_path2fid` "

Note

Lustre programming interface man pages are found in the `lustre/doc` folder.

35.1. `llapi_file_create`

Use `llapi_file_create` to set Lustre properties for a new file.

35.1.1. Synopsis

```
#include <lustre/lustreapi.h>

int llapi_file_create(char *name, long stripe_size, int stripe_offset, int stripe_
```

35.1.2. Description

The `llapi_file_create()` function sets a file descriptor's Lustre file system striping information. The file descriptor is then accessed with `open()`.

Option	Description
`llapi_file_create()`	If the file already exists, this parameter returns to '`EEXIST`'. If the stripe parameters are invalid, this parameter returns to '`EINVAL`'.
`stripe_size`	This value must be an even multiple of system page size, as shown by `getpagesize()`. The default Lustre stripe size is 4MB.
`stripe_offset`	Indicates the starting OST for this file.
`stripe_count`	Indicates the number of OSTs that this file will be striped across.
`stripe_pattern`	Indicates the RAID pattern.

Note

Currently, only RAID 0 is supported. To use the system defaults, set these values: stripe_size = 0, stripe_offset = -1, stripe_count = 0, stripe_pattern = 0

35.1.3. Examples

System default size is 4 MB.

```
char *tfile = TESTFILE;
int stripe_size = 65536
```

To start at default, run:

```
int stripe_offset = -1
```

To start at the default, run:

```
int stripe_count = 1
```

To set a single stripe for this example, run:

```
int stripe_pattern = 0
```

Currently, only RAID 0 is supported.

```
int stripe_pattern = 0;
int rc, fd;
rc = llapi_file_create(tfile, stripe_size,stripe_offset, stripe_count,stripe_patte
```

Result code is inverted, you may return with 'EINVAL' or an ioctl error.

```
if (rc) {
fprintf(stderr,"llapi_file_create failed: %d (%s) 0, rc, strerror(-rc));return -1;
```

llapi_file_create closes the file descriptor. You must re-open the descriptor. To do this, run:

```
fd = open(tfile, O_CREAT | O_RDWR | O_LOV_DELAY_CREATE, 0644); if (fd < 0) \ {
fprintf(stderr, "Can't open %s file: %s0, tfile,
str-
error(errno));
return -1;
}
```

35.2. llapi_file_get_stripe

Use llapi_file_get_stripe to get striping information for a file or directory on a Lustre file system.

35.2.1. Synopsis

```
#include <lustre/lustreapi.h>
```

```
int llapi_file_get_stripe(const char *path, void *lum);
```

35.2.2. Description

The `llapi_file_get_stripe()` function returns striping information for a file or directory *path* in *lum* (which should point to a large enough memory region) in one of the following formats:

```
struct lov_user_md_v1 {
__u32 lmm_magic;
__u32 lmm_pattern;
__u64 lmm_object_id;
__u64 lmm_object_seq;
__u32 lmm_stripe_size;
__u16 lmm_stripe_count;
__u16 lmm_stripe_offset;
struct lov_user_ost_data_v1 lmm_objects[0];
} __attribute__((packed));
struct lov_user_md_v3 {
__u32 lmm_magic;
__u32 lmm_pattern;
__u64 lmm_object_id;
__u64 lmm_object_seq;
__u32 lmm_stripe_size;
__u16 lmm_stripe_count;
__u16 lmm_stripe_offset;
char lmm_pool_name[LOV_MAXPOOLNAME];
struct lov_user_ost_data_v1 lmm_objects[0];
} __attribute__((packed));
```

Option	Description
`lmm_magic`	Specifies the format of the returned striping information. `LOV_MAGIC_V1` is used for lov_user_md_v1. LOV_MAGIC_V3 is used for `lov_user_md_v3`.
`lmm_pattern`	Holds the striping pattern. Only `LOV_PATTERN_RAID0` is possible in this Lustre software release.
`lmm_object_id`	Holds the MDS object ID.
`lmm_object_gr`	Holds the MDS object group.
`lmm_stripe_size`	Holds the stripe size in bytes.
`lmm_stripe_count`	Holds the number of OSTs over which the file is striped.
`lmm_stripe_offset`	Holds the OST index from which the file starts.
`lmm_pool_name`	Holds the OST pool name to which the file belongs.

`lmm_objects`	An array of `lmm_stripe_count` members containing per OST file information in
	
	the following format:
	
	`struct lov_user_ost_data_v1 {`
	`__u64 l_object_id;`

Option	Description
	__u64 l_object_seq; __u32 l_ost_gen; __u32 l_ost_idx; } __attribute__((packed));
l_object_id	Holds the OST's object ID.
l_object_seq	Holds the OST's object group.
l_ost_gen	Holds the OST's index generation.
l_ost_idx	Holds the OST's index in LOV.

35.2.3. Return Values

llapi_file_get_stripe() returns:

0 On success

!= 0 On failure, errno is set appropriately

35.2.4. Errors

Errors	Description
ENOMEM	Failed to allocate memory
ENAMETOOLONG	Path was too long
ENOENT	Path does not point to a file or directory
ENOTTY	Path does not point to a Lustre file system
EFAULT	Memory region pointed by lum is not properly mapped

35.2.5. Examples

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <lustre/lustreapi.h>

static inline int maxint(int a, int b)
{
 return a > b ? a : b;
}
static void *alloc_lum()
{
 int v1, v3, join;
 v1 = sizeof(struct lov_user_md_v1) +
  LOV_MAX_STRIPE_COUNT * sizeof(struct lov_user_ost_data_v1);
 v3 = sizeof(struct lov_user_md_v3) +
  LOV_MAX_STRIPE_COUNT * sizeof(struct lov_user_ost_data_v1);
 return malloc(maxint(v1, v3));
}
```

```
int main(int argc, char** argv)
{
 struct lov_user_md *lum_file = NULL;
 int rc;
 int lum_size;
 if (argc != 2) {
  fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
  return 1;
 }
 lum_file = alloc_lum();
 if (lum_file == NULL) {
  rc = ENOMEM;
  goto cleanup;
 }
 rc = llapi_file_get_stripe(argv[1], lum_file);
 if (rc) {
  rc = errno;
  goto cleanup;
 }
 /* stripe_size stripe_count */
 printf("%d %d\n",
    lum_file->lmm_stripe_size,
    lum_file->lmm_stripe_count);
cleanup:
 if (lum_file != NULL)
  free(lum_file);
 return rc;
}
```

35.3. llapi_file_open

The llapi_file_open command opens (or creates) a file or device on a Lustre file system.

35.3.1. Synopsis

```
#include <lustre/lustreapi.h>
int llapi_file_open(const char *name, int flags, int mode,
   unsigned long long stripe_size, int stripe_offset,
   int stripe_count, int stripe_pattern);
int llapi_file_create(const char *name, unsigned long long stripe_size,
   int stripe_offset, int stripe_count,
   int stripe_pattern);
```

35.3.2. Description

The llapi_file_create() call is equivalent to the llapi_file_open call with *flags* equal to O_CREAT|O_WRONLY and *mode* equal to 0644, followed by file close.

llapi_file_open() opens a file with a given name on a Lustre file system.

Option	Description
flags	Can be a combination of O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_NOCTTY,

Option	Description
	O_TRUNC, O_APPEND, O_NONBLOCK, O_SYNC, FASYNC, O_DIRECT, O_LARGEFILE, O_DIRECTORY, O_NOFOLLOW, O_NOATIME.
mode	Specifies the permission bits to be used for a new file when O_CREAT is used.
stripe_size	Specifies stripe size (in bytes). Should be multiple of 64 KB, not exceeding 4 GB.
stripe_offset	Specifies an OST index from which the file should start. The default value is -1.
stripe_count	Specifies the number of OSTs to stripe the file across. The default value is -1.
stripe_pattern	Specifies the striping pattern. In this release of the Lustre software, only LOV_PATTERN_RAID0 is available. The default value is 0.

35.3.3. Return Values

llapi_file_open() and llapi_file_create() return:

>=0 On success, for llapi_file_open the return value is a file descriptor

<0 On failure, the absolute value is an error code

35.3.4. Errors

Errors	Description
EINVAL	stripe_size or stripe_offset or stripe_count or stripe_pattern is invalid.
EEXIST	Striping information has already been set and cannot be altered; name already exists.
EALREADY	Striping information has already been set and cannot be altered
ENOTTY	name may not point to a Lustre file system.

35.3.5. Example

```
#include <stdio.h>
#include <lustre/lustreapi.h>

int main(int argc, char *argv[])
{
 int rc;
 if (argc != 2)
  return -1;
 rc = llapi_file_create(argv[1], 1048576, 0, 2, LOV_PATTERN_RAID0);
 if (rc < 0) {
```

```
  fprintf(stderr, "file creation has failed, %s\n",         strerror(-rc));
  return -1;
 }
 printf("%s with stripe size 1048576, striped across 2 OSTs,"
   " has been created!\n", argv[1]);
 return 0;
}
```

35.4. llapi_quotactl

Use llapi_quotactl to manipulate disk quotas on a Lustre file system.

35.4.1. Synopsis

```
#include <lustre/lustreapi.h>
int llapi_quotactl(char" " *mnt," " struct if_quotactl" " *qctl)

struct if_quotactl {
        __u32                   qc_cmd;
        __u32                   qc_type;
        __u32                   qc_id;
        __u32                   qc_stat;
        struct obd_dqinfo       qc_dqinfo;
        struct obd_dqblk        qc_dqblk;
        char                    obd_type[16];
        struct obd_uuid         obd_uuid;
};
struct obd_dqblk {
        __u64 dqb_bhardlimit;
        __u64 dqb_bsoftlimit;
        __u64 dqb_curspace;
        __u64 dqb_ihardlimit;
        __u64 dqb_isoftlimit;
        __u64 dqb_curinodes;
        __u64 dqb_btime;
        __u64 dqb_itime;
        __u32 dqb_valid;
        __u32 padding;
};
struct obd_dqinfo {
        __u64 dqi_bgrace;
        __u64 dqi_igrace;
        __u32 dqi_flags;
        __u32 dqi_valid;
};
struct obd_uuid {
        char uuid[40];
};
```

35.4.2. Description

The llapi_quotactl() command manipulates disk quotas on a Lustre file system mount. qc_cmd indicates a command to be applied to UID qc_id or GID qc_id.

Option	Description
LUSTRE_Q_QUOTAON	Turns on quotas for a Lustre file system. Deprecated as of 2.4.0. *qc_type* is USRQUOTA, GRPQUOTA or UGQUOTA (both user and group quota). The quota files must exist. They are normally created with the llapi_quotacheck call. This call is restricted to the super user privilege. As of 2.4.0, quota is now enabled on a per file system basis via lctl conf_param (see Section 21.2, “Enabling Disk Quotas”) on the MGS node and quotacheck isn't needed any more.
LUSTRE_Q_QUOTAOFF	Turns off quotas for a Lustre file system. Deprecated as of 2.4.0. *qc_type* is USRQUOTA, GRPQUOTA or UGQUOTA (both user and group quota). This call is restricted to the super user privilege. As of 2.4.0, quota is disabled via lctl conf_param (see Section 21.2, “Enabling Disk Quotas”).
LUSTRE_Q_GETQUOTA	Gets disk quota limits and current usage for user or group *qc_id*. *qc_type* is USRQUOTA or GRPQUOTA. *uuid* may be filled with OBD UUID string to query quota information from a specific node. *dqb_valid* may be set nonzero to query information only from MDS. If *uuid* is an empty string and *dqb_valid* is zero then cluster-wide limits and usage are returned. On return, *obd_dqblk* contains the requested information (block limits unit is kilobyte). Quotas must be turned on before using this command.
LUSTRE_Q_SETQUOTA	Sets disk quota limits for user or group *qc_id*. *qc_type* is USRQUOTA or GRPQUOTA. *dqb_valid* must be set to QIF_ILIMITS, QIF_BLIMITS or QIF_LIMITS (both inode limits and block limits) dependent on updating limits. *obd_dqblk* must be filled with limits values (as set in *dqb_valid*, block limits unit is kilobyte). Quotas must be turned on before using this command.
LUSTRE_Q_GETINFO	Gets information about quotas. *qc_type* is either USRQUOTA or GRPQUOTA. On return, *dqi_igrace* is inode grace time (in seconds), *dqi_bgrace* is block grace time (in seconds), *dqi_flags* is not used by the current release of the Lustre software.
LUSTRE_Q_SETINFO	Sets quota information (like grace times). *qc_type* is either USRQUOTA or GRPQUOTA. *dqi_igrace* is inode grace time (in seconds), *dqi_bgrace* is block grace time (in seconds), *dqi_flags* is not used by the current release of the Lustre software and must be zeroed.

35.4.3. Return Values

llapi_quotactl() returns:

0 On success

-1 On failure and sets error number (errno) to indicate the error

35.4.4. Errors

llapi_quotactl errors are described below.

Errors	Description
EFAULT	*qctl* is invalid.
ENOSYS	Kernel or Lustre modules have not been compiled with the QUOTA option.
ENOMEM	Insufficient memory to complete operation.
ENOTTY	*qc_cmd* is invalid.
EBUSY	Cannot process during quotacheck.
ENOENT	*uuid* does not correspond to OBD or *mnt* does not exist.
EPERM	The call is privileged and the caller is not the super user.
ESRCH	No disk quota is found for the indicated user. Quotas have not been turned on for this file system.

35.5. llapi_path2fid

Use llapi_path2fid to get the FID from the pathname.

35.5.1. Synopsis

```
#include <lustre/lustreapi.h>

int llapi_path2fid(const char *path, unsigned long long *seq, unsigned long *oid,
```

35.5.2. Description

The llapi_path2fid function returns the FID (sequence : object ID : version) for the pathname.

35.5.3. Return Values

llapi_path2fid returns:

0 On success

non-zero value On failure

35.6. Example Using the llapi Library

Use llapi_file_create to set Lustre software properties for a new file. For a synopsis and description of llapi_file_create and examples of how to use it, see Chapter 36, *Configuration Files and Module Parameters*.

You can set striping from inside programs like `ioctl`. To compile the sample program, you need to install the Lustre client source RPM.

A simple C program to demonstrate striping API - libtest.c

```
/* -*- mode: c; c-basic-offset: 8; indent-tabs-mode: nil; -*-
 * vim:expandtab:shiftwidth=8:tabstop=8:
 *
 * lustredemo - a simple example of lustreapi functions
 */
#include <stdio.h>
#include <fcntl.h>
#include <dirent.h>
#include <errno.h>
#include <stdlib.h>
#include <lustre/lustreapi.h>
#define MAX_OSTS 1024
#define LOV_EA_SIZE(lum, num) (sizeof(*lum) + num * sizeof(*lum->lmm_objects))
#define LOV_EA_MAX(lum) LOV_EA_SIZE(lum, MAX_OSTS)

/*
 * This program provides crude examples of using the lustreapi API functions
 */
/* Change these definitions to suit */

#define TESTDIR "/tmp"           /* Results directory */
#define TESTFILE "lustre_dummy"  /* Name for the file we create/destroy */
#define FILESIZE 262144                  /* Size of the file in words */
#define DUMWORD "DEADBEEF"       /* Dummy word used to fill files */
#define MY_STRIPE_WIDTH 2                /* Set this to the number of OST requir
#define MY_LUSTRE_DIR "/mnt/lustre/ftest"

int close_file(int fd)
{
        if (close(fd) < 0) {
                fprintf(stderr, "File close failed: %d (%s)\n", errno, strerror(er
                return -1;
        }
        return 0;
}

int write_file(int fd)
{
        char *stng =  DUMWORD;
        int cnt = 0;

        for( cnt = 0; cnt < FILESIZE; cnt++) {
                write(fd, stng, sizeof(stng));
        }
        return 0;
}
/* Open a file, set a specific stripe count, size and starting OST
 *    Adjust the parameters to suit */
```

```
int open_stripe_file()
{
        char *tfile = TESTFILE;
        int stripe_size = 65536;    /* System default is 4M */
        int stripe_offset = -1;     /* Start at default */
        int stripe_count = MY_STRIPE_WIDTH;  /*Single stripe for this demo*/
        int stripe_pattern = 0;     /* only RAID 0 at this time */
        int rc, fd;

        rc = llapi_file_create(tfile,
                        stripe_size,stripe_offset,stripe_count,stripe_pattern);
        /* result code is inverted, we may return -EINVAL or an ioctl error.
         * We borrow an error message from sanity.c
         */
        if (rc) {
                fprintf(stderr,"llapi_file_create failed: %d (%s) \n", rc, strerro
                return -1;
        }
        /* llapi_file_create closes the file descriptor, we must re-open */
        fd = open(tfile, O_CREAT | O_RDWR | O_LOV_DELAY_CREATE, 0644);
        if (fd < 0) {
                fprintf(stderr, "Can't open %s file: %d (%s)\n", tfile, errno, str
                return -1;
        }
        return fd;
}

/* output a list of uuids for this file */
int get_my_uuids(int fd)
{
        struct obd_uuid uuids[1024], *uuidp;        /* Output var */
        int obdcount = 1024;
        int rc,i;

        rc = llapi_lov_get_uuids(fd, uuids, &obdcount);
        if (rc != 0) {
                fprintf(stderr, "get uuids failed: %d (%s)\n",errno, strerror(errn
        }
        printf("This file system has %d obds\n", obdcount);
        for (i = 0, uuidp = uuids; i < obdcount; i++, uuidp++) {
                printf("UUID %d is %s\n",i, uuidp->uuid);
        }
        return 0;
}

/* Print out some LOV attributes. List our objects */
int get_file_info(char *path)
{

        struct lov_user_md *lump;
        int rc;
        int i;

        lump = malloc(LOV_EA_MAX(lump));
```

```
        if (lump == NULL) {
                return -1;
        }

        rc = llapi_file_get_stripe(path, lump);

        if (rc != 0) {
                fprintf(stderr, "get_stripe failed: %d (%s)\n",errno, strerror(err
                return -1;
        }

        printf("Lov magic %u\n", lump->lmm_magic);
        printf("Lov pattern %u\n", lump->lmm_pattern);
        printf("Lov object id %llu\n", lump->lmm_object_id);
        printf("Lov stripe size %u\n", lump->lmm_stripe_size);
        printf("Lov stripe count %hu\n", lump->lmm_stripe_count);
        printf("Lov stripe offset %u\n", lump->lmm_stripe_offset);
        for (i = 0; i < lump->lmm_stripe_count; i++) {
                printf("Object index %d Objid %llu\n", lump->lmm_objects[i].l_ost_
        }

        free(lump);
        return rc;

}

/* Ping all OSTs that belong to this filesystem */
int ping_osts()
{
        DIR *dir;
        struct dirent *d;
        char osc_dir[100];
        int rc;

        sprintf(osc_dir, "/proc/fs/lustre/osc");
        dir = opendir(osc_dir);
        if (dir == NULL) {
                printf("Can't open dir\n");
                return -1;
        }
        while((d = readdir(dir)) != NULL) {
                if ( d->d_type == DT_DIR ) {
                        if (! strncmp(d->d_name, "OSC", 3)) {
                                printf("Pinging OSC %s ", d->d_name);
                                rc = llapi_ping("osc", d->d_name);
                                if (rc) {
                                        printf("  bad\n");
                                } else {
                                        printf("  good\n");
                                }
                        }
                }
        }
        return 0;
```

```
}

int main()
{
        int file;
        int rc;
        char filename[100];
        char sys_cmd[100];

        sprintf(filename, "%s/%s",MY_LUSTRE_DIR, TESTFILE);

        printf("Open a file with striping\n");
        file = open_stripe_file();
        if ( file < 0 ) {
                printf("Exiting\n");
                exit(1);
        }
        printf("Getting uuid list\n");
        rc = get_my_uuids(file);
        printf("Write to the file\n");
        rc = write_file(file);
        rc = close_file(file);
        printf("Listing LOV data\n");
        rc = get_file_info(filename);
        printf("Ping our OSTs\n");
        rc = ping_osts();

        /* the results should match lfs getstripe */
        printf("Confirming our results with lfs getstripe\n");
        sprintf(sys_cmd, "/usr/bin/lfs getstripe %s/%s", MY_LUSTRE_DIR, TESTFILE);
        system(sys_cmd);

        printf("All done\n");
        exit(rc);
}
```

Makefile for sample application:

```
gcc -g -O2 -Wall -o lustredemo libtest.c -llustreapi
clean:
rm -f core lustredemo *.o
run:
make
rm -f /mnt/lustre/ftest/lustredemo
rm -f /mnt/lustre/ftest/lustre_dummy
cp lustredemo /mnt/lustre/ftest/
```

35.6.1. See Also

- Section 35.1, “ llapi_file_create ”
- Section 35.2, “llapi_file_get_stripe”

- Section 35.3, “ `llapi_file_open` ”
- Section 35.4, “ `llapi_quotactl` ”

Chapter 36. Configuration Files and Module Parameters

This section describes configuration files and module parameters and includes the following sections:

- Section 36.1, " Introduction"
- Section 36.2, " Module Options"

36.1. Introduction

LNET network hardware and routing are now configured via module parameters. Parameters should be specified in the `/etc/modprobe.d/lustre.conf`file, for example:

```
options lnet networks=tcp0(eth2)
```

The above option specifies that this node should use the TCP protocol on the eth2 network interface.

Module parameters are read when the module is first loaded. Type-specific LND modules (for instance, `ksocklnd`) are loaded automatically by the LNET module when LNET starts (typically upon `modprobe ptlrpc`).

LNET configuration parameters can be viewed under `/sys/module/lnet/parameters/`, and LND-specific parameters under the name of the corresponding LND, for example `/sys/module/ksocklnd/parameters/` for the socklnd (TCP) LND.

For the following parameters, default option settings are shown in parenthesis. Changes to parameters marked with a W affect running systems. Unmarked parameters can only be set when LNET loads for the first time. Changes to parameters marked with `Wc` only have effect when connections are established (existing connections are not affected by these changes.)

36.2. Module Options

- With routed or other multi-network configurations, use `ip2nets` rather than networks, so all nodes can use the same configuration.
- For a routed network, use the same 'routes' configuration everywhere. Nodes specified as routers automatically enable forwarding and any routes that are not relevant to a particular node are ignored. Keep a common configuration to guarantee that all nodes have consistent routing tables.
- A separate `lustre.conf` file makes distributing the configuration much easier.
- If you set `config_on_load=1`, LNET starts at `modprobe` time rather than waiting for the Lustre file system to start. This ensures routers start working at module load time.

```
# lctl
# lctl> net down
```

- Remember the `lctl ping {nid}` command - it is a handy way to check your LNET configuration.

36.2.1. LNET Options

This section describes LNET options.

36.2.1.1. Network Topology

Network topology module parameters determine which networks a node should join, whether it should route between these networks, and how it communicates with non-local networks.

Here is a list of various networks and the supported software stacks:

Network	Software Stack
o2ib	OFED Version 2
mx	Myrinet MX
gm	Myrinet GM-2

Note

The Lustre software ignores the loopback interface (`lo0`), but the Lustre file system uses any IP addresses aliased to the loopback (by default). When in doubt, explicitly specify networks.

`ip2nets` ("") is a string that lists globally-available networks, each with a set of IP address ranges. LNET determines the locally-available networks from this list by matching the IP address ranges with the local IPs of a node. The purpose of this option is to be able to use the same `modules.conf` file across a variety of nodes on different networks. The string has the following syntax.

```
<ip2nets> :== <net-match> [ <comment> ] { <net-sep> <net-match> }
<net-match> :== [ <w> ] <net-spec> <w> <ip-range> { <w> <ip-range> }
[ <w> ]
<net-spec> :== <network> [ "(" <interface-list> ")" ]
<network> :== <nettype> [ <number> ]
<nettype> :== "tcp" | "elan" | "openib" | ...
<iface-list> :== <interface> [ "," <iface-list> ]
<ip-range> :== <r-expr> "." <r-expr> "." <r-expr> "." <r-expr>
<r-expr> :== <number> | "*" | "[" <r-list> "]"
<r-list> :== <range> [ "," <r-list> ]
<range> :== <number> [ "-" <number> [ "/" <number> ] ]
<comment :== "#" { <non-net-sep-chars> }
<net-sep> :== ";" | "\n"
<w> :== <whitespace-chars> { <whitespace-chars> }
```

`<net-spec>` contains enough information to uniquely identify the network and load an appropriate LND. The LND determines the missing "address-within-network" part of the NID based on the interfaces it can use.

`<iface-list>` specifies which hardware interface the network can use. If omitted, all interfaces are used. LNDs that do not support the `<iface-list>` syntax cannot be configured to use particular interfaces and just use what is there. Only a single instance of these LNDs can exist on a node at any time, and `<iface-list>` must be omitted.

`<net-match>` entries are scanned in the order declared to see if one of the node's IP addresses matches one of the `<ip-range>` expressions. If there is a match, `<net-spec>` specifies the network to

instantiate. Note that it is the first match for a particular network that counts. This can be used to simplify the match expression for the general case by placing it after the special cases. For example:

```
ip2nets="tcp(eth1,eth2) 134.32.1.[4-10/2]; tcp(eth1) *.*.*.*"
```

4 nodes on the 134.32.1.* network have 2 interfaces (134.32.1.{4,6,8,10}) but all the rest have 1.

```
ip2nets="vib 192.168.0.*; tcp(eth2) 192.168.0.[1,7,4,12]"
```

This describes an IB cluster on 192.168.0.*. Four of these nodes also have IP interfaces; these four could be used as routers.

Note that match-all expressions (For instance, `*.*.*.*`) effectively mask all other

`<net-match>` entries specified after them. They should be used with caution.

Here is a more complicated situation, the route parameter is explained below. We have:

- Two TCP subnets
- One Elan subnet
- One machine set up as a router, with both TCP and Elan interfaces
- IP over Elan configured, but only IP will be used to label the nodes.

```
options lnet ip2nets=â€tcp 198.129.135.* 192.128.88.98; \
        elan 198.128.88.98 198.129.135.3; \
        routes='cp 1022@elan # Elan NID of router; \
        elan  198.128.88.98@tcp # TCP NID of router  '
```

36.2.1.2. networks ("tcp")

This is an alternative to "`ip2nets`" which can be used to specify the networks to be instantiated explicitly. The syntax is a simple comma separated list of `<net-spec>`s (see above). The default is only used if neither 'ip2nets' nor 'networks' is specified.

36.2.1.3. routes ("")

This is a string that lists networks and the NIDs of routers that forward to them.

It has the following syntax (`<w>` is one or more whitespace characters):

```
<routes> :== <route>{ ; <route> }
<route> :== [<net>[<w><hopcount>]<w><nid>[:<priority>]{<w><nid>[:<priority>]}
```

Note: the priority parameter was added in release 2.5.

So a node on the network `tcp1` that needs to go through a router to get to the Elan network:

```
options lnet networks=tcp1 routes="elan 1 192.168.2.2@tcpA"
```

The hopcount and priority numbers are used to help choose the best path between multiply-routed configurations.

A simple but powerful expansion syntax is provided, both for target networks and router NIDs as follows.

```
<expansion> :== "[" <entry> { "," <entry> } "]"
<entry> :== <numeric range> | <non-numeric item>
<numeric range> :== <number> [ "-" <number> [ "/" <number> ] ]
```

The expansion is a list enclosed in square brackets. Numeric items in the list may be a single number, a contiguous range of numbers, or a strided range of numbers. For example, `routes="elan 192.168.1.[22-24]@tcp"` says that network `elan0` is adjacent (hopcount defaults to 1); and is accessible via 3 routers on the `tcp0` network (`192.168.1.22@tcp`, `192.168.1.23@tcp` and `192.168.1.24@tcp`).

`routes="[tcp,vib] 2 [8-14/2]@elan"` says that 2 networks (`tcp0` and `vib0`) are accessible through 4 routers (`8@elan`, `10@elan`, `12@elan` and `14@elan`). The hopcount of 2 means that traffic to both these networks will be traversed 2 routers - first one of the routers specified in this entry, then one more.

Duplicate entries, entries that route to a local network, and entries that specify routers on a non-local network are ignored.

Prior to release 2.5, a conflict between equivalent entries was resolved in favor of the route with the shorter hopcount. The hopcount, if omitted, defaults to 1 (the remote network is adjacent)..

Introduced in Lustre 2.5

Since 2.5, equivalent entries are resolved in favor of the route with the lowest priority number or shorter hopcount if the priorities are equal. The priority, if omitted, defaults to 0. The hopcount, if omitted, defaults to 1 (the remote network is adjacent).

It is an error to specify routes to the same destination with routers on different local networks.

If the target network string contains no expansions, then the hopcount defaults to 1 and may be omitted (that is, the remote network is adjacent). In practice, this is true for most multi-network configurations. It is an error to specify an inconsistent hop count for a given target network. This is why an explicit hopcount is required if the target network string specifies more than one network.

36.2.1.4. forwarding ("")

This is a string that can be set either to "`enabled`" or "`disabled`" for explicit control of whether this node should act as a router, forwarding communications between all local networks.

A standalone router can be started by simply starting LNET ('`modprobe ptlrpc`') with appropriate network topology options.

Variable	Description
`acceptor`	The acceptor is a TCP/IP service that some LNDs use to establish communications. If a local network requires it and it has not been disabled, the acceptor listens on a single port for connection requests that it redirects to the appropriate local network. The acceptor is part of the LNET module and configured by the following options: • `secure` - Accept connections only from reserved TCP ports (below 1023). • `all` - Accept connections from any TCP port.

Variable	Description
	Note This is required for liblustre clients to allow connections on non-privileged ports. • `none` - Do not run the acceptor.
`accept_port` `(988)`	Port number on which the acceptor should listen for connection requests. All nodes in a site configuration that require an acceptor must use the same port.
`accept_backlog` `(127)`	Maximum length that the queue of pending connections may grow to (see listen(2)).
`accept_timeout` `(5, W)`	Maximum time in seconds the acceptor is allowed to block while communicating with a peer.
`accept_proto_version`	Version of the acceptor protocol that should be used by outgoing connection requests. It defaults to the most recent acceptor protocol version, but it may be set to the previous version to allow the node to initiate connections with nodes that only understand that version of the acceptor protocol. The acceptor can, with some restrictions, handle either version (that is, it can accept connections from both 'old' and 'new' peers). For the current version of the acceptor protocol (version 1), the acceptor is compatible with old peers if it is only required by a single local network.

36.2.1.5. `rnet_htable_size`

Introduced in Lustre 2.3

`rnet_htable_size` is an integer that indicates how many remote networks the internal LNet hash table is configured to handle. `rnet_htable_size` is used for optimizing the hash table size and does not put a limit on how many remote networks you can have. The default hash table size when this parameter is not specified is: 128.

36.2.2. `SOCKLND` Kernel TCP/IP LND

The `SOCKLND` kernel TCP/IP LND (`socklnd`) is connection-based and uses the acceptor to establish communications via sockets with its peers.

It supports multiple instances and load balances dynamically over multiple interfaces. If no interfaces are specified by the `ip2nets` or networks module parameter, all non-loopback IP interfaces are used. The address-within-network is determined by the address of the first IP interface an instance of the `socklnd` encounters.

Consider a node on the 'edge' of an InfiniBand network, with a low-bandwidth management Ethernet (`eth0`), IP over IB configured (`ipoib0`), and a pair of GigE NICs (`eth1`,`eth2`) providing off-cluster

connectivity. This node should be configured with '`networks=vib,tcp(eth1,eth2)`' to ensure that the `socklnd` ignores the management Ethernet and IPoIB.

Variable	Description

`timeout`
 	
`(50,W)`
	Time (in seconds) that communications may be stalled before the LND completes them with failure.

`nconnds`
 	
`(4)`
	Sets the number of connection daemons.

`min_reconnectms`
 	
`(1000,W)`
	Minimum connection retry interval (in milliseconds). After a failed connection attempt, this is the time that must elapse before the first retry. As connections attempts fail, this time is doubled on each successive retry up to a maximum of '`max_reconnectms`'.

`max_reconnectms`
 	
`(6000,W)`
	Maximum connection retry interval (in milliseconds).

`eager_ack`
 	
`(0 on linux,`
 	
`1 on darwin,W)`
	Boolean that determines whether the `socklnd` should attempt to flush sends on message boundaries.

`typed_conns`
 	
`(1,Wc)`
	Boolean that determines whether the `socklnd` should use different sockets for different types of messages. When clear, all communication with a particular peer takes place on the same socket. Otherwise, separate sockets are used for bulk sends, bulk receives and everything else.

`min_bulk`
 	
`(1024,W)`
	Determines when a message is considered "bulk".

`tx_buffer_size, rx_buffer_size`
 	
`(8388608,Wc)`
	Socket buffer sizes. Setting this option to zero (0), allows the system to auto-tune buffer sizes.
	
	Warning
	
	Be very careful changing this value as improper sizing can harm performance.

`nagle`
 	
`(0,Wc)`
	Boolean that determines if `nagle` should be enabled. It should never be set in production systems.

`keepalive_idle`
 	
`(30,Wc)`
	Time (in seconds) that a socket can remain idle before a keepalive probe is sent. Setting this value to zero (0) disables keepalives.

`keepalive_intvl`
 	
`(2,Wc)`
	Time (in seconds) to repeat unanswered keepalive probes. Setting this value to zero (0) disables keepalives.

Variable	Description
keepalive_count (10,Wc)	Number of unanswered keepalive probes before pronouncing socket (hence peer) death.
enable_irq_affinity (0,Wc)	Boolean that determines whether to enable IRQ affinity. The default is zero (0). When set, socklnd attempts to maximize performance by handling device interrupts and data movement for particular (hardware) interfaces on particular CPUs. This option is not available on all platforms. This option requires an SMP system to exist and produces best performance with multiple NICs. Systems with multiple CPUs and a single NIC may see increase in the performance with this parameter disabled.
zc_min_frag (2048,W)	Determines the minimum message fragment that should be considered for zero-copy sends. Increasing it above the platform's PAGE_SIZE disables all zero copy sends. This option is not available on all platforms.

36.2.3. Portals LND Linux (ptllnd)

The Portals LND Linux (ptllnd) can be used as a interface layer to communicate with Sandia Portals networking devices. This version is intended to work on Cray XT3 Linux nodes that use Cray Portals as a network transport.

Message Buffers

When ptllnd starts up, it allocates and posts sufficient message buffers to allow all expected peers (set by concurrent_peers) to send one unsolicited message. The first message that a peer actually sends is (so-called) "HELLO" message, used to negotiate how much additional buffering to setup (typically 8 messages). If 10000 peers actually exist, then enough buffers are posted for 80000 messages.

The maximum message size is set by the max_msg_size module parameter (default value is 512). This parameter sets the bulk transfer breakpoint. Below this breakpoint, payload data is sent in the message itself. Above this breakpoint, a buffer descriptor is sent and the receiver gets the actual payload.

The buffer size is set by the rxb_npages module parameter (default value is 1). The default conservatively avoids allocation problems due to kernel memory fragmentation. However, increasing this value to 2 is probably not risky.

The ptllnd also keeps an additional rxb_nspare buffers (default value is 8) posted to account for full buffers being handled.

Assuming a 4K page size with 10000 peers, 1258 buffers can be expected to be posted at startup, increasing to a maximum of 10008 as peers that are actually connected. By doubling rxb_npages halving max_msg_size, this number can be reduced by a factor of 4.

ME/MD Queue Length

The ptllnd uses a single portal set by the portal module parameter (default value of 9) for both message and bulk buffers. Message buffers are always attached with PTL_INS_AFTER and match anything sent

with "message" matchbits. Bulk buffers are always attached with `PTL_INS_BEFORE` and match only specific matchbits for that particular bulk transfer.

This scheme assumes that the majority of ME/MDs posted are for "message" buffers, and that the overhead of searching through the preceding "bulk" buffers is acceptable. Since the number of "bulk" buffers posted at any time is also dependent on the bulk transfer breakpoint set by `max_msg_size`, this seems like an issue worth measuring at scale.

TX Descriptors

The `ptllnd` has a pool of so-called "tx descriptors", which it uses not only for outgoing messages, but also to hold state for bulk transfers requested by incoming messages. This pool should scale with the total number of peers.

To enable the building of the Portals LND (`ptllnd.ko`) configure with this option:

`./configure --with-portals=`*`/path/to/portals/headers`*

Variable	Description

`ntx`
 	
`(256)`
	Total number of messaging descriptors.

`concurrent_peers`
 	
`(1152)`
	Maximum number of concurrent peers. Peers that attempt to connect beyond the maximum are not allowed.

`peer_hash_table_size`
 	
`(101)`
	Number of hash table slots for the peers. This number should scale with `concurrent_peers`. The size of the peer hash table is set by the module parameter `peer_hash_table_size` which defaults to a value of 101. This number should be prime to ensure the peer hash table is populated evenly. It is advisable to increase this value to 1001 for ~10000 peers.

`cksum`
 	
`(0)`
	Set to non-zero to enable message (not RDMA) checksums for outgoing packets. Incoming packets are always check-summed if necessary, independent of this value.

`timeout`
 	
`(50)`
	Amount of time (in seconds) that a request can linger in a peers-active queue before the peer is considered dead.

`portal`
 	
`(9)`
	Portal ID to use for the `ptllnd` traffic.

`rxb_npages`
 	
`(64 * #cpus)`
	Number of pages in an RX buffer.

`credits`
 	
`(128)`
	Maximum total number of concurrent sends that are outstanding to a single peer at a given time.

`peercredits`
 	
`(8)`
	Maximum number of concurrent sends that are outstanding to a single peer at a given time.

Variable	Description
max_msg_size (512)	Maximum immediate message size. This MUST be the same on all nodes in a cluster. A peer that connects with a different max_msg_size value will be rejected.

36.2.4. MX LND

MXLND supports a number of load-time parameters using Linux's module parameter system. The following variables are available:

Variable	Description
n_waitd	Number of completion daemons.
max_peers	Maximum number of peers that may connect.
cksum	Enables small message (below 4 KB) checksums if set to a non-zero value.
ntx	Number of total tx message descriptors.
credits	Number of concurrent sends to a single peer.
board	Index value of the Myrinet board (NIC).
ep_id	MX endpoint ID.
polling	Use zero (0) to block (wait). A value greater than 0 will poll that many times before blocking.
hosts	IP-to-hostname resolution file.

Of the described variables, only hosts is required. It must be the absolute path to the MXLND hosts file.

For example:

```
options kmxlnd hosts=/etc/hosts.mxlnd
```

The file format for the hosts file is:

```
IP  HOST  BOARD  EP_ID
```

The values must be space and/or tab separated where:

IP is a valid IPv4 address

HOST is the name returned by `hostname` on that machine

BOARD is the index of the Myricom NIC (0 for the first card, etc.)

EP_ID is the MX endpoint ID

To obtain the optimal performance for your platform, you may want to vary the remaining options.

n_waitd(1) sets the number of threads that process completed MX requests (sends and receives).

max_peers(1024) tells MXLND the upper limit of machines that it will need to communicate with. This affects how many receives it will pre-post and each receive will use one page of memory. Ideally, on clients, this value will be equal to the total number of Lustre servers (MDS and OSS). On servers, it needs

to equal the total number of machines in the storage system. cksum (0) turns on small message checksums. It can be used to aid in troubleshooting. MX also provides an optional checksumming feature which can check all messages (large and small). For details, see the MX README.

`ntx(256)` is the number of total sends in flight from this machine. In actuality, MXLND reserves half of them for connect messages so make this value twice as large as you want for the total number of sends in flight.

`credits(8)` is the number of in-flight messages for a specific peer. This is part of the flow-control system in provided by the Lustre software. Increasing this value may improve performance but it requires more memory because each message requires at least one page.

`board(0)` is the index of the Myricom NIC. Hosts can have multiple Myricom NICs and this identifies which one MXLND should use. This value must match the board value in your MXLND hosts file for this host.

`ep_id(3)` is the MX endpoint ID. Each process that uses MX is required to have at least one MX endpoint to access the MX library and NIC. The ID is a simple index starting at zero (0). This value must match the endpoint ID value in your MXLND hosts file for this host.

`polling(0)` determines whether this host will poll or block for MX request completions. A value of 0 blocks and any positive value will poll that many times before blocking. Since polling increases CPU usage, we suggest that you set this to zero (0) on the client and experiment with different values for servers.

Chapter 37. System Configuration Utilities

This chapter includes system configuration utilities and includes the following sections:

37.1. e2scan

The e2scan utility is an ext2 file system-modified inode scan program. The e2scan program uses libext2fs to find inodes with ctime or mtime newer than a given time and prints out their pathname. Use e2scan to efficiently generate lists of files that have been modified. The e2scan tool is included in the e2fsprogs package, located at:

http://downloads.hpdd.intel.com/public/e2fsprogs/latest/

37.1.1. Synopsis

```
e2scan [options] [-f file] block_device
```

37.1.2. Description

When invoked, the e2scan utility iterates all inodes on the block device, finds modified inodes, and prints their inode numbers. A similar iterator, using libext2fs(5), builds a table (called parent database) which lists the parent node for each inode. With a lookup function, you can reconstruct modified pathnames from root.

37.1.3. Options

Option	Description
`-b` *`inode buffer blocks`*	Sets the readahead inode blocks to get excellent performance when scanning the block device.
`-o` *`output file`*	If an output file is specified, modified pathnames are written to this file. Otherwise, modified parameters are written to stdout.
`-t` *`inode`*`|` *`pathname`*	Sets the e2scan type if type is inode. The e2scan utility prints modified inode numbers to stdout. By default, the type is set as pathname. The e2scan utility lists modified pathnames based on modified inode numbers.
`-u`	Rebuilds the parent database from scratch. Otherwise, the current parent database is used.

37.2. l_getidentity

The l_getidentity utility handles Lustre user / group cache upcall.

37.2.1. Synopsis

```
l_getidentity ${FSNAME}-MDT{xxxx} {uid}
```

37.2.2. Description

The group upcall file contains the path to an executable file that is invoked to resolve a numeric UID to a group membership list. This utility opens `/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_info` and writes the related `identity_downcall_data` structure (see Section 34.1.4, “Data Structures”.) The data is persisted with `lctl set_param mdt.${FSNAME}-MDT{xxxx}.identity_info`.

The l_getidentity utility is the reference implementation of the user or group cache upcall.

37.2.3. Options

Option	Description
`${FSNAME}-MDT{xxxx}`	Metadata server target name
`uid`	User identifier

37.2.4. Files

The l_getidentity files are located at:

```
/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_upcall
```

37.3. lctl

The lctl utility is used for root control and configuration. With lctl you can directly control Lustre via an ioctl interface, allowing various configuration, maintenance and debugging features to be accessed.

37.3.1. Synopsis

```
lctl [--device devno] command [args]
```

37.3.2. Description

The lctl utility can be invoked in interactive mode by issuing the lctl command. After that, commands are issued as shown below. The most common lctl commands are:

```
dl
dk
device
network up|down
list_nids
ping nidhelp
quit
```

For a complete list of available commands, type `help` at the `lctl` prompt. To get basic help on command meaning and syntax, type `help` *`command`*. Command completion is activated with the TAB key (depending on compile options), and command history is available via the up- and down-arrow keys.

For non-interactive use, use the second invocation, which runs the command after connecting to the device.

37.3.3. Setting Parameters with lctl

Lustre parameters are not always accessible using the procfs interface, as it is platform-specific. As a solution, lctl {get,set}_param has been introduced as a platform-independent interface to the Lustre tunables. Avoid direct references to /proc/{fs,sys}/{lustre,lnet}. For future portability, use lctl {get,set}_param .

When the file system is running, use the `lctl set_param` command on the affected node(s) to *temporarily* set parameters (mapping to items in /proc/{fs,sys}/{lnet,lustre}). The `lctl set_param` command uses this syntax:

```
lctl set_param [-n] [-P] [-d] obdtype.obdname.property=value
```

For example:

```
mds# lctl set_param mdt.testfs-MDT0000.identity_upcall=NONE
```

Introduced in Lustre 2.5

Use `-P` option to set parameters permanently. Option `-d` deletes permanent parameters. For example:

```
mgs# lctl set_param -P mdt.testfs-MDT0000.identity_upcall=NONE
```

```
mgs# lctl set_param -P -d mdt.testfs-MDT0000.identity_upcall
```

Many permanent parameters can be set with `lctl conf_param`. In general, `lctl conf_param` can be used to specify any OBD device parameter settable in a /proc/fs/lustre file. The `lctl conf_param` command must be run on the MGS node, and uses this syntax:

obd|fsname.obdtype.property=*value*)

For example:

```
mgs# lctl conf_param testfs-MDT0000.mdt.identity_upcall=NONE
$ lctl conf_param testfs.llite.max_read_ahead_mb=16
```

Caution

The `lctl conf_param` command *permanently* sets parameters in the file system configuration for all nodes of the specified type.

To get current Lustre parameter settings, use the `lctl get_param` command on the desired node with the same parameter name as `lctl set_param`:

`lctl get_param [-n] obdtype.obdname.parameter`

For example:

```
mds# lctl get_param mdt.testfs-MDT0000.identity_upcall
```

To list Lustre parameters that are available to set, use the `lctl list_param` command, with this syntax:

`lctl list_param [-R] [-F] obdtype.obdname.*`

For example, to list all of the parameters on the MDT:

```
oss# lctl list_param -RF mdt
```

For more information on using lctl to set temporary and permanent parameters, see Section 13.9.3, “Setting Parameters with `lctl`”.

Network Configuration

Option	Description
`network up|down|tcp|elan|myrinet`	Starts or stops LNET, or selects a network type for other `lctl` LNET commands.
`list_nids`	Prints all NIDs on the local node. LNET must be running.
`which_nid` *nidlist*	From a list of NIDs for a remote node, identifies the NID on which interface communication will occur.
`ping` *nid*	Checks LNET connectivity via an LNET ping. This uses the fabric appropriate to the specified NID.
`interface_list`	Prints the network interface information for a given *network* type.
`peer_list`	Prints the known peers for a given *network* type.
`conn_list`	Prints all the connected remote NIDs for a given *network* type.

Option	Description
`active_tx`	This command prints active transmits. It is only used for the Elan *network* type.
`route_list`	Prints the complete routing table.

Device Selection

Option		Description
`device` *devname*		This selects the specified OBD device. All other commands depend on the device being set.
`device_list`		Shows the local Lustre OBDs, a/k/a `dl`.

Device Operations

Option		Description
`list_param [-F|-R]` *parameter [parameter ...]*		Lists the Lustre or LNET parameter name.
	`-F`	Adds '/', '@' or '=' for directories, symlinks and writeable files, respectively.
	`-R`	Recursively lists all parameters under the specified path. If `param_path` is unspecified, all parameters are shown.
`get_param [-n|-N|-F]` *parameter [parameter ...]*		Gets the value of a Lustre or LNET parameter from the specified path.
	`-n`	Prints only the parameter value and not the parameter name.
	`-N`	Prints only matched parameter names and not the values; especially useful when using patterns.
	`-F`	When `-N` is specified, adds '/', '@' or '=' for directories, symlinks and writeable files, respectively.
`set_param [-n]` *parameter=value*		Sets the value of a Lustre or LNET parameter from the specified path.
	`-n`	Disables printing of the key name when printing values.
`conf_param [-d]` *device|fsname parameter=value*		Sets a permanent configuration parameter for any device via the MGS. This command must be run on the MGS node.

Option		Description
		All writeable parameters under `lctl list_param` (e.g. `lctl list_param -F osc.*.* | grep` =) can be permanently set using `lctl conf_param`, but the format is slightly different. For `conf_param`, the device is specified first, then the obdtype. Wildcards are not supported. Additionally, failover nodes may be added (or removed), and some system-wide parameters may be set as well (sys.at_max, sys.at_min, sys.at_extra, sys.at_early_margin, sys.at_history, sys.timeout, sys.ldlm_timeout). For system-wide parameters, *`device`* is ignored. For more information on setting permanent parameters and `lctl conf_param` command examples, see Section 13.9.3.2, “Setting Permanent Parameters” (Setting Permanent Parameters).
	`-d` *`device`*| *`fsname.parameter`*	Deletes a parameter setting (use the default value at the next restart). A null value for *`value`* also deletes the parameter setting.
`activate`		Re-activates an import after the deactivate operation. This setting is only effective until the next restart (see `conf_param`).
`deactivate`		Deactivates an import, in particular meaning do not assign new file stripes to an OSC. Running lctl deactivate on the MDS stops new objects from being allocated on the OST. Running lctl deactivate on Lustre clients causes them to return -EIO when accessing objects on the OST instead of waiting for recovery.
`abort_recovery`		Aborts the recovery process on a re-starting MDT or OST.

Note

Lustre tunables are not always accessible using the procfs interface, as it is platform-specific. As a solution, `lctl {get,set,list}_param` has been introduced as a platform-independent interface to the Lustre tunables. Avoid direct references to `/proc/{fs,sys}/{lustre,lnet}`. For future portability, use `lctl {get,set,list}_param` instead.

Virtual Block Device Operations

Lustre can emulate a virtual block device upon a regular file. This emulation is needed when you are trying to set up a swap space via the file.

Option	Description
`blockdev_attach` *`filename /dev/lloop_device`*	Attaches a regular Lustre file to a block device. If the device node does not exist, `lctl` creates it. It is recommend that a device node is created by `lctl` since the emulator uses a dynamical major number.
`blockdev_detach` *`/dev/lloop_device`*	Detaches the virtual block device.
`blockdev_info` *`/dev/lloop_device`*	Provides information about the Lustre file attached to the device node.

Changelogs

Option	Description
`changelog_register`	Registers a new changelog user for a particular device. Changelog entries are not purged beyond a registered user's set point (see `lfs changelog_clear`).
changelog_deregister *`id`*	Unregisters an existing changelog user. If the user's "clear" record number is the minimum for the device, changelog records are purged until the next minimum.

Debug

Option	Description
`debug_daemon`	Starts and stops the debug daemon, and controls the output filename and size.
`debug_kernel` *`[file]`* `[raw]`	Dumps the kernel debug buffer to stdout or a file.
`debug_file` *`input_file [output_file]`*	Converts the kernel-dumped debug log from binary to plain text format.
`clear`	Clears the kernel debug buffer.
`mark` *`text`*	Inserts marker text in the kernel debug buffer.
`filter` *`subsystem_id|debug_mask`*	Filters kernel debug messages by subsystem or mask.
`show` *`subsystem_id|debug_mask`*	Shows specific types of messages.
`debug_list` *`subsystems|types`*	Lists all subsystem and debug types.

Option	Description
`modules` *path*	Provides GDB-friendly module information.

37.3.4. Options

Use the following options to invoke lctl.

Option	Description
`--device`	Device to be used for the operation (specified by name or number). See device_list.
`--ignore_errors | ignore_errors`	Ignores errors during script processing.

37.3.5. Examples

```
lctl

$ lctl
lctl > dl
   0 UP mgc MGC192.168.0.20@tcp btbb24e3-7deb-2ffa-eab0-44dffe00f692 5
   1 UP ost OSS OSS_uuid 3
   2 UP obdfilter testfs-OST0000 testfs-OST0000_UUID 3
lctl > dk /tmp/log Debug log: 87 lines, 87 kept, 0 dropped.
lctl > quit
```

37.3.6. See Also

- Section 37.14, “ mkfs.lustre”
- Section 37.15, “ mount.lustre”
- Section 37.3, “ lctl”
- Section 33.1, “ `lfs` ”

37.4. ll_decode_filter_fid

The ll_decode_filter_fid utility displays the Lustre object ID and MDT parent FID.

37.4.1. Synopsis

```
ll_decode_filter_fid object_file [object_file ...]
```

37.4.2. Description

The ll_decode_filter_fid utility decodes and prints the Lustre OST object ID, MDT FID, stripe index for the specified OST object(s), which is stored in the "trusted.fid" attribute on each OST object. This is accessible to `ll_decode_filter_fid` when the OST file system is mounted locally as type ldiskfs for maintenance.

The "trusted.fid" extended attribute is stored on each OST object when it is first modified (data written or attributes set), and is not accessed or modified by Lustre after that time.

The OST object ID (objid) is useful in case of OST directory corruption, though normally the ll_recover_lost_found_objs(8) utility is able to reconstruct the entire OST object directory hierarchy. The MDS FID can be useful to determine which MDS inode an OST object is (or was) used by. The stripe index can be used in conjunction with other OST objects to reconstruct the layout of a file even if the MDT inode was lost.

37.4.3. Examples

```
root@oss1# cd /mnt/ost/lost+found
root@oss1# ll_decode_filter_fid #12345[4,5,8]
#123454: objid=690670 seq=0 parent=[0x751c5:0xfce6e605:0x0]
#123455: objid=614725 seq=0 parent=[0x18d11:0xebba84eb:0x1]
#123458: objid=533088 seq=0 parent=[0x21417:0x19734d61:0x0]
```

This shows that the three files in lost+found have decimal object IDs - 690670, 614725, and 533088, respectively. The object sequence number (formerly object group) is 0 for all current OST objects.

The MDT parent inode FIDs are hexadecimal numbers of the form sequence:oid:idx. Since the sequence number is below 0x100000000 in all these cases, the FIDs are in the legacy Inode and Generation In FID (IGIF) namespace and are mapped directly to the MDT inode = seq and generation = oid values; the MDT inodes are 0x751c5, 0x18d11, and 0x21417 respectively. For objects with MDT parent sequence numbers above 0x200000000, this indicates that the FID needs to be mapped via the MDT Object Index (OI) file on the MDT to determine the internal inode number.

The idx field shows the stripe number of this OST object in the Lustre RAID-0 striped file.

37.4.4. See Also

Section 37.5, “ ll_recover_lost_found_objs”

37.5. ll_recover_lost_found_objs

The ll_recover_lost_found_objs utility helps recover Lustre OST objects (file data) from a lost and found directory and return them to their correct locations.

> **Note**
>
> Running the ll_recover_lost_found_objs tool is not strictly necessary to bring an OST back online, it just avoids losing access to objects that were moved to the lost and found directory due to directory corruption.

37.5.1. Synopsis

```
$ ll_recover_lost_found_objs [-hv] -d directory
```

37.5.2. Description

The first time Lustre writes to an object, it saves the MDS inode number and the objid as an extended attribute on the object, so in case of directory corruption of the OST, it is possible to recover the objects.

Running e2fsck fixes the corrupted OST directory, but it puts all of the objects into a lost and found directory, where they are inaccessible to Lustre. Use the ll_recover_lost_found_objs utility to recover all (or at least most) objects from a lost and found directory and return them to the O/0/d* directories.

To use ll_recover_lost_found_objs, mount the file system locally (using the -t ldiskfs command), run the utility and then unmount it again. The OST must not be mounted by Lustre when ll_recover_lost_found_objs is run.

37.5.3. Options

Option	Description
`-h`	Prints a help message
`-v`	Increases verbosity
`-d` *`directory`*	Sets the lost and found directory path

37.5.4. Example

```
ll_recover_lost_found_objs -d /mnt/ost/lost+found
```

37.6. llobdstat

The llobdstat utility displays OST statistics.

37.6.1. Synopsis

```
llobdstat ost_name [interval]
```

37.6.2. Description

The llobdstat utility displays a line of OST statistics for the given ost_name every interval seconds. It should be run directly on an OSS node. Type `CTRL-C` to stop statistics printing.

37.6.3. Example

```
# llobdstat liane-OST0002 1
/usr/bin/llobdstat on /proc/fs/lustre/obdfilter/liane-OST0002/stats
Processor counters run at 2800.189 MHz
Read: 1.21431e+07, Write: 9.93363e+08, create/destroy: 24/1499, stat: 34, p\
unch: 18
[NOTE: cx: create, dx: destroy, st: statfs, pu: punch ]
Timestamp Read-delta ReadRate Write-delta WriteRate
--------------------------------------------------------
1217026053 0.00MB 0.00MB/s 0.00MB 0.00MB/s
1217026054 0.00MB 0.00MB/s 0.00MB 0.00MB/s
1217026055 0.00MB 0.00MB/s 0.00MB 0.00MB/s
1217026056 0.00MB 0.00MB/s 0.00MB 0.00MB/s
1217026057 0.00MB 0.00MB/s 0.00MB 0.00MB/s
1217026058 0.00MB 0.00MB/s 0.00MB 0.00MB/s
1217026059 0.00MB 0.00MB/s 0.00MB 0.00MB/s st:1
```

37.6.4. Files

`/proc/fs/lustre/obdfilter/`*`ostname`*`/stats`

37.7. llog_reader

The llog_reader utility parses Lustre's on-disk configuration logs.

37.7.1. Synopsis

```
llog_reader filename
```

37.7.2. Description

The llog_reader utility parses the binary format of Lustre's on-disk configuration logs. Llog_reader can only read logs; use tunefs.lustre to write to them.

To examine a log file on a stopped Lustre server, mount its backing file system as ldiskfs, then use llog_reader to dump the log file's contents, for example:

```
mount -t ldiskfs /dev/sda /mnt/mgs
llog_reader /mnt/mgs/CONFIGS/tfs-client
```

To examine the same log file on a running Lustre server, use the ldiskfs-enabled debugfs utility (called debug.ldiskfs on some distributions) to extract the file, for example:

```
debugfs -c -R 'dump CONFIGS/tfs-client /tmp/tfs-client' /dev/sda
llog_reader /tmp/tfs-client
```

Caution

Although they are stored in the CONFIGS directory, mountdata files do not use the configuration log format and will confuse the llog_reader utility.

37.7.3. See Also

Section 37.18, “ tunefs.lustre”

37.8. llstat

The llstat utility displays Lustre statistics.

37.8.1. Synopsis

```
llstat [-c] [-g] [-i interval] stats_file
```

37.8.2. Description

The llstat utility displays statistics from any of the Lustre statistics files that share a common format and are updated at `interval` seconds. To stop statistics printing, use `ctrl-c`.

37.8.3. Options

Option	Description
`-c`	Clears the statistics file.
`-i`	Specifies the polling period (in seconds).
`-g`	Specifies graphable output format.
`-h`	Displays help information.
`stats_file`	Specifies either the full path to a statistics file or the shorthand reference, `mds` or `ost`

37.8.4. Example

To monitor /proc/fs/lustre/ost/OSS/ost/stats at 1 second intervals, run;

```
llstat -i 1 ost
```

37.8.5. Files

The llstat files are located at:

```
/proc/fs/lustre/mdt/MDS/*/stats
/proc/fs/lustre/mds/*/exports/*/stats
/proc/fs/lustre/mdc/*/stats
/proc/fs/lustre/ldlm/services/*/stats
/proc/fs/lustre/ldlm/namespaces/*/pool/stats
/proc/fs/lustre/mgs/MGS/exports/*/stats
/proc/fs/lustre/ost/OSS/*/stats
/proc/fs/lustre/osc/*/stats
/proc/fs/lustre/obdfilter/*/exports/*/stats
/proc/fs/lustre/obdfilter/*/stats
/proc/fs/lustre/llite/*/stats
```

37.9. llverdev

The llverdev verifies a block device is functioning properly over its full size.

37.9.1. Synopsis

```
llverdev [-c chunksize] [-f] [-h] [-o offset] [-l] [-p] [-r] [-t timestamp] [-v] [
```

37.9.2. Description

Sometimes kernel drivers or hardware devices have bugs that prevent them from accessing the full device size correctly, or possibly have bad sectors on disk or other problems which prevent proper data storage. There are often defects associated with major system boundaries such as 2^32 bytes, 2^31 sectors, 2^31 blocks, 2^32 blocks, etc.

The llverdev utility writes and verifies a unique test pattern across the entire device to ensure that data is accessible after it was written, and that data written to one part of the disk is not overwriting data on another part of the disk.

It is expected that llverdev will be run on large size devices (TB). It is always better to run llverdev in verbose mode, so that device testing can be easily restarted from the point where it was stopped.

Running a full verification can be time-consuming for very large devices. We recommend starting with a partial verification to ensure that the device is minimally sane before investing in a full verification.

37.9.3. Options

Option		Description
`-c|--chunksize`		I/O chunk size in bytes (default value is 1048576).
`>-f|--force>`		Forces the test to run without a confirmation that the device will be overwritten and all data will be permanently destroyed.
`-h|--help`		Displays a brief help message.
`-o >`*`offset`*		Offset (in kilobytes) of the start of the test (default value is 0).
`-l|--long`		Runs a full check, writing and then reading and verifying every block on the disk.
`-p|--partial`		Runs a partial check, only doing periodic checks across the device (1 GB steps).
`-r|--read`		Runs the test in read (verify) mode only, after having previously run the test in `-w` mode.
`-t` *`timestamp`*		Sets the test start time as printed at the start of a previously-interrupted test to ensure that validation data is the same across the entire file system (default value is the current time()).
`-v|--verbose`		Runs the test in verbose mode, listing each read and write operation.
`-w|--write`		Runs the test in write (test-pattern) mode (default runs both read and write).

37.9.4. Examples

Runs a partial device verification on /dev/sda:

```
llverdev -v -p /dev/sda
llverdev: permanently overwrite all data on /dev/sda (yes/no)? y
llverdev: /dev/sda is 4398046511104 bytes (4096.0 GB) in size
Timestamp: 1009839028
Current write offset: 4096 kB
```

Continues an interrupted verification at offset 4096kB from the start of the device, using the same timestamp as the previous run:

```
llverdev -f -v -p --offset=4096 --timestamp=1009839028 /dev/sda
llverdev: /dev/sda is 4398046511104 bytes (4096.0 GB) in size
Timestamp: 1009839028
write complete
read complete
```

37.10. lshowmount

The lshowmount utility shows Lustre exports.

37.10.1. Synopsis

```
lshowmount [-ehlv]
```

37.10.2. Description

The lshowmount utility shows the hosts that have Lustre mounted to a server. This utility looks for exports from the MGS, MDS, and obdfilter.

37.10.3. Options

Option	Description
`-e|--enumerate`	Causes lshowmount to list each client mounted on a separate line instead of trying to compress the list of clients into a hostrange string.
`-h|--help`	Causes lshowmount to print out a usage message.
`-l|--lookup`	Causes lshowmount to try to look up the hostname for NIDs that look like IP addresses.
`-v|--verbose`	Causes lshowmount to output export information for each service instead of only displaying the aggregate information for all Lustre services on the server.

37.10.4. Files

```
/proc/fs/lustre/mgs/server/exports/uuid/nid
/proc/fs/lustre/mds/server/exports/uuid/nid
/proc/fs/lustre/obdfilter/server/exports/uuid/nid
```

37.11. lst

The lst utility starts LNET self-test.

37.11.1. Synopsis

```
lst
```

37.11.2. Description

LNET self-test helps site administrators confirm that Lustre Networking (LNET) has been properly installed and configured. The self-test also confirms that LNET and the network software and hardware underlying it are performing as expected.

Each LNET self-test runs in the context of a session. A node can be associated with only one session at a time, to ensure that the session has exclusive use of the nodes on which it is running. A session is create, controlled and monitored from a single node; this is referred to as the self-test console.

Any node may act as the self-test console. Nodes are named and allocated to a self-test session in groups. This allows all nodes in a group to be referenced by a single name.

Test configurations are built by describing and running test batches. A test batch is a named collection of tests, with each test composed of a number of individual point-to-point tests running in parallel. These individual point-to-point tests are instantiated according to the test type, source group, target group and distribution specified when the test is added to the test batch.

37.11.3. Modules

To run LNET self-test, load these modules: libcfs, lnet, lnet_selftest and any one of the klnds (ksocklnd, ko2iblnd...). To load all necessary modules, run modprobe lnet_selftest, which recursively loads the modules on which lnet_selftest depends.

There are two types of nodes for LNET self-test: the console node and test nodes. Both node types require all previously-specified modules to be loaded. (The userspace test node does not require these modules).

Test nodes can be in either kernel or in userspace. A console user can invite a kernel test node to join the test session by running lst add_group NID, but the user cannot actively add a userspace test node to the test session. However, the console user can passively accept a test node to the test session while the test node runs lst client to connect to the console.

37.11.4. Utilities

LNET self-test includes two user utilities, lst and lstclient.

lst is the user interface for the self-test console (run on the console node). It provides a list of commands to control the entire test system, such as create session, create test groups, etc.

lstclient is the userspace self-test program which is linked with userspace LNDs and LNET. A user can invoke lstclient to join a self-test session:

```
lstclient -sesid CONSOLE_NID group NAME
```

37.11.5. Example Script

This is a sample LNET self-test script which simulates the traffic pattern of a set of Lustre servers on a TCP network, accessed by Lustre clients on an IB network (connected via LNET routers), with half the clients reading and half the clients writing.

```
#!/bin/bash
export LST_SESSION=$$
lst new_session read/write
lst add_group servers 192.168.10.[8,10,12-16]@tcp
```

```
lst add_group readers 192.168.1.[1-253/2]@o2ib
lst add_group writers 192.168.1.[2-254/2]@o2ib
lst add_batch bulk_rw
lst add_test --batch bulk_rw --from readers --to servers     brw read check\
=simple size=1M
lst add_test --batch bulk_rw --from writers --to servers     brw write chec\
k=full size=4K
# start running
lst run bulk_rw
# display server stats for 30 seconds
lst stat servers & sleep 30; kill $!
# tear down
lst end_session
```

37.12. lustre_rmmod.sh

The lustre_rmmod.sh utility removes all Lustre and LNET modules (assuming no Lustre services are running). It is located in /usr/bin.

Note

The lustre_rmmod.sh utility does not work if Lustre modules are being used or if you have manually run the lctl network up command.

37.13. lustre_rsync

The lustre_rsync utility synchronizes (replicates) a Lustre file system to a target file system.

37.13.1. Synopsis

```
lustre_rsync --source|-s src --target|-t tgt
   --mdt|-m mdt [--user|-u userid]
   [--xattr|-x yes|no] [--verbose|-v]
   [--statuslog|-l log] [--dry-run] [--abort-on-err]

lustre_rsync --statuslog|-l log

lustre_rsync --statuslog|-l log --source|-s source
   --target|-t tgt --mdt|-m mdt
```

37.13.2. Description

The lustre_rsync utility is designed to synchronize (replicate) a Lustre file system (source) to another file system (target). The target can be a Lustre file system or any other type, and is a normal, usable file system. The synchronization operation is efficient and does not require directory walking, as lustre_rsync uses Lustre MDT changelogs to identify changes in the Lustre file system.

Before using lustre_rsync:

- A changelog user must be registered (see lctl (8) changelog_register)

- AND -

- Verify that the Lustre file system (source) and the replica file system (target) are identical before the changelog user is registered. If the file systems are discrepant, use a utility, e.g. regular rsync (not lustre_rsync) to make them identical.

37.13.3. Options

Option	Description
`--source=`*`src`*	The path to the root of the Lustre file system (source) which will be synchronized. This is a mandatory option if a valid status log created during a previous synchronization operation (--statuslog) is not specified.
`--target=`*`tgt`*	The path to the root where the source file system will be synchronized (target). This is a mandatory option if the status log created during a previous synchronization operation (--statuslog) is not specified. This option can be repeated if multiple synchronization targets are desired.
`--mdt=`*`mdt`*	The metadata device to be synchronized. A changelog user must be registered for this device. This is a mandatory option if a valid status log created during a previous synchronization operation (--statuslog) is not specified.
`--user=`*`userid`*	The changelog user ID for the specified MDT. To use lustre_rsync, the changelog user must be registered. For details, see the changelog_register parameter in the lctl man page. This is a mandatory option if a valid status log created during a previous synchronization operation (--statuslog) is not specified.
`--statuslog=`*`log`*	A log file to which synchronization status is saved. When lustre_rsync starts, the state of a previous replication is read from here. If the status log from a previous synchronization operation is specified, otherwise mandatory options like --source, --target and --mdt options may be skipped. By specifying options like --source, --target and/or --mdt in addition to the --statuslog option, parameters in the status log can be overridden. Command line options take precedence over options in the status log.
`--xattr`*`yes`*`|`*`no`*	Specifies whether extended attributes (xattrs) are synchronized or not. The default is to synchronize extended attributes. NOTE: Disabling xattrs causes Lustre striping information not to be synchronized.
`--verbose`	Produces a verbose output.
`--dry-run`	Shows the output of lustre_rsync commands (copy, mkdir, etc.) on the target file system without actually executing them.

Option	Description
--abort-on-err	Shows the output of lustre_rsync commands (copy, mkdir, etc.) on the target file system without actually executing them.

37.13.4. Examples

Register a changelog user for an MDT (e.g., MDT lustre-MDT0000).

```
$ ssh
$ MDS lctl changelog_register \
           --device lustre-MDT0000 -n
cl1
```

Synchronize/replicate a Lustre file system (/mnt/lustre) to a target file system (/mnt/target).

```
$ lustre_rsync --source=/mnt/lustre --target=/mnt/target \
           --mdt=lustre-MDT0000 --user=cl1 \
           --statuslog replicate.log  --verbose
Lustre filesystem: lustre
MDT device: lustre-MDT0000
Source: /mnt/lustre
Target: /mnt/target
Statuslog: sync.log
Changelog registration: cl1
Starting changelog record: 0
Errors: 0
lustre_rsync took 1 seconds
Changelog records consumed: 22
```

After the file system undergoes changes, synchronize the changes with the target file system. Only the statuslog name needs to be specified, as it has all the parameters passed earlier.

```
$ lustre_rsync --statuslog replicate.log --verbose
Replicating Lustre filesystem: lustre
MDT device: lustre-MDT0000
Source: /mnt/lustre
Target: /mnt/target
Statuslog: replicate.log
Changelog registration: cl1
Starting changelog record: 22
Errors: 0
lustre_rsync took 2 seconds
Changelog records consumed: 42
```

Synchronize a Lustre file system (/mnt/lustre) to two target file systems (/mnt/target1 and /mnt/target2).

```
$ lustre_rsync --source=/mnt/lustre \
   --target=/mnt/target1 --target=/mnt/target2 \
   --mdt=lustre-MDT0000 --user=cl1
   --statuslog replicate.log
```

37.13.5. See Also

Section 33.1, “ lfs ”

37.14. mkfs.lustre

The `mkfs.lustre` utility formats a disk for a Lustre service.

37.14.1. Synopsis

`mkfs.lustre` *`target_type`* `[options]` *`device`*

where *`target_type`* is one of the following:

Option	Description
`--ost`	Object storage target (OST)
`--mdt`	Metadata storage target (MDT)
`--network=`*`net,...`*	Network(s) to which to restrict this OST/MDT. This option can be repeated as necessary.
`--mgs`	Configuration management service (MGS), one per site. This service can be combined with one `--mdt` service by specifying both types.

37.14.2. Description

`mkfs.lustre` is used to format a disk device for use as part of a Lustre file system. After formatting, a disk can be mounted to start the Lustre service defined by this command.

When the file system is created, parameters can simply be added as a `--param` option to the `mkfs.lustre` command. See Section 13.9.1, “Setting Tunable Parameters with `mkfs.lustre`”.

Option	Description
`--backfstype=`*`fstype`*	Forces a particular format for the backing file system (such as ext3, ldiskfs).
`--comment=`*`comment`*	Sets a user comment about this disk, ignored by the Lustre software.
`--device-size=#>KB`	Sets the device size for loop devices.
`--dryrun`	Only prints what would be done; it does not affect the disk.
`--servicenode=`*`nid,...`*	Sets the NID(s) of all service nodes, including primary and failover partner service nodes. The `--servicenode` option cannot be used with `--failnode` option. See Section 11.2, “Preparing a Lustre File System for Failover” for more details.
`--failnode=`*`nid,...`*	Sets the NID(s) of a failover service node for a primary server for a target. The `--failnode` option cannot be used with `--servicenode` option. See Section 11.2, “Preparing a Lustre File System for Failover” for more details. **Note** When the `--failnode` option is used, certain restrictions apply (see Section 11.2, “Preparing a Lustre File System for Failover”).
`--fsname=`*`filesystem_name`*	The Lustre file system of which this service/node will be a part. The default file system name is `lustre`.

Option		Description
		Note The file system name is limited to 8 characters.
`--index=`*`index_number`*		Specifies the OST or MDT number (0...N). This allows mapping between the OSS and MDS node and the device on which the OST or MDT is located.
`--mkfsoptions=`*`opts`*		Formats options for the backing file system. For example, ext3 options could be set here.
`--mountfsoptions=`*`opts`*		Sets the mount options used when the backing file system is mounted. **Warning** Unlike earlier versions of `mkfs.lustre`, this version completely replaces the default mount options with those specified on the command line, and issues a warning on stderr if any default mount options are omitted. The defaults for ldiskfs are: OST: `errors=remount-ro`; MGS/MDT: `errors=remount-ro,iopen_nopriv,user_xattr` Use care when altering the default mount options.
`--network=`*`net,...`*		Network(s) to which to restrict this OST/MDT. This option can be repeated as necessary.
`--mgsnode=`*`nid,...`*		Sets the NIDs of the MGS node, required for all targets other than the MGS.
`--param` *`key=value`*		Sets the permanent parameter *`key`* to value *`value`*. This option can be repeated as necessary. Typical options might include:
	`--param sys.timeout=40`>	System obd timeout.
	`--param lov.stripesize=2M`	Default stripe size.
	`param lov.stripecount=2`	Default stripe count.
	`--param failover.mode=failout`	Returns errors instead of waiting for recovery.
`--quiet`		Prints less information.
`--reformat`		Reformats an existing Lustre disk.
`--stripe-count-hint=stripes`		Used to optimize the MDT's inode size.

Option	Description
`--verbose`	Prints more information.

37.14.3. Examples

Creates a combined MGS and MDT for file system `testfs` on, e.g., node `cfs21`:

```
mkfs.lustre --fsname=testfs --mdt --mgs /dev/sda1
```

Creates an OST for file system `testfs` on any node (using the above MGS):

```
mkfs.lustre --fsname=testfs --mgsnode=cfs21@tcp0 --ost --index=0 /dev/sdb
```

Creates a standalone MGS on, e.g., node `cfs22`:

```
mkfs.lustre --mgs /dev/sda1
```

Creates an MDT for file system `myfs1` on any node (using the above MGS):

```
mkfs.lustre --fsname=myfs1 --mdt --mgsnode=cfs22@tcp0 /dev/sda2
```

37.14.4. See Also

- Section 37.14, “ mkfs.lustre”mkfs.lustre,
- Section 37.15, “ mount.lustre”mount.lustre,
- Section 33.1, “ `lfs` ”lfs

37.15. mount.lustre

The mount.lustre utility starts a Lustre client or target service.

37.15.1. Synopsis

```
mount -t lustre [-o options] directory
```

37.15.2. Description

The mount.lustre utility starts a Lustre client or target service. This program should not be called directly; rather, it is a helper program invoked through mount(8), as shown above. Use the umount command to stop Lustre clients and targets.

There are two forms for the device option, depending on whether a client or a target service is started:

Option	Description
`mgs_nid``:/`*`fsname`*	Mounts the Lustre file system named `fsname` on the client by contacting the Management Service at `mgsspec` on the pathname given by `directory`. The format for `mgsspec` is defined below. A

Option	Description
	mounted client file system appears in fstab(5) and is usable, like any local file system, and provides a full POSIX standard-compliant interface.
block_device	Starts the target service defined by the mkfs.lustre command on the physical disk *block_device*. A mounted target service file system is only useful for df(1) operations and appears in fstab(5) to show the device is in use.

37.15.3. Options

Option	Description
mgsspec:=*mgsnode*[:*mgsnode*]	The MGS specification may be a colon-separated list of nodes.
mgsnode:=*mgsnid*[,*mgsnid*]	Each node may be specified by a comma-separated list of NIDs.

In addition to the standard mount options, Lustre understands the following client-specific options:

Option	Description
flock	Enables full flock support, coherent across all client nodes.
localflock	Enables local flock support, using only client-local flock (faster, for applications that require flock, but do not run on multiple nodes).
noflock	Disables flock support entirely. Applications calling flock get an error. It is up to the administrator to choose either localflock (fastest, low impact, not coherent between nodes) or flock (slower, performance impact for use, coherent between nodes).
user_xattr	Enables get/set of extended attributes by regular users. See the attr(5) manual page.
nouser_xattr	Disables use of extended attributes by regular users. Root and system processes can still use extended attributes.
acl	Enables POSIX Access Control List support. See the acl(5) manual page.
noacl	Disables Access Control List support.
verbose	Enable mount/umount console messages.
noverbose	Disable mount/umount console messages.
user_fid2path	Introduced in Lustre 2.3 Enable FID to path translation by regular users. Note: This option allows a potential security hole because it allows regular users direct access to a file

Option	Description
	by its FID, bypassing POSIX path-based permission checks which could otherwise prevent the user from accessing a file in a directory that they do not have access to. Regular permission checks are still performed on the file itself, so the user cannot access a file to which they have no access rights.
`nouser_fid2path`	Introduced in Lustre 2.3 Disable FID to path translation by regular users. Root and processes with CAP_DAC_READ_SEARCH can still perform FID to path translation.

In addition to the standard mount options and backing disk type (e.g. ext3) options, Lustre understands the following server-specific options:

Option	Description
`nosvc`	Starts the MGC (and MGS, if co-located) for a target service, not the actual service.
`nomgs`	Starts only the MDT (with a co-located MGS), without starting the MGS.
`exclude=`*`ostlist`*	Starts a client or MDT with a colon-separated list of known inactive OSTs.
`abort_recov`	Aborts client recovery and starts the target service immediately.
`md_stripe_cache_size`	Sets the stripe cache size for server-side disk with a striped RAID configuration.
`recovery_time_soft=`*`timeout`*	Allows `timeout` seconds for clients to reconnect for recovery after a server crash. This timeout is incrementally extended if it is about to expire and the server is still handling new connections from recoverable clients. The default soft recovery timeout is 300 seconds (5 minutes).
`recovery_time_hard=`*`timeout`*	The server is allowed to incrementally extend its timeout up to a hard maximum of `timeout` seconds. The default hard recovery timeout is set to 900 seconds (15 minutes).

37.15.4. Examples

Starts a client for the Lustre file system testfs at mount point /mnt/myfilesystem. The Management Service is running on a node reachable from this client via the cfs21@tcp0 NID.

```
mount -t lustre cfs21@tcp0:/testfs /mnt/myfilesystem
```

Starts the Lustre metadata target service from /dev/sda1 on mount point /mnt/test/mdt.

```
mount -t lustre /dev/sda1 /mnt/test/mdt
```

Starts the testfs-MDT0000 service (using the disk label), but aborts the recovery process.

```
mount -t lustre -L testfs-MDT0000 -o abort_recov /mnt/test/mdt
```

37.15.5. See Also

- Section 37.14, “ mkfs.lustre”
- Section 37.18, “ tunefs.lustre”
- Section 37.3, “ lctl”
- Section 33.1, “ `lfs` ”

37.16. plot-llstat

The plot-llstat utility plots Lustre statistics.

37.16.1. Synopsis

```
plot-llstat results_filename [parameter_index]
```

37.16.2. Description

The plot-llstat utility generates a CSV file and instruction files for gnuplot from the output of llstat. Since llstat is generic in nature, plot-llstat is also a generic script. The value of parameter_index can be 1 for count per interval, 2 for count per second (default setting) or 3 for total count.

The plot-llstat utility creates a .dat (CSV) file using the number of operations specified by the user. The number of operations equals the number of columns in the CSV file. The values in those columns are equal to the corresponding value of parameter_index in the output file.

The plot-llstat utility also creates a .scr file that contains instructions for gnuplot to plot the graph. After generating the .dat and .scr files, the plot-llstat tool invokes gnuplot to display the graph.

37.16.3. Options

Option	Description
`results_filename`	Output generated by plot-llstat
`parameter_index`	Value of parameter_index can be: 1 - count per interval 2 - count per second (default setting) 3 - total count

37.16.4. Example

```
llstat -i2 -g -c lustre-OST0000 > log
plot-llstat log 3
```

37.17. routerstat

The routerstat utility prints Lustre router statistics.

37.17.1. Synopsis

```
routerstat [interval]
```

37.17.2. Description

The routerstat utility watches LNET router statistics. If no *interval* is specified, then statistics are sampled and printed only one time. Otherwise, statistics are sampled and printed at the specified *interval* (in seconds).

37.17.3. Options

The routerstat output includes the following fields:

Option	Description
M	msgs_alloc(msgs_max)
E	errors
S	send_count/send_length
R	recv_count/recv_length
F	route_count/route_length
D	drop_count/drop_length

37.17.4. Files

The routerstat utility extracts statistics data from:

```
/proc/sys/lnet/stats
```

37.18. tunefs.lustre

The tunefs.lustre utility modifies configuration information on a Lustre target disk.

37.18.1. Synopsis

```
tunefs.lustre [options] /dev/device
```

37.18.2. Description

tunefs.lustre is used to modify configuration information on a Lustre target disk. This does not reformat the disk or erase the target information, but modifying the configuration information can result in an unusable file system.

Caution

Changes made here affect a file system only when the target is mounted the next time.

With tunefs.lustre, parameters are "additive" -- new parameters are specified in addition to old parameters, they do not replace them. To erase all old tunefs.lustre parameters and just use newly-specified parameters, run:

```
$ tunefs.lustre --erase-params --param=new_parameters
```

The tunefs.lustre command can be used to set any parameter settable in a /proc/fs/lustre file and that has its own OBD device, so it can be specified as *{obd|fsname}.obdtype.proc_file_name=value*. For example:

```
$ tunefs.lustre --param mdt.identity_upcall=NONE /dev/sda1
```

37.18.3. Options

The tunefs.lustre options are listed and explained below.

Option	Description
`--comment=`*comment*	Sets a user comment about this disk, ignored by Lustre.
`--dryrun`	Only prints what would be done; does not affect the disk.
`--erase-params`	Removes all previous parameter information.
`--servicenode=`*nid,...*	Sets the NID(s) of all service nodes, including primary and failover partner service nodes. The `--servicenode` option cannot be used with `--failnode` option. See Section 11.2, “Preparing a Lustre File System for Failover” for more details.
`--failnode=`*nid,...*	Sets the NID(s) of a failover service node for a primary server for a target. The `--failnode` option cannot be used with `--servicenode` option. See Section 11.2, “Preparing a Lustre File System for Failover” for more details. **Note** When the `--failnode` option is used, certain restrictions apply (see Section 11.2, “Preparing a Lustre File System for Failover”).
`--fsname=`*filesystem_name*	The Lustre file system of which this service will be a part. The default file system name is `lustre`.
`--index=`*index*	Forces a particular OST or MDT index.
`--mountfsoptions=`*opts*	Sets the mount options used when the backing file system is mounted. **Warning** Unlike earlier versions of tunefs.lustre, this version completely replaces the existing mount options with those specified on the command line, and issues a warning on

Option	Description
	stderr if any default mount options are omitted. The defaults for ldiskfs are: OST: `errors=remount-ro,mballoc,extents`; MGS/MDT: `errors=remount-ro,iopen_nopriv,user_xattr` Do not alter the default mount options unless you know what you are doing.
`--network=`*`net,...`*	Network(s) to which to restrict this OST/MDT. This option can be repeated as necessary.
`--mgs`	Adds a configuration management service to this target.
`--msgnode=`*`nid,...`*	Sets the NID(s) of the MGS node; required for all targets other than the MGS.
`--nomgs`	Removes a configuration management service to this target.
`--quiet`	Prints less information.
`--verbose`	Prints more information.
`--writeconf`	Erases all configuration logs for the file system to which this MDT belongs, and regenerates them. This is dangerous operation. All clients must be unmounted and servers for this file system should be stopped. All targets (OSTs/MDTs) must then be restarted to regenerate the logs. No clients should be started until all targets have restarted. The correct order of operations is: 1. Unmount all clients on the file system 2. Unmount the MDT and all OSTs on the file system 3. Run `tunefs.lustre --writeconf` *`device`* on every server 4. Mount the MDT and OSTs 5. Mount the clients

37.18.4. Examples

Change the MGS's NID address. (This should be done on each target disk, since they should all contact the same MGS.)

```
tunefs.lustre --erase-param --mgsnode=new_nid --writeconf /dev/sda
```

Add a failover NID location for this target.

```
tunefs.lustre --param="failover.node=192.168.0.13@tcp0" /dev/sda
```

37.18.5. See Also

- Section 37.14, “ mkfs.lustre”
- Section 37.15, “ mount.lustre”
- Section 37.3, “ lctl”
- Section 33.1, “ `lfs` ”

37.19. Additional System Configuration Utilities

This section describes additional system configuration utilities for Lustre.

37.19.1. Application Profiling Utilities

The following utilities are located in /usr/bin.

`lustre_req_history.sh`

The lustre_req_history.sh utility (run from a client), assembles as much Lustre RPC request history as possible from the local node and from the servers that were contacted, providing a better picture of the coordinated network activity.

`llstat.sh`

The llstat.sh utility handles a wider range of statistics files, and has command line switches to produce more graphable output.

`plot-llstat.sh`

The plot-llstat.sh utility plots the output from llstat.sh using gnuplot.

37.19.2. More /proc Statistics for Application Profiling

The following utilities provide additional statistics.

`vfs_ops_stats`

The client vfs_ops_stats utility tracks Linux VFS operation calls into Lustre for a single PID, PPID, GID or everything.

```
/proc/fs/lustre/llite/*/vfs_ops_stats
/proc/fs/lustre/llite/*/vfs_track_[pid|ppid|gid]
```

`extents_stats`

The client extents_stats utility shows the size distribution of I/O calls from the client (cumulative and by process).

```
/proc/fs/lustre/llite/*/extents_stats, extents_stats_per_process
```

`offset_stats`

The client offset_stats utility shows the read/write seek activity of a client by offsets and ranges.

`/proc/fs/lustre/llite/*/offset_stats`

Lustre includes per-client and improved MDT statistics:

- Per-client statistics tracked on the servers

Each MDT and OST now tracks LDLM and operations statistics for every connected client, for comparisons and simpler collection of distributed job statistics.

`/proc/fs/lustre/mds|obdfilter/*/exports/`

- Improved MDT statistics

More detailed MDT operations statistics are collected for better profiling.

`/proc/fs/lustre/mds/*/stats`

37.19.3. Testing / Debugging Utilities

Lustre offers the following test and debugging utilities.

37.19.3.1. loadgen

The Load Generator (loadgen) is a test program designed to simulate large numbers of Lustre clients connecting and writing to an OST. The loadgen utility is located at lustre/utils/loadgen (in a build directory) or at /usr/sbin/loadgen (from an RPM).

Loadgen offers the ability to run this test:

1. Start an arbitrary number of (echo) clients.
2. Start and connect to an echo server, instead of a real OST.
3. Create/bulk_write/delete objects on any number of echo clients simultaneously.

Currently, the maximum number of clients is limited by MAX_OBD_DEVICES and the amount of memory available.

37.19.3.2. Usage

The loadgen utility can be run locally on the OST server machine or remotely from any LNET host. The device command can take an optional NID as a parameter; if unspecified, the first local NID found is used.

The obdecho module must be loaded by hand before running loadgen.

```
# cd lustre/utils/
# insmod ../obdecho/obdecho.ko
# ./loadgen
loadgen> h
This is a test program used to simulate large numbers of clients. The echo \
obds are used, so the obdecho module must be loaded.

Typical usage would be:
loadgen> dev lustre-OST0000       set the target device
loadgen> start 20                 start 20 echo clients
```

```
loadgen> wr 10 5                  have 10 clients do simultaneous brw_write \
tests 5 times each

Available commands are:
   device
   dl
   echosrv
   start
   verbose
   wait
   write
   help
   exit
   quit

For more help type: help command-name
loadgen>
loadgen> device lustre-OST0000 192.168.0.21@tcp
Added uuid OSS_UUID: 192.168.0.21@tcp
Target OST name is 'lustre-OST0000'
loadgen>
loadgen> st 4
start 0 to 4
./loadgen: running thread #1
./loadgen: running thread #2
./loadgen: running thread #3
./loadgen: running thread #4
loadgen> wr 4 5
Estimate 76 clients before we run out of grant space (155872K / 2097152)
1: i0
2: i0
4: i0
3: i0
1: done (0)
2: done (0)
4: done (0)
3: done (0)
wrote 25MB in 1.419s (17.623 MB/s)
loadgen>
```

The loadgen utility prints periodic status messages; message output can be controlled with the verbose command.

To insure that a file can be written to (a requirement of write cache), OSTs reserve ("grants"), chunks of space for each newly-created file. A grant may cause an OST to report that it is out of space, even though there is plenty of space on the disk, because the space is "reserved" by other files. The loadgen utility estimates the number of simultaneous open files as the disk size divided by the grant size and reports that number when the write tests are first started.

Echo Server

The loadgen utility can start an echo server. On another node, loadgen can specify the echo server as the device, thus creating a network-only test environment.

```
loadgen> echosrv
```

```
loadgen> dl
   0 UP obdecho echosrv echosrv 3
   1 UP ost OSS OSS 3
```

On another node:

```
loadgen> device echosrv cfs21@tcp
Added uuid OSS_UUID: 192.168.0.21@tcp
Target OST name is 'echosrv'
loadgen> st 1
start 0 to 1
./loadgen: running thread #1
loadgen> wr 1
start a test_brw write test on X clients for Y iterations
usage: write <num_clients> <num_iter> [<delay>]
loadgen> wr 1 1
loadgen>
1: i0
1: done (0)
wrote 1MB in 0.029s (34.023 MB/s)
```

Scripting

The threads all perform their actions in non-blocking mode; use the wait command to block for the idle state. For example:

```
#!/bin/bash
./loadgen << EOF
device lustre-OST0000
st 1
wr 1 10
wait
quit
EOF
```

Feature Requests

The loadgen utility is intended to grow into a more comprehensive test tool; feature requests are encouraged. The current feature requests include:

- Locking simulation
- • Many (echo) clients cache locks for the specified resource at the same time.
  - Many (echo) clients enqueue locks for the specified resource simultaneously.
- obdsurvey functionality
- • Fold the Lustre I/O kit's obdsurvey script functionality into loadgen

37.19.3.3. llog_reader

The llog_reader utility translates a Lustre configuration log into human-readable form.

37.19.3.4. Synopsis

```
llog_reader filename
```

37.19.3.5. Description

llog_reader parses the binary format of Lustre's on-disk configuration logs. It can only read the logs. Use tunefs.lustre to write to them.

To examine a log file on a stopped Lustre server, mount its backing file system as ldiskfs, then use llog_reader to dump the log file's contents. For example:

```
mount -t ldiskfs /dev/sda /mnt/mgs
llog_reader /mnt/mgs/CONFIGS/tfs-client
```

To examine the same log file on a running Lustre server, use the ldiskfs-enabled debugfs utility (called debug.ldiskfs on some distributions) to extract the file. For example:

```
debugfs -c -R 'dump CONFIGS/tfs-client /tmp/tfs-client' /dev/sda
llog_reader /tmp/tfs-client
```

Caution

Although they are stored in the CONFIGS directory, mountdata files do not use the config log format and will confuse llog_reader.

See Also Section 37.18, “ tunefs.lustre”

37.19.3.6. lr_reader

The lr_reader utility translates a last received (last_rcvd) file into human-readable form.

The following utilities are part of the Lustre I/O kit. For more information, see Chapter 25, *Benchmarking Lustre File System Performance (Lustre I/O Kit)*.

37.19.3.7. sgpdd-survey

The `sgpdd-survey` utility tests 'bare metal' performance, bypassing as much of the kernel as possible. The `sgpdd-survey` tool does not require Lustre, but it does require the sgp_dd package.

Caution

The `sgpdd-survey` utility erases all data on the device.

37.19.3.8. obdfilter-survey

The `obdfilter-survey` utility is a shell script that tests performance of isolated OSTS, the network via echo clients, and an end-to-end test.

37.19.3.9. ior-survey

The ior-survey utility is a script used to run the IOR benchmark. Lustre includes IOR version 2.8.6.

37.19.3.10. ost-survey

The `ost-survey` utility is an OST performance survey that tests client-to-disk performance of the individual OSTs in a Lustre file system.

37.19.3.11. stats-collect

The stats-collect utility contains scripts used to collect application profiling information from Lustre clients and servers.

37.19.4. Flock Feature

Lustre now includes the flock feature, which provides file locking support. Flock describes classes of file locks known as 'flocks'. Flock can apply or remove a lock on an open file as specified by the user. However, a single file may not, simultaneously, have both shared and exclusive locks.

By default, the flock utility is disabled on Lustre. Two modes are available.

`local mode`	In this mode, locks are coherent on one node (a single-node flock), but not across all clients. To enable it, use -o localflock. This is a client-mount option. **Note** This mode does not impact performance and is appropriate for single-node databases.
`consistent mode`	In this mode, locks are coherent across all clients. To enable it, use the -o flock. This is a client-mount option. **Warning** This mode affects the performance of the file being flocked and may affect stability, depending on the Lustre version used. Consider using a newer Lustre version which is more stable. If the consistent mode is enabled and no applications are using flock, then it has no effect.

A call to use flock may be blocked if another process is holding an incompatible lock. Locks created using flock are applicable for an open file table entry. Therefore, a single process may hold only one type of lock (shared or exclusive) on a single file. Subsequent flock calls on a file that is already locked converts the existing lock to the new lock mode.

37.19.4.1. Example

```
$ mount -t lustre -o flock mds@tcp0:/lustre /mnt/client
```

You can check it in /etc/mtab. It should look like,

```
mds@tcp0:/lustre /mnt/client lustre rw,flock         0       0
```

Glossary

A

ACL	Access control list. An extended attribute associated with a file that contains enhanced authorization directives.
Administrative OST failure	A manual configuration change to mark an OST as unavailable, so that operations intended for that OST fail immediately with an I/O error instead of waiting indefinitely for OST recovery to complete

C

Completion callback	An RPC made by the lock server on an OST or MDT to another system, usually a client, to indicate that the lock is now granted.
configlog	An llog file used in a node, or retrieved from a management server over the network with configuration instructions for the Lustre file system at startup time.
Configuration lock	A lock held by every node in the cluster to control configuration changes. When the configuration is changed on the MGS, it revokes this lock from all nodes. When the nodes receive the blocking callback, they quiesce their traffic, cancel and re-enqueue the lock and wait until it is granted again. They can then fetch the configuration updates and resume normal operation.

D

Default stripe pattern	Information in the LOV descriptor that describes the default stripe count, stripe size, and layout pattern used for new files in a file system. This can be amended by using a directory stripe descriptor or a per-file stripe descriptor.
Direct I/O	A mechanism that can be used during read and write system calls to avoid memory cache overhead for large I/O requests. It bypasses the data copy between application and kernel memory, and avoids buffering the data in the client memory.
Directory stripe descriptor	An extended attribute that describes the default stripe pattern for new files created within that directory. This is also inherited by new subdirectories at the time they are created.
Distributed namespace (DNE)	A collection of metadata targets implementing a single file system namespace.

E

EA	Extended attribute. A small amount of data that can be retrieved through a name (EA or attr) associated with a particular inode. A Lustre file system uses EAs to store striping information (indicating the location of file data on OSTs). Examples of extended attributes are ACLs, striping information, and the FID of the file.
Eviction	The process of removing a client's state from the server if the client is unresponsive to server requests after a timeout or if server recovery fails. If a client is still running, it is required to flush the cache associated with the server when it becomes aware that it has been evicted.

Export — The state held by a server for a client that is sufficient to transparently recover all in-flight operations when a single failure occurs.

Extent — A range of contiguous bytes or blocks in a file that are addressed by a {start, length} tuple instead of individual block numbers.

Extent lock — An LDLM lock used by the OSC to protect an extent in a storage object for concurrent control of read/write, file size acquisition, and truncation operations.

F

Failback — The failover process in which the default active server regains control from the backup server that had taken control of the service.

Failout OST — An OST that is not expected to recover if it fails to answer client requests. A failout OST can be administratively failed, thereby enabling clients to return errors when accessing data on the failed OST without making additional network requests or waiting for OST recovery to complete.

Failover — The process by which a standby computer server system takes over for an active computer server after a failure of the active node. Typically, the standby computer server gains exclusive access to a shared storage device between the two servers.

FID — Lustre File Identifier. A 128-bit file system-unique identifier for a file or object in the file system. The FID structure contains a unique 64-bit sequence number (see *FLDB*), a 32-bit object ID (OID), and a 32-bit version number. The sequence number is unique across all Lustre targets (OSTs and MDTs).

Fileset — A group of files that are defined through a directory that represents the start point of a file system.

FLDB — FID location database. This database maps a sequence of FIDs to a specific target (MDT or OST), which manages the objects within the sequence. The FLDB is cached by all clients and servers in the file system, but is typically only modified when new servers are added to the file system.

Flight group — Group of I/O RPCs initiated by the OSC that are concurrently queued or processed at the OST. Increasing the number of RPCs in flight for high latency networks can increase throughput and reduce visible latency at the client.

G

Glimpse callback — An RPC made by an OST or MDT to another system (usually a client) to indicate that a held extent lock should be surrendered. If the system is using the lock, then the system should return the object size and timestamps in the reply to the glimpse callback instead of cancelling the lock. Glimpses are introduced to optimize the acquisition of file attributes without introducing contention on an active lock.

I

Import — The state held held by the client for each target that it is connected to. It holds server NIDs, connection state, and uncommitted RPCs needed to fully recover a transaction sequence after a server failure and restart.

Intent lock — A special Lustre file system locking operation in the Linux kernel. An intent lock combines a request for a lock with the full information to perform the operation(s) for which the lock was requested. This offers the server the option of granting the lock or performing the operation and informing the client of the operation result without granting a lock. The use of intent locks enables metadata operations (even complicated ones) to be implemented with a single RPC from the client to the server.

L

LBUG — A fatal error condition detected by the software that halts execution of the kernel thread to avoid potential further corruption of the system state. It is printed to the console log and triggers a dump of the internal debug log. The system must be rebooted to clear this state.

LDLM — Lustre Distributed Lock Manager.

lfs — The Lustre file system command-line utility that allows end users to interact with Lustre software features, such as setting or checking file striping or per-target free space. For more details, see Section 33.1, “ `lfs` ”.

lfsck — Lustre file system check. A distributed version of a disk file system checker. Normally, `lfsck` does not need to be run, except when file systems are damaged by events such as multiple disk failures and cannot be recovered using file system journal recovery.

llite — Lustre lite. This term is in use inside code and in module names for code that is related to the Linux client VFS interface.

llog — Lustre log. An efficient log data structure used internally by the file system for storing configuration and distributed transaction records. An `llog` is suitable for rapid transactional appends of records and cheap cancellation of records through a bitmap.

llog catalog — Lustre log catalog. An `llog` with records that each point at an `llog`. Catalogs were introduced to give `llogs` increased scalability. `llogs` have an originator which writes records and a replicator which cancels records when the records are no longer needed.

LMV — Logical metadata volume. A module that implements a DNE client-side abstraction device. It allows a client to work with many MDTs without changes to the llite module. The LMV code forwards requests to the correct MDT based on name or directory striping information and merges replies into a single result to pass back to the higher `llite` layer that connects the Lustre file system with Linux VFS, supports VFS semantics, and complies with POSIX interface specifications.

LND — Lustre network driver. A code module that enables LNET support over particular transports, such as TCP and various kinds of InfiniBand networks.

LNET — Lustre networking. A message passing network protocol capable of running and routing through various physical layers. LNET forms the underpinning of LNETrpc.

Lock client — A module that makes lock RPCs to a lock server and handles revocations from the server.

Lock server	A service that is co-located with a storage target that manages locks on certain objects. It also issues lock callback requests, calls while servicing or, for objects that are already locked, completes lock requests.
LOV	Logical object volume. The object storage analog of a logical volume in a block device volume management system, such as LVM or EVMS. The LOV is primarily used to present a collection of OSTs as a single device to the MDT and client file system drivers.
LOV descriptor	A set of configuration directives which describes which nodes are OSS systems in the Lustre cluster and providing names for their OSTs.
Lustre client	An operating instance with a mounted Lustre file system.
Lustre file	A file in the Lustre file system. The implementation of a Lustre file is through an inode on a metadata server that contains references to a storage object on OSSs.

M

mballoc	Multi-block allocate. Functionality in ext4 that enables the `ldiskfs` file system to allocate multiple blocks with a single request to the block allocator.
MDC	Metadata client. A Lustre client component that sends metadata requests via RPC over LNET to the metadata target (MDT).
MDD	Metadata disk device. Lustre server component that interfaces with the underlying object storage device to manage the Lustre file system namespace (directories, file ownership, attributes).
MDS	Metadata server. The server node that is hosting the metadata target (MDT).
MDT	Metadata target. A storage device containing the file system namespace that is made available over the network to a client. It stores filenames, attributes, and the layout of OST objects that store the file data.
MDT0	The metadata target for the file system root. Since Lustre software release 2.4, multiple metadata targets are possible in the same file system. MDT0 is the root of the file system, which must be available for the file system to be accessible.
MGS	Management service. A software module that manages the startup configuration and changes to the configuration. Also, the server node on which this system runs.
mountconf	The Lustre configuration protocol that formats disk file systems on servers with the `mkfs.lustre` program, and prepares them for automatic incorporation into a Lustre cluster. This allows clients to be configured and mounted with a simple `mount` command.

N

NID	Network identifier. Encodes the type, network number, and network address of a network interface on a node for use by the Lustre file system.
NIO API	A subset of the LNET RPC module that implements a library for sending large network requests, moving buffers with RDMA.

Node affinity — Node affinity describes the property of a multi-threaded application to behave sensibly on multiple cores. Without the property of node affinity, an operating scheduler may move application threads across processors in a sub-optimal way that significantly reduces performance of the application overall.

NRS — Network request scheduler. A subcomponent of the PTLRPC layer, which specifies the order in which RPCs are handled at servers. This allows optimizing large numbers of incoming requests for disk access patterns, fairness between clients, and other administrator-selected policies.

NUMA — Non-uniform memory access. Describes a multi-processing architecture where the time taken to access given memory differs depending on memory location relative to a given processor. Typically machines with multiple sockets are NUMA architectures.

O

OBD — Object-based device. The generic term for components in the Lustre software stack that can be configured on the client or server. Examples include MDC, OSC, LOV, MDT, and OST.

OBD API — The programming interface for configuring OBD devices. This was formerly also the API for accessing object IO and attribute methods on both the client and server, but has been replaced by the OSD API in most parts of the code.

OBD type — Module that can implement the Lustre object or metadata APIs. Examples of OBD types include the LOV, OSC and OSD.

Obdfilter — An older name for the OBD API data object operation device driver that sits between the OST and the OSD. In Lustre software release 2.4 this device has been renamed OFD."

Object storage — Refers to a storage-device API or protocol involving storage objects. The two most well known instances of object storage are the T10 iSCSI storage object protocol and the Lustre object storage protocol (a network implementation of the Lustre object API). The principal difference between the Lustre protocol and T10 protocol is that the Lustre protocol includes locking and recovery control in the protocol and is not tied to a SCSI transport layer.

opencache — A cache of open file handles. This is a performance enhancement for NFS.

Orphan objects — Storage objects to which no Lustre file points. Orphan objects can arise from crashes and are automatically removed by an `llog` recovery between the MDT and OST. When a client deletes a file, the MDT unlinks it from the namespace. If this is the last link, it will atomically add the OST objects into a per-OST `llog` (if a crash has occurred) and then wait until the unlink commits to disk. (At this point, it is safe to destroy the OST objects. Once the destroy is committed, the MDT `llog` records can be cancelled.)

OSC — Object storage client. The client module communicating to an OST (via an OSS).

OSD — Object storage device. A generic, industry term for storage devices with a more extended interface than block-oriented devices such as disks. For the Lustre file system, this name is used to describe a software module that implements an object storage API in the kernel. It is also used to refer to an instance of an object storage

	device created by that driver. The OSD device is layered on a file system, with methods that mimic create, destroy and I/O operations on file inodes.
OSS	Object storage server. A server OBD that provides access to local OSTs.
OST	Object storage target. An OSD made accessible through a network protocol. Typically, an OST is associated with a unique OSD which, in turn is associated with a formatted disk file system on the server containing the data objects.

P

pdirops	A locking protocol in the Linux VFS layer that allows for directory operations performed in parallel.
Pool	OST pools allows the administrator to associate a name with an arbitrary subset of OSTs in a Lustre cluster. A group of OSTs can be combined into a named pool with unique access permissions and stripe characteristics.
Portal	A service address on an LNET NID that binds requests to a specific request service, such as an MDS, MGS, OSS, or LDLM. Services may listen on multiple portals to ensure that high priority messages are not queued behind many slow requests on another portal.
PTLRPC	An RPC protocol layered on LNET. This protocol deals with stateful servers and has exactly-once semantics and built in support for recovery.

R

Recovery	The process that re-establishes the connection state when a client that was previously connected to a server reconnects after the server restarts.
Replay request	The concept of re-executing a server request after the server has lost information in its memory caches and shut down. The replay requests are retained by clients until the server(s) have confirmed that the data is persistent on disk. Only requests for which a client received a reply and were assigned a transaction number by the server are replayed. Requests that did not get a reply are resent.
Resent request	An RPC request sent from a client to a server that has not had a reply from the server. This might happen if the request was lost on the way to the server, if the reply was lost on the way back from the server, or if the server crashes before or after processing the request. During server RPC recovery processing, resent requests are processed after replayed requests, and use the client RPC XID to determine if the resent RPC request was already executed on the server.
Revocation callback	Also called a "blocking callback". An RPC request made by the lock server (typically for an OST or MDT) to a lock client to revoke a granted DLM lock.
Root squash	A mechanism whereby the identity of a root user on a client system is mapped to a different identity on the server to avoid root users on clients from accessing or modifying root-owned files on the servers. This does not prevent root users on the client from assuming the identity of a non-root user, so should not be considered a robust security mechanism. Typically, for management purposes, at least one client system should not be subject to root squash.
Routing	LNET routing between different networks and LNDs.

RPC — Remote procedure call. A network encoding of a request.

S

Stripe — A contiguous, logical extent of a Lustre file written to a single OST. Used synonymously with a single OST data object that makes up part of a file visible to user applications.

Stripe size — The maximum number of bytes that will be written to an OST object before the next object in a file's layout is used when writing sequential data to a file. Once a full stripe has been written to each of the objects in the layout, the first object will be written to again in round-robin fashion.

Stripe count — The number of OSTs holding objects for a RAID0-striped Lustre file.

Striping metadata — The extended attribute associated with a file that describes how its data is distributed over storage objects. See also *Default stripe pattern*.

T

T10 object protocol — An object storage protocol tied to the SCSI transport layer. The Lustre file system does not use T10.

W

Wide striping — Strategy of using many OSTs to store stripes of a single file. This obtains maximum bandwidth access to a single file through parallel utilization of many OSTs. For more information about wide striping, see Section 18.6, “Lustre Striping Internals”.

Index

I

J

L

M

O

P

Q

R

S

T

U

V

W

X