Lustre File System™

Operations Manual for Lustre - Version 2.0

Part No.: 821-2076-10,
January 2011, Revision 02

Please
Recycle

Contents

Preface

This operations manual provides detailed information and procedures to install, configure and tune the Lustre file system. The manual covers topics such as failover, quotas, striping and bonding. The Lustre manual also contains troubleshooting information and tips to improve Lustre operation and performance.

UNIX Commands

This document might not contain information about basic UNIX 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
- Oracle Solaris Operating System documentation, which is at:

  http://docs.sun.com

Shell Prompts

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

Related Documentation

The documents listed as online are available at:

http://docs.sun.com/app/docs/prod/lustre.fs20?l=en&a=view

Application	Title	Format	Location
Latest information	*Lustre 2.0 Release Notes*	PDF	Online
Service	*Lustre 2.0 Operations Manual*	PDF HTML	Online

Documentation, Support, and Training

These web sites provide additional resources:

- Documentation http://docs.sun.com/
- Support http://www.sun.com/support/
- Training http://www.sun.com/training/

Revision History

BookTitle	Part Number	Date	Comments
Lustre 2.0 Operations Manual	821-2076-10	July 2010	First release of Lustre 2.0 manual
Lustre 2.0 Operations Manual	821-2076-10	January 2011	Second release of Lustre 2.0 manual

PART I Introducing Lustre

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

Understanding Lustre

Understanding Lustre Networking (LNET)

Understanding Failover in Lustre

CHAPTER 1

Understanding Lustre

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

1.1 What Lustre Is (and What It Isn't)

Lustre 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-compliant UNIX file system interface.

The Lustre storage architecture is used for many different kinds of clusters. It is best known for powering seven of the ten 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 Lustre 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 Lustre 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 Lustre can perform better with a single server than other filesystems due to its strong locking and data coherency.

Lustre is currently not particularly well suited for "peer-to-peer" usage models where there are clients and servers running on the same node, each sharing a small amount of storage, due to the lack of Lustre-level data replication. 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 runs on a variety of vendor's kernels. For more details see http://wiki.lustre.org/index.php/Lustre_Release_Information.

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 bandwith and the processing power of the servers in the system. Lustre 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 shows the practical range of scalability and performance characteristics of the Lustre file system and some test results in production systems.

TABLE 1-1 Lustre Scalability and Performance

	Feature	Practical Range	Tested in Production
Clients	*Scalabity*	100-100,000	50,000+ clients Many installations in 10,000 to 20,000 range
	Performance	*Single client:* 2 GB/sec I/O, 1,000 metadata ops/sec *File system:* 2.5 TB/sec	*Single client:* 2 GB/sec *File system:* 240 GB/sec I/O
OSSs	*Scalabity*	*OSSs:* 4-500 with up to 4000 OSTs *File system:* 64 PB, file size 320 TB	*OSSs:* 450 OSSs with 1,000 OSTs 192 OSSs with 1344 OSTs *File system:* 10 PB, file size multi-TB
	Performance	Up to 5 GB/sec	OSS throughput at 2.0+ GB/sec
MDSs	*Scalabity*	1 + 1 (failover with one backup)	
	Performance	Up to 35,000/s create, 100,000/s stat metadata operations	15,000/s create, 25,000/s stat metadata operations

Other Lustre features are:

- **Performance-enhanced ext4 file system:** Lustre uses a modified 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 Lustre.
- **POSIX compliance**: The full POSIX test suite passes with limited exceptions on Lustre clients. In a cluster, most operations are atomic so that clients never see stale data or metadata. Lustre supports mmap() file I/O.

- **High-performance heterogeneous networking:** Lustre supports a variety of high performance, low latency networks and permits Remote Direct Memory Access (RDMA) for Infiniband (OFED). This enables multiple, bridging RDMA networks to useLustre routing for maximum performance. Lustre also provides integrated network diagnostics.
- **High-availability:** Lustre offers active/active failover using shared storage partitions for OSS targets (OSTs) and active/passive failover using a shared storage partition for the MDS target (MDT). This allows application transparent recovery. Lustre 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:** In Lustre, an option is available to have TCP connections only from privileged ports. Group membership handling is server-based.
- **Access control list (ACL), exended attributes**: Currently, the Lustre security model follows that of a UNIX file system, enhanced with POSIX ACLs. Noteworthy additional features include root squash and connecting only from privileged ports.
- **Interoperability:** Lustre runs on a variety of CPU architectures and mixed-endian clusters and is interoperable between adjacent 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:** Any clients can operate on the same file and directory concurrently. A Lustre distributed lock manager (DLM) ensures that files are coherent between all the clients in a file system and the servers. Multiple clients can access the same files concurrently, and the DLM ensures that all the clients see consistent data at all times. The DLM on each MDT and each OST manages the locking for objects stored in that file system. The MDT manages locks on inodes permissions and path names. OST manages locks for each stripe of a file and the data within each object
- **Quotas**: User and group quotas are available for Lustre.
- **OSS addition**: The capacity of a Lustre file system and aggregate cluster bandwidth can be increased without interrupting any operations by adding a new OSS with OSTs to the cluster.
- **Controlled striping**: The distribution 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. Lustre 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:** Lustre has a dedicated MPI ADIO layer that optimizes parallel I/O to match the underlying file system architecture.

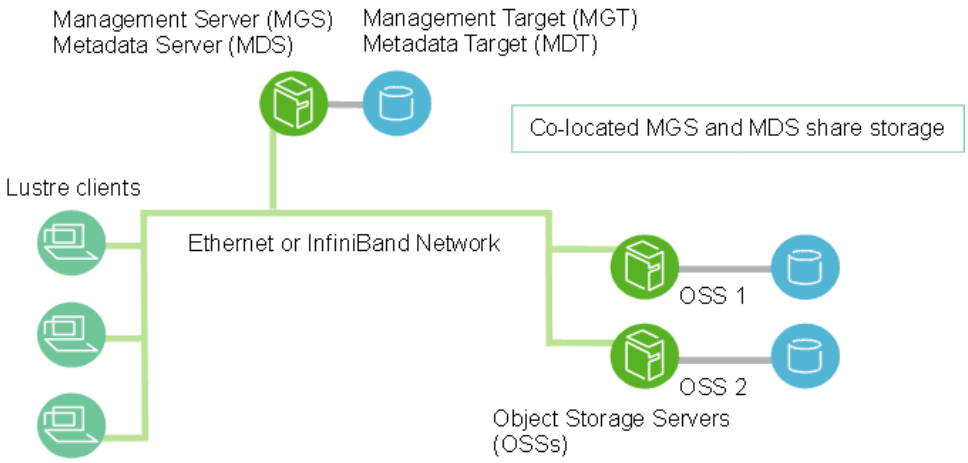
- **NFS and CIFS export:** Lustre files can be re-exported using NFS or CIFS (via Samba) enabling them to be shared with a non-Linux client.
- **Disaster recovery tool:** Lustre provides a distributed file system check (`lfsck`) that can restore consistency between storage components in case of a major file system error. Lustre can operate even in the presence of file system inconsistencies, so `lfsck` is not required before returning the file system to production.
- **Internal monitoring and instrumentation interfaces:** Lustre offers a variety of mechanisms to examine performance and tuning.
- **Open source**: Lustre is licensed under the GPL 2.0 license for use with Linux.

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 components is shown in FIGURE 1-1.

FIGURE 1-1 Lustre 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.

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**) - 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.
- **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 2 and 8 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 provides the requirements for attached storage for each Lustre file system component and describes desirable characterics of the hardware used.

TABLE 1-2 Storage and hardware requirements for Lustre 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 bandwith network.

For additional hardware requirements and considerations, see Chapter 5: Setting Up a Lustre File System.

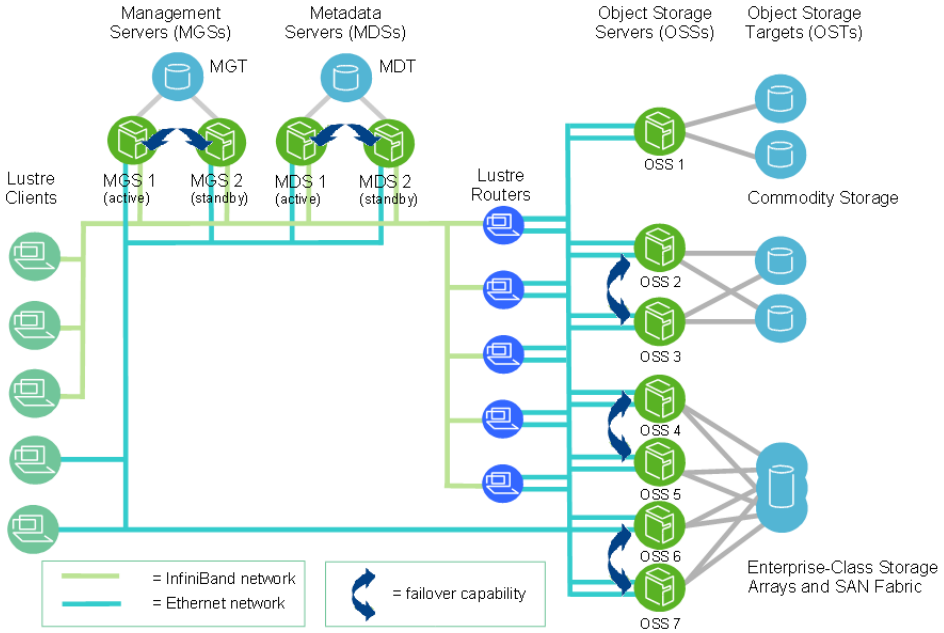
1.2.3 Lustre Networking (LNET)

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, the Lustre cluster can include hundreds of OSSs and thousands of clients (see FIGURE 1-2). 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 Lustre.

FIGURE 1-2 Lustre cluster at scale

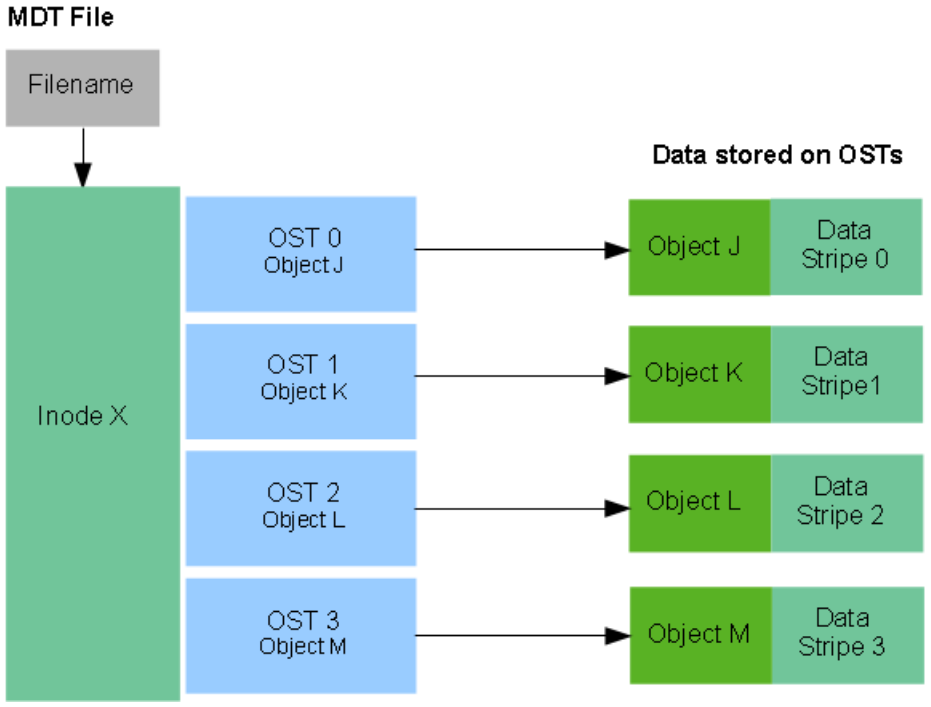


1.3 Lustre Storage and I/O

In a Lustre file system, a file stored on the MDT points to one or more objects associated with a data file, as shown in FIGURE 1-3. Each object contains data and is stored on an OST. If the MDT file points to one object, all the file data is stored in that object. If the 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 Lustre, see Section 1.3.1, "Lustre File System and Striping" on page 1-11.)

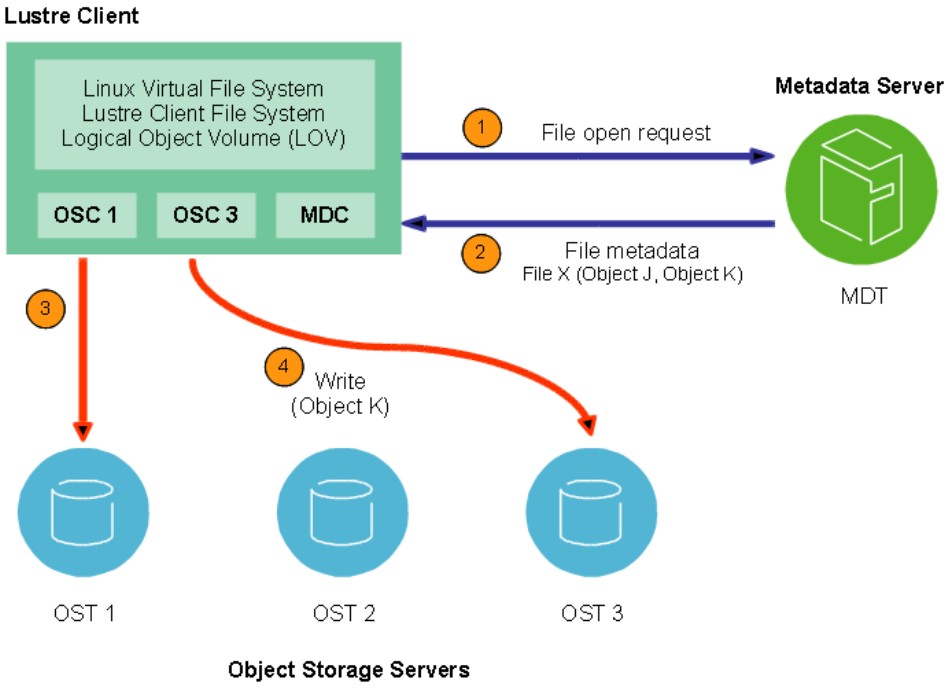
In FIGURE 1-3, each filename points to an inode. The inode contains all of the file attributes, such as owner, access permissions, Lustre striping layout, access time, and access control. Multiple filenames may point to the same inode.

FIGURE 1-3 MDT file points to objects on OSTs containing file data



When a client opens a file, the `file open` operation transfers the file layout from the MDS to the client. 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.

FIGURE 1-4 File open and file I/O in Lustre



Each file on the MDT contains the layout of the associated data file, including the OST number and object identifier. Clients request the file layout from the MDS and then perform file I/O operations by communicating directly with the OSSs that manage that 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 minimium 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 exeeds the bandwidth of a single OST. The ability to stripe is also useful when a single OST does not have anough free space to hold an entire file. For more information about benefits and drawbacks of file striping, see Section 18.2, “Lustre File Striping Considerations” on page 18-2.

Striping allows segments or “chunks” of data in a file to be stored on different OSTs, as shown in FIGURE 1-5. 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`.

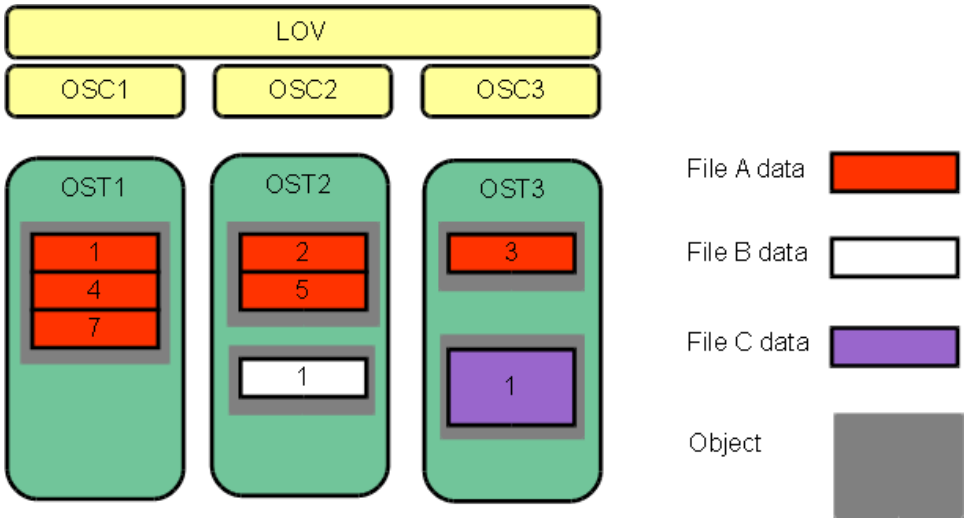
Each object contains a chunk of data from the file. When the chunk of data being written to a particular object exceeds the `stripe_size`, the next chunk of data in the file is stored on the next object.

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`)” on page 18-4.

In FIGURE 1-5, 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 is a sparse file that is missing chunk 6.

FIGURE 1-5 File striping pattern across three OSTs for three different data files



The maximum file size is not limited by the size of a single target. Lustre can stripe files across multiple objects (up to 160), and each object can be up to 2 TB in size. This leads to a maximum file size of 320 TB. (Note that Lustre itself can support files up to 2^64 bytes depending on the backing storage used by OSTs.)

Athough a single file can only be striped over 160 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 160 servers. On systems with more than 160 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 Striping and Free Space.

CHAPTER 2

Understanding Lustre Networking (LNET)

This chapter introduces Lustre Networking (LNET) and includes the following sections:

2.1 Introducing LNET

In a cluster with a Lustre file system, the system network connecting the servers and the clients is implemented using Lustre Networking (LNET), which provides the communication infrastructure required by the Lustre file system.

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. 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 such as InfiniBand or Myrinet MX
- Support for many commonly-used network types such as InfiniBand and TCP/IP
- High availability and recovery features enabling transparent recovery in conjunction with failover servers
- Simultaneous availability of multiple network types with routing between them.

LNET provides end-to-end throughput over Gigabit Ethernet (GigE) networks in excess of 100 MB/s, throughput up to 1.5 GB/s over InfiniBand double data rate (DDR) links, and throughput over 1 GB/s across 10GigE interfaces.

Lustre can use bonded networks ,such as bonded Ethernet networks, when the underlying network technology supports bonding. For more information, see Chapter 7: Understanding Lustre Networking (LNET).

2.3 Supported Network Types

LNET 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 Lustre

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

3.1 What is Failover?

A computer system is "highly available" when the services it provides are available with minimal downtime. In a highly-available system, if a failure condition occurs, such as the loss of a server or a network or software fault, the system's services continue without interruption. Generally, we measure availability by 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*, should be automatic 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 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 – Lustre 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 0+1.

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 Lustre.

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 Lustre, see Chapter 11: Configuring Lustre Failover.

HA software is responsible for detecting failure of the primary Lustre server node and controlling the failover. Lustre 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.

Typically, Lustre MDSs are configured as an active/passive pair, while OSSs are 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.

3.2 Failover Functionality in Lustre

The failover functionality provided in Lustre 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.

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

- For MDT failover, two MDSs are configured to serve the same MDT. Only one MDS node can serve an MDT at a time.
- For OST failover, multiple OSS nodes are 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.

To add a failover partner to a Lustre configuration, the `--failnode` option is used. This can be done at creation time (using `mkfs.lustre`) or later when the Lustre system is active (using `tunefs.lustre`). For explanations of these utilities, see Section 36.14, “mkfs.lustre” on page 36-28 and Section 36.18, “tunefs.lustre” on page 36-38.

Lustre failover capability can be used to upgrade the Lustre software between successive minor versions without cluster downtime. For more information, see Chapter 16: Upgrading Lustre.

For information about configuring failover, see Chapter 11: Configuring Lustre Failover.

Note – Failover functionality in Lustre is provided 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.

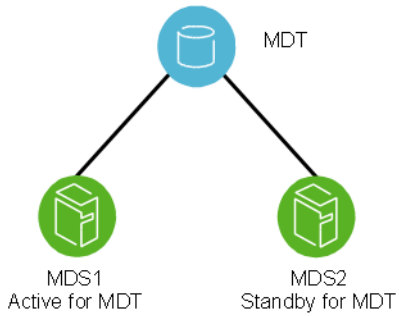
Caution – OST failover functionality does not protect against corruption caused by a disk failure. If the storage media (i.e., physical disk) used for an OST fails, Lustre cannot recover it. We strongly recommend that some form of RAID be used for OSTs. Lustre functionality assumes that the storage is 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. 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 an MDT

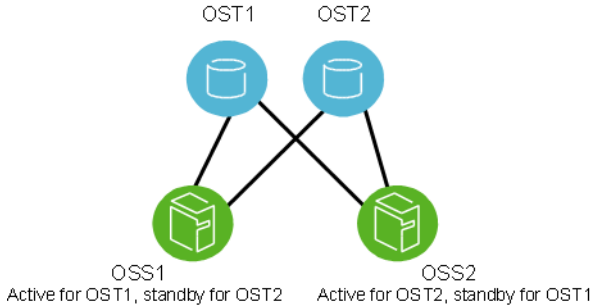


3.2.2 OST Failover Configuration (Active/Active)

OSTs are usually configured in a load-balanced, active/active failover configuration. A failover cluster is built from two OSSs as shown in FIGURE 3-2.

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

FIGURE 3-2 Lustre failover configuration for 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.

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

Setting Up a Lustre File System

Configuring Storage on a Lustre File System *(optional)*

Setting Up Network Interface Bonding *(optional)*

Installing the Lustre Software

Configuring Lustre Networking (LNET) *(optional)*

Configuring Lustre

Configuring Lustre Failover *(optional)*

For more information about required and optional steps to installing and configuring Lustre, proceed to Chapter 4: Installation Overview.

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

4.1 Steps to Installing Lustre

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: Setting Up a Lustre File System - 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 Lustre.**

   See Chapter 10: Configuring Lustre - 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 Lustre Failover - Describes how to configure Lustre failover using the Heartbeat cluster infrastructure daemon.

CHAPTER 5

Setting Up a Lustre File System

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

5.1 Hardware Considerations

Lustre can work with 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 Lustre 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 2.x. filesystems 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). It is then formatted by Lustre as a file system. Lustre OSS and MDS servers read, write and modify data in the format imposed by the file system.

Lustre 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 minimium 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 MDT Storage Hardware Considerations

The data access pattern for MDS storage is 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.

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 16 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 like InfiniBand 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.

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 Section 18.3, "Setting the File Layout/Striping Configuration (`lfs setstripe`)" on page 18-4.

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 Lustre. Additionally, Lustre reserves up to 400 MB on each OST for journal use and a small amount of space outside the journal to store accounting data for Lustre. 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 MDS/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 4 KB per inode on the MDT, which is the default value. Attached storage required for Lustre 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:

4 KB/inode * 40 million inodes = 160 GB

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

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.

5.2.2 Determining OSS/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. Lustre 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 5.3.3, “Setting the Number of Inodes for an OST” on page 5-8 for more details.

5.3 Setting File System Formatting Options

To override the default formatting options for any of the Lustre backing file systems, use this argument to `mkfs.lustre` to pass formatting options to the backing `mkfs:`

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

For other options to format backing ldiskfs filesystems, see the Linux man page for `mke2fs(8)`.

5.3.1 Setting the Number of Inodes for the MDS

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 MDT inode ratio is one inode for every 4096 bytes of file system space. To override the inode ratio, use the following option:

```
-i <bytes per inode>
```

For example, use the following option to create one inode per 2048 bytes of file system space.

```
--mkfsoptions="-i 2048"
```

To avoid `mke2fs` creating an unusable file system, do not specify the `-i` option with an inode ratio below one inode per 1024 bytes. Instead, specify an absolute number of inodes, using this option:

```
-N <number of inodes>
```

For example, by default, a 2 TB MDT will have 512M inodes. The largest currently-supported file system size is 16 TB, which would hold 4B inodes, the maximum possible number of inodes in a ldiskfs file system. With an MDS inode ratio of 1024 bytes per inode, a 2 TB MDT would hold 2B inodes, and a 4 TB MDT would hold 4B inodes.

5.3.2 Setting the Inode Size for the MDT

Lustre uses "large" inodes on backing file systems to efficiently store Lustre metadata with each file. On the MDT, each inode is at least 512 bytes in size (by default), while on the OST each inode is 256 bytes in size.
The backing ldiskfs file system also needs sufficient space for other metadata like the journal (up to 400 MB), bitmaps and directories and a few files that Lustre uses to maintain cluster consistency.

To specify a larger inode size, use the `-I <inodesize>` option. We recommend you do NOT specify a smaller-than-default inode size, as this can lead to serious performance problems; and you cannot change this parameter after formatting the file system. The inode ratio must always be larger than the inode size.

5.3.3 Setting the Number of Inodes for an OST

When formatting OST file systems, it is normally advantageous to take local file system usage into account. Try to minimize the number of inodes on each OST, while keeping enough margin for potential variance in future usage. This helps reduce the format and file system check time, and makes more space available for data.
The current default is to create one inode per 16 KB of space in the OST file system, but in many environments, this is far too many inodes for the average file size. As a good rule of thumb, the OSTs should have at least:

```
num_ost_inodes =
   4 * <num_mds_inodes> * <default_stripe_count> / <number_osts>
```

You can specify the number of inodes on the OST file systems using the following option to the `--mkfs` option:

```
-N <num_inodes>
```

Alternately, if you know the average file size, then you can specify the OST inode count for the OST file systems using:

```
-i <average_file_size / (number_of_stripes * 4)>
```

For example, if the average file size is 16 MB and there are, by default 4 stripes per file, then `--mkfsoptions='-i 1048576'` would be appropriate.

Note – In addition to the number of inodes, file system check time on OSTs is affected by a number of other variables: size of the file system, number of allocated blocks, distribution of allocated blocks on the disk, disk speed, CPU speed, and amount of RAM on the server. Reasonable file system check times (without serious file system problems), are expected to take five and thirty minutes per TB.

For more details on formatting MDT and OST file systems, see Section 6.4, "Formatting Options for RAID Devices" on page 6-3.

5.3.4 File and File System Limits

TABLE 5-1 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 Lustre (see Chapter 29: Installing Lustre from Source Code). In these cases, the indicated limit was used for Lustre testing.

TABLE 5-1 File and file system limits

Limit	Value	Description
Maximum Stripe Count	160	This limit is hard-coded, but is near the upper limit imposed by the underlying ldiskfs file system.
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	2 TB	The amount of data that can be stored in a single object. The ldiskfs limit of 2TB for a single file applies. Lustre allows 160 stripes of 2 TB each.
Maximum number of OSTs	8150	The maximum number of OSTs is a constant that can be changed at compile time. Lustre has been tested with up to 4000 OSTs.
Maximum number of MDTs	1	Maximum of 1 MDT per file system, but a single MDS can host multiple MDTs, each one for a separate file system.
Maximum number of clients	131072	The number of clients is a constant that can be changed at compile time.

TABLE 5-1 File and file system limits

Limit	Value	Description
Maximum size of a file system	64 PB	Each OST or MDT can have a file system up to 16 TB, regardless of whether 32-bit or 64-bit kernels are on the server. You can have multiple OST file systems on a single OSS node.
Maximum file size	16 TB on 32-bit systems 320 TB on 64-bit systems	Individual files have a hard limit of nearly 16 TB on 32-bit systems imposed by the kernel memory subsystem. On 64-bit systems this limit does not exist. Hence, files can be 64-bits in size. Lustre imposes an additional size limit of up to the number of stripes, where each stripe is 2 TB. A single file can have a maximum of 160 stripes, which gives an upper single file limit of 320 TB 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	Lustre 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 is tested with ten million files in a single directory.
Maximum number of files in the file system	4 billion	The ldiskfs file system imposes an upper limit of 4 billion inodes. By default, the MDS file system is formatted with 4 KB of space per inode, meaning 512 million inodes per file system of 2 TB. This can be increased initially, at the time of MDS file system creation. For more information, see Section 5.3, "Setting File System Formatting Options" on page 5-7.

TABLE 5-1 File and file system limits

Limit	Value	Description
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 Lustre file systems	None	Lustre 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.

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 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 5.2.1, "Determining MDS/MDT Space Requirements" on page 5-6, the OSS requirements include:

- **Service threads**: The service threads on the OSS node pre-allocate a 1 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 Linux, 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. as shown in 5.2.3.1.

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 (1 MB * 512 threads)	=	512 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 + filesystem metadata TOTAL = 3520MB

Per OSS DLM locks + filesystem 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 Lustre

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

A typical Lustre 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.conf` or `/etc/modprobe.conf.local` (depending on your Linux distribution).

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

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

   A network is a group of nodes that communicate directly with one another. Lustre 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'll 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 `ip2nets as` 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 `modprobe.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 perforance on RAID, and includes the following sections:

- Selecting Storage for the MDT and OSTs
- Reliability Best Practices
- Performance Tradeoffs
- Formatting Options for RAID Devices
- Connecting a SAN to a Lustre File System

Note – ***It is strongly recommended that hardware RAID be used with Lustre.*** Lustre currently does not support any redundancy at the file system level and RAID is required to protect agains 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 2PB 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 the 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" on page 5-7.

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 *<chunkblocks>*, such that *<stripe_width_blocks>* 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_blocks>"
```

Example:

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 1MB.

```
--mkfsoptions "<other options> -E stride=<chunk_blocks>, \
  stripe_width=<stripe_width_blocks>"...
```

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.

Lustre's 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.luster` 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 --ost --mgsnode=mds@osib \
   --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 without Lustre, clients allocate and lock blocks or inodes individually as they are updated. The Lustre design avoids the high contention that some of these blocks and inodes may have.
- Lustre 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:

- Network Interface Bonding Overview
- Requirements
- Bonding Module Parameters
- Setting Up Bonding
- Configuring Lustre with Bonding
- 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 Linux. 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. For Lustre, 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 in:**

   ```
   /etc/sysconfig/network-scripts/ # 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.conf. Start the slave interfaces by your normal network method.**

   ```
   # vi /etc/modprobe.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 `modprobe.conf` are required.

The examples below are from RedHat systems. 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 `modprobe.conf` entries for bonding Ethernet interfaces `eth1` and `eth2` to `bond0`:

```
# cat /etc/modprobe.conf
alias eth0 8139too
alias scsi_hostadapter sata_via
alias scsi_hostadapter1 usb-storage
alias snd-card-0 snd-via82xx
options snd-card-0 index=0
options snd-via82xx index=0
alias bond0 bonding
options bond0 mode=balance-alb miimon=100
options lnet networks=tcp
alias eth1 via-rhine

# 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 Lustre with Bonding

Lustre uses the IP address of the bonded interfaces and requires no special configuration. It treats the bonded interface 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. It includes:

- Preparing to Install the Lustre Software
- Lustre Installation Procedure

For hardware/system requirements, see Chapter 5: Setting Up a Lustre File System.

8.1 Preparing to Install the Lustre Software

If you are using a supported Linux distribution and architecture, you can install Lustre from downloaded packages (RPMs). For a list of supported configurations, see the topic Lustre_2.0 on the Lustre wiki.

A list of supported Lustre LNET drivers can be found in Chapter 2: Understanding Lustre Networking (LNET).

If you are not using a supported configuration, you can install Lustre directly from the source code. For more information on this installation method, see Chapter 29: Installing Lustre from Source Code.

8.1.1 Required Software

To install Lustre, the following are required:

- *(On Linux servers only)* Linux kernel patched with Lustre-specific patches for your platform and architecture. A Linux patched kernel can be installed on a client if it is desireable to use the same kernel on all nodes, but this is not required.
- Lustre modules compiled for the Linux kernel (see TABLE 8-1 for which packages are required for servers and clients in your configuration).
- Lustre utilities required for configuring Lustre (see TABLE 8-1 for which packages are required for servers and clients in your configuration).
- *(On Linux servers only)* `e2fsprogs` package containing Lustre-specific tools (`e2fsck` and `lfsck`) used to repair a backing file system. This replaces the existing e2fsprogs package but provides complete e2fsprogs functionality
- *(Optional)* Network-specific kernel modules and libraries such as the Lustre-specific OFED package required for an InfiniBand network interconnect.

At least one Lustre RPM must be installed on each server and on each client in a Lustre file system. TABLE 8-1 lists required Lustre packages and indicates where they are to be installed. Some Lustre packages are installed on Lustre servers (MGS, MDS, and OSSs), some are installed on Lustre clients, and some are installed on all Lustre nodes

.

TABLE 8-1 Lustre required packages, descriptions and installation guidance

Lustre Package	Description	Install on servers*	Install on clients
Lustre patched kernel RPMs:			
kernel-<ver>_lustre.<ver>	For OEL 5 or RHEL 5 server platform	X*	
kernel-ib-<ver>	Lustre OFED package. Required only if the network interconnect is InfiniBand.	X*	X
Lustre module RPMs:			
lustre-modules-<ver>	For Lustre-patched kernel.	X*	
lustre-client-modules-<ver>	For clients.		X
Lustre utilities:			
lustre-<ver>	Lustre utilities package. This includes userspace utilities to configure and run Lustre.	X*	
lustre-client-<ver>	Lustre utilities for clients.		X
lustre-ldiskfs-<ver>	Lustre-patched backing file system kernel module package for the ldiskfs file system.	X	
e2fsprogs-<ver>	Utilities package used to maintain the ldiskfs backing file system.	X	

* Installing a patched kernel on a client node is not required. However, if a client node will be used as both a client and a server, or if you want to install the same kernel on all nodes for any reason, install the server packages designated with an asterisk (*) on the client node.

In all supported Lustre installations, a patched kernel must be run on each server, including the the MGS, the MDS, and all OSSs. Running a patched kernel on a Lustre client is only required if the client will be used for multiple purposes, such as running as both a client and an OST or if you want to use the same kernel on all nodes.

Lustre RPM packages are available on the Lustre download site. They must be installed in the order described in Section 8.2, "Lustre Installation Procedure" on page 8-6.

8.1.1.1 Network-specific kernel modules and libraries

Network-specific kernel modules and libraries may be needed such as the Lustre-specific OFED package required for an InfiniBand network interconnect.

8.1.1.2 Lustre-Specific Tools and Utilities

Several third-party utilities are must be installed on servers:

- **e2fsprogs**: Lustre requires a recent version of e2fsprogs that understands extents. Use `e2fsprogs-1.41-10-sun2` or later, available at:

  http://downloads.lustre.org/public/tools/e2fsprogs/

  A quilt patchset of all changes to the vanilla `e2fsprogs` is available in `e2fsprogs-{version}-patches.tgz`.

Note – The Lustre-patched `e2fsprogs` utility is only required on machines that mount backend (ldiskfs) file systems, such as the OSS, MDS and MGS nodes. It does not need to be loaded on clients.

- **Perl** - Various userspace utilities are written in Perl. Any recent version of Perl will work with Lustre.

8.1.1.3 (Optional) High-Availability Software

If you plan to enable failover server functionality with Lustre (either on an OSS or the MDS), you must add high-availability (HA) software to your cluster software. For more information, see Section 11.2, "Setting up High-Availability (HA) Software with Lustre" on page 11-3.

8.1.1.4 (Optional) Debugging Tools and Other Optional Packages

A variety of optional packages are provided on the Lustre download site. These include debugging tools, test programs and scripts, Linux kernel and Lustre source code, and other packages.

For more information about debugging tools, see the topic Debugging Lustre on the Lustre wiki.

8.1.2 Environmental Requirements

Make sure the following environmental requirements are met before installing Lustre:

- ***(Required)*** **Disable Security-Enhanced Linux (SELinux) on servers and clients**. Lustre does not support SELinux. Therefore, disable the SELinux system extension on all Lustre nodes and make sure other security extensions, like Novell AppArmorand network packet filtering tools (such as iptables) do not interfere with Lustre. See Step 3 in the Section 8.2, "Lustre Installation Procedure" on page 8-6 below.
- ***(Required)*** **Maintain uniform user and group databases on all cluster nodes**. Use the same user IDs (UID) and group IDs (GID) on all clients. If use of supplemental groups is required, verify that the `group_upcall` requirements have been met. See Section 33.1, "User/Group Cache Upcall" on page 33-2.
- ***(Recommended)*** **Provide remote shell access to clients**. Although not strictly required to run Lustre, we recommend 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**. Lustre 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, for example, by 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.

8.2 Lustre Installation Procedure

Caution – *Before installing Lustre, back up ALL data.* Lustre contains kernel modifications which interact with storage devices and may introduce security issues and data loss if not installed, configured or administered properly.

Use this procedure to install Lustre from RPMs.

1. **Verify that all Lustre installation requirements have been met.**

   For more information on these prerequisites, see:

   - Hardware requirements in Chapter 5: Setting Up a Lustre File System
   - Software and environmental requirements in Section 8.1, "Preparing to Install the Lustre Software" on page 8-2

2. **Download the Lustre RPMs.**

   a. **On the Lustre download site, select your platform.**

      The files required to install Lustre (kernels, modules and utilities RPMs) are listed for the selected platform.

   b. **Download the required files.**

3. **Install the Lustre packages on the servers.**

   a. **Refer to TABLE 8-1 to determine which Lustre packages are to be installed on servers for your platform and architecture.**

      Some Lustre packages are installed on the Lustre servers (MGS, MDS, and OSSs). Others are installed on Lustre clients.

      Lustre packages must be installed in the order specified in the following steps.

   b. **Install the kernel, modules and ldiskfs packages.**

      Use the `rpm -ivh` command to install the kernel, module and ldiskfs packages.

Note – It is not recommended that you use the `rpm -Uvh` command to install a kernel, because this may leave you with an unbootable system if the new kernel doesn't work for some reason.

For example, the command in the following example would install required packages on a server with Infiniband networking

```
$ rpm -ivh kernel-<ver>_lustre-<ver> kernel-ib-<ver> \
  lustre-modules-<ver> lustre-ldiskfs-<ver>
```

c. Update the bootloader (`grub.conf` or `lilo.conf`) configuration file as needed.

i. Verify that the bootloader configuration file has been updated with an entry for the patched kernel.

Before you can boot to a new distribution or kernel, there must be an entry for it in the bootloader configuration file. Often this is added automatically when the kernel RPM is installed.

ii. Disable security-enhanced (SE) Linux on servers and clients by including an entry in the bootloader configuration file as shown below:

```
selinux=0
```

d. Install the utilities/userspace packages.

Use the `rpm -ivh` command to install the utilities packages. For example:

```
$ rpm -ivh lustre-<ver>
```

e. Install the e2fsprogs package.

Use the `rpm -ivh` command to install the `e2fsprogs` package. For example:

```
$ rpm -ivh e2fsprogs-<ver>
```

If e2fsprogs is already installed on your Linux system, install the Lustre-specific e2fsprogs version by using `rpm -Uvh` to upgrade the existing `e2fsprogs` package. For example:

```
$ rpm -Uvh e2fsprogs-<ver>
```

Note – The `rpm` command options `--force` or `--nodeps` should not be used to install or update the Lustre-specific e2fsprogs package. If errors are reported, file a bug (for instructions see the topic Reporting Bugs on the Lustre wiki.

---

f. (Optional) To add optional packages to your Lustre file system, install them now.

Optional packages include file system creation and repair tools, debugging tools, test programs and scripts, Linux kernel and Lustre source code, and other packages. A complete list of optional packages for your platform is provided on the Lustre download site.

4. **Install the Lustre packages on the clients.**

   a. **Refer to TABLE 8-1 to determine which Lustre packages are to be installed on clients for your platform and architecture.**

   b. **Install the module packages for clients.**

      Use the `rpm -ivh` command to install the `lustre-client` and `lustre-client-modules-<ver>` packages. For example:

      ```
      $ rpm -ivh lustre-client-modules-<ver> kernel-ib-<ver>
      ```

   c. **Install the utilities/userspace packages for clients.**

      Use the `rpm -ivh` command to install the utilities packages. For example:

      ```
      $ rpm -ivh lustre-client
      ```

   d. **Update the bootloader (`grub.conf` or `lilo.conf`) configuration file as needed.**

      i. **Verify that the bootloader configuration file has been updated with an entry for the patched kernel.**

         Before you can boot to a new distribution or kernel, there must be an entry for it in the bootloader configuration file. Often this is added automatically when the kernel RPM is installed.

      ii. **Disable security-enhanced (SE) Linux on servers and clients by including an entry in the bootloader configuration file as shown below:**

         ```
         selinux=0
         ```

5. **Reboot the patched clients and the servers.**

   a. **If you applied the patched kernel to any clients, reboot them.**

      Unpatched clients do not need to be rebooted.

   b. **Reboot the servers.**

To configure LNET, go next to Chapter 9: Configuring Lustre Networking (LNET). If default settings will be used for LNET, go to Chapter 10: Configuring Lustre.

CHAPTER 9

Configuring Lustre Networking (LNET)

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

Note – Configuring LNET is optional.

LNET will, by default, use the first TCP/IP interface it discovers on a system (`eth0`). 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.

9.1 Overview of LNET Module Parameters

LNET kernel module (`lnet`) parameters specify how LNET is to be configured to work with Lustre, including which NICs will be configured to work with Lustre and the routing to be used with Lustre.

Parameters for `lnet` are specified in the `modprobe.conf` or `modules.conf` file (depending on your Linux distribution) in one or more entries with the syntax:

```
options lnet <parameter>=<parameter value>
```

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

- `networks` - Specifies the networks 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, "Setting the LNET Module `networks` Parameter" on page 9-4 and Section 9.3, "Setting the LNET Module `ip2nets` Parameter" on page 9-6 for more details.

To set up routing between networks, use:

- `routes` - Lists networks and the NIDs of routers that forward to them.

See Section 9.4, "Setting the LNET Module `routes` Parameter" on page 9-7 for more details.

A router checker can be configured to enable Lustre nodes to detect router health status, avoid routers that appear dead, and reuse those that restore service after failures. See Section 9.6, "Configuring the Router Checker" on page 9-8 for more details.

For a complete reference to the LNET module parameters, see Section 35.2.1, "LNET Options" on page 35-3.

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.

9.1.1 Using a Lustre Network Identifier (NID) to Identify a Node

A Lustre network identifier (NID) is used to uniquely identify a Lustre network endpoint by node ID and network type. The format of the NID is:

<network id>@*<network type>*

Examples are:

```
10.67.73.200@tcp0
10.67.75.100@o2ib
```

The first entry above identifes a TCP/IP node, while the second entry identifies an InfiniBand node.

When a `mount` command is run on a client, the client uses the NID of the MDS to retrieve configuration information. If an MDS has more than one NID, the client should use the appropriate NID for its local network.

To determine the appropriate NID to specify in the `mount` command, use the `lctl` command. *To display MDS NIDs*, run on the MDS :

```
lctl list_nids
```

To determine if a client can reach the MDS using a particular NID, run on the client:

```
lctl which_nid <MDS NID>
```

9.2 Setting the LNET Module networks Parameter

If a node has more than one network interface, you'll typically want to dedicate a specific interface to Lustre. You can do this by including an entry in the modprobe.conf file on the node that sets the LNET module networks parameter:

```
options lnet networks=<comma-separated list of networks>
```

This example specifies that a Lustre node will use a TCP/IP interface and an InfiniBand interface:

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

This example specifies that the Lustre node will use the TCP/IP interface eth1:

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

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, Lustre chooses the best route based on the hop count. Once the network connection is established, Lustre 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.

Note – LNET lines in modprobe.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 Multihome Server Example

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

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

To set the `networks` option for this example:

- On each server, `svr1` and `svr2`, include the following line in the `modprobe.conf` file:
  ```
  options lnet networks=tcp0(eth0),tcp1(eth1),o2ib
  ```
- 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 `modprobe.conf` file.
- On the InfiniBand clients, include the following line in the `modprobe.conf` file:
  ```
  options lnet networks=o2ib
  ```

Note – By default, Lustre ignores the loopback (`lo0`) interface. Lustre does not ignore IP addresses aliased to the loopback. If you alias IP addresses to the loopback interface, you must specify all Lustre networks using the LNET networks parameter.

Note – 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 of Linux, not Lustre. In this case, network interface bonding should be used. For more information about network interface bonding, see Chapter 7: Setting Up Network Interface Bonding.

9.3 Setting the LNET Module ip2nets Parameter

The ip2nets option is typically used when a single, universal modprobe.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 modprobe.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 modprobe.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 Setting the LNET Module routes Parameter

The LNET module routes parameter is used to identify routers in a Lustre configuration. These parameters are set in modprob.conf on each Lustre node.

The LNET routes parameter specifies a colon-separated list of router definitions. Each route is defined as a network number, followed by a list of routers:

routes=<*net type*> <*router NID(s)*>

This example specifies bi-directional routing in which TCP clients can reach Lustre resources on the IB networks and IB servers can access the TCP networks:

```
options lnet 'ip2nets="tcp0 192.168.0.*; \
  o2ib0(ib0) 132.6.1.[1-128]"' 'routes="tcp0 132.6.1.[1-8]@o2ib0; \
  o2ib0 192.168.0.[1-8]@tcp0"'
```

All LNET routers that bridge two networks are equivalent. They are not configured as primary or secondary, and the load is balanced across all available routers.

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.

9.4.1 Routing Example

On the clients, place the following entry in the modprobe.conf file
```
lnet networks="tcp" routes="o2ib0 192.168.0.[1-8]@tcp0"
```

On the router nodes, use:
```
lnet networks="tcp o2ib" forwarding=enabled
```

On the MDS, use the reverse as shown below:
```
lnet networks="o2ib0" rountes="tcp0 132.6.1.[1-8]@o2ib0"
```

To start the routers, run:

```
modprobe lnet
lctl network configure
```

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 23: Testing Lustre Network Performance (LNET Self-Test).

9.6 Configuring the Router Checker

In a Lustre configuration in which different types of networks, such as a TCP/IP network and an Infiniband network, are connected by routers, a router checker can be run on the clients and servers in the routed configuration to monitor the status of the routers. In a multi-hop routing configuration, router checkers can be configured on routers to monitor the health of their next-hop routers.

A router checker is configured by setting `lnet` parameters in `modprobe.conf` by including an entry in this form:

```
options lnet <router checker parameter>=<parameter value>
```

The router checker parameters are:

- `live_router_check_interval` - Specifies a time interval in seconds after which the router checker will ping the live routers. The default value is `0`, meaning no checking is done. To set the value to `60`, enter:

  ```
  options lnet live_router_check_interval=60
  ```

- `dead_router_check_interval` - Specifies a time interval in seconds after which the router checker will check for dead routers. The default value is `0`, meaning no checking is done. To set the value to `60`, enter:

  ```
  options lnet dead_router_check_interval=60
  ```

- `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
  ```

- `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.

If a router is marked “up” and 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.

9.7 Best Practices for LNET Options

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

Escaping commas with quotes

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.

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 Lustre

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

10.1 Configuring a Simple Lustre File System

A Lustre system can be set up in a variety of configurations by using the administrative utilities provided with Lustre. The procedure below shows how to 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: Setting Up a Lustre File System.
- ***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 Lustre 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 23: 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 Lustre and ensure you are getting the best possible performance with your installation. For information about running `sgpdd_survey`, see Section 24.2, "Testing I/O Performance of Raw Hardware (sgpdd_survey)" on page 24-3.

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 <block device name>
   ```

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

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

```
mkfs.lustre --fsname=<fsname> --mgs <block device name>
```

---

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

   ```
   mount -t lustre <block device name> <mount point>
   ```

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

---

3. **Create the OST. On the OSS node, run:**

   ```
   mkfs.lustre --ost --fsname=<fsname> --mgsnode=<NID> <block device name>
   ```

   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: Setting Up a Lustre File System.

   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.

   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 Section 20.1, “Lustre Failover and Multiple-Mount Protection” on page 20-2.

Note – Lustre currently supports block devices up to 16 TB on OEL 5/RHEL 5 (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. If the size is significantly larger than 16 TB, you should reconfigure the storage into devices smaller than 16 TB. We recommend that you not place partitions on top of RAID 5/6 block devices due to negative impacts on performance.

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

```
mount -t lustre <block device name> <mount point>
```

Note – To create additional OSTs, repeat Step 3 and Step 4.

5. **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 5.

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

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. ***(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 24.3, "Testing OST Performance (obdfilter_survey)" on page 24-6.
   - `ost_survey` - Performs I/O against OSTs to detect anomalies between otherwise identical disk subsystems. For details, see Section 24.4, "Testing OST I/O Performance (ost_survey)" on page 24-13.

10.1.1 Simple Lustre Configuration Example

To see the steps in a simple Lustre configuration, follow this example in which a combined MGS/MDT and two OSTs are created. 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
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	`mdt1`	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 `mdt1` block device (`/dev/sdb`) on the MGS/MDS node
First OSS node		
OSS node	`oss1`	First OSS node in Lustre file system `temp`
OST	`ost1`	First OST in Lustre file system `temp`
block device	`/dev/sdc`	Block device for the first OSS node (`oss1`)
mount point	`/mnt/ost1`	Mount point for the `ost1` block device (`/dev/sdc`) on the `oss1` node
Second OSS node		
OSS node	`oss2`	Second OSS node in Lustre file system `temp`
OST	`ost2`	Second OST in Lustre file system `temp`
block device	`/dev/sdd`	Block device for the second OSS node (`oss2`)
mount point	`/mnt/ost2`	Mount point for the `ost2` block device (`/dev/sdd`) on the `oss2` 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 /dev/sdb
   ```

   This command generates this output:

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

   checking for existing Lustre data: not found
   device size = 16MB
   2 6 18
   formatting backing filesystem ldiskfs on /dev/sdb
              target nametemp-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_group_upcall()) \
   temp-MDT0000: group upcall set to /usr/sbin/l_getgroups
   Lustre: temp-MDT0000.mdt: set parameter \
   group_upcall=/usr/sbin/l_getgroups
   Lustre: Server temp-MDT0000 on device /dev/sdb has started
   ```

3. **Create and mount ost1.**

   In this example, the OSTs (ost1 and ost2) are being created on different OSSs (oss1 and oss2 respectively).

   a. Create ost1. On oss1 node, run:

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

   The command generates this output:

```
            Permanent disk data:
Target:             temp-OSTffff
Index:              unassigned
Lustre FS:          temp
Mount type:         ldiskfs
Flags:              0x72
(OST needs_index 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-OSTffff
            4k blocks     0
            options       -I 256 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L temp-OSTffff  -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/sdc /mnt/ost1
```

   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 ost2.**

   a. **Create ost2. On oss2 node, run:**

```
[root@oss2 /]# mkfs.lustre --ost --fsname=temp --mgsnode=
10.2.0.1@tcp0 /dev/sdd
```

   The command generates this output:

```
            Permanent disk data:
Target:         temp-OSTffff
Index:          unassigned
Lustre FS:      temp
Mount type:     ldiskfs
Flags:          0x72
(OST needs_index 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-OSTffff
        4k blocks      0
        options        -I 256 -q -O dir_index,uninit_groups -F
mkfs_cmd = mkfs.ext2 -j -b 4096 -L temp-OSTffff  -I 256 -q -O
dir_index,uninit_groups -F /dev/sdc
Writing CONFIGS/mountdata
```

   b. **Mount ost2 on the OSS on which it was created. On oss2 node, run:**

```
root@oss2 /] mount -t lustre /dev/sdd /mnt/ost2
```

   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
```

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_UUID800.0G     400.0M      799.6G   0% /lustre[OST:0]
   temp-OST0001_UUID800.0G     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 (0s). 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 4 above. For mounting additional clients, repeat Step 5 for each client.

10.2.2 Changing Striping Defaults

The default settings for the file layout stripe pattern are shown in TABLE 10-1.

TABLE 10-1

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 Section 18.3, "Setting the File Layout/Striping Configuration (`lfs setstripe`)" on page 18-4 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 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 36: System Configuration Utilities on the Lustre wiki.

The `lfs` utility is usful for configuring and querying a variety of options related to files. For more information, see Section 32.1, "lfs" on page 32-2.

Note – Some sample scripts are included in the directory where Lustre 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 Lustre Failover

This chapter describes how to configure Lustre failover using the Heartbeat cluster infrastructure daemon. It includes:

- Creating a Failover Environment
- Setting up High-Availability (HA) Software with Lustre

Note – ***Using Lustre Failover is optional.***

11.1 Creating a Failover Environment

Lustre provides failover mechanisms only at the file system level. No failover functionality is provided for system-level components, such as node failure detection or power control, as would typically be provided in a complete failover solution. Additional tools are also needed to provide resource fencing, control and monitoring.

11.1.1 Power Management Software

Lustre failover requires power control 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, but two packages that are commonly used with Lustre are STONITH and PowerMan.

Shoot The Other Node In The HEAD (STONITH), is a set of power management tools provided with the Linux-HA package. STONITH has native support for many power control devices and is extensible. It uses expect scripts to automate control.

PowerMan, available from the Lawrence Livermore National Laboratory (LLNL), is used to control remote power 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://sourceforge.net/projects/powerman

For more information about PowerMan, go to:

https://computing.llnl.gov/linux/powerman.html

11.1.2 Power Equipment

Lustre failover also requires the use of RPC devices, which come in different configurations. Lustre server nodes may be equipped with some kind of service processor that allows remote power control. If a Lustre server node is not equipped with a service processor, then a multi-port, Ethernet-addressable RPC may be used as an alternative. For recommended products, refer to the list of supported RPC devices on the PowerMan website.

https://computing.llnl.gov/linux/powerman.html

11.2 Setting up High-Availability (HA) Software with Lustre

Lustre must be combined with high-availability (HA) software to enable a complete Lustre failover solution. Lustre can be used with several HA packages including:

- *Red Hat Cluster Manager* - For more information about setting up Lustre failover with Red Hat Cluster Manager, see the Lustre wiki topic Using Red Hat Cluster Manager with Lustre.
- *Pacemaker* - For more information about setting up Lustre failover with Pacemaker, see the Lustre wiki topic Using Pacemaker with 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:

- Lustre Monitoring
- Lustre Operations
- Lustre Maintenance
- Managing Lustre Networking (LNET)
- Upgrading Lustre
- Backing Up and Restoring a File System
- Managing File Striping and Free Space
- Managing the File System and I/O
- Managing Failover
- Configuring and Managing Quotas
- Managing Lustre Security

Tip – The starting point for administering Lustre 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 by Linux.

CHAPTER 12

Lustre Monitoring

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

- Lustre Changelogs
- Lustre Monitoring Tool
- CollectL
- 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
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.

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 <mdt_device> changelog_register
```

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

lfs changelog

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

```
lfs changelog <MDT name> [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=
[0x200000405:0x15f9:0x0] chloe.jpg
4 06UNLNK 4298404466 0x0 t=[0x200000405:0x15fa:0x0] p=
[0x200000405:0x15f9:0x0] chloe.jpg
5 07RMDIR 4298405394 0x0 t=[0x200000405:0x15f9:0x0] p=
[0x13:0x15e5a7a3:0x0] pics
```

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> <user ID> <endrec>
```

The `changelog_clear` command indicates that changelog records previous to `<endrec>` are no longer of interest to a particular user <user ID>, 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.

lctl changelog_deregister

To deregister (unregister) a changelog user, run:

```
lctl --device <mdt_device> changelog_deregister <user ID>
```

`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.

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'
```

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=
[0:0x0:0x0] mdd_obd-lustre-MDT0000-0
2 02MKDIR 19:10:21.509659173 2010.03.24 0x0 t=[0x200000420:0x3:0x0]
p=[0x61b4:0xca2c7dde: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=[0x200000420:0x3:0x0] 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=[0x200000420:0x3:0x0] hosts
```

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=[0x200000420:0x3:0x0] hosts
```

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 (cli).

```
$ lfs changelog lustre-MDT0000

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

Note – MARK records typically indicate changelog recording status changes.

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
```

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
```

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=[0x200000420:0x3:0x0] myhardlink
```

12.2 Lustre Monitoring Tool

The Lustre Monitoring Tool (LMT) is a Python-based, distributed system developed and maintained by Lawrence Livermore National Lab (LLNL)). It 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:

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

LMT questions can be directed to:

lmt-discuss@googlegroups.com

12.3 CollectL

CollectL is another tool that can be used to monitor Lustre. 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

12.4 Other Monitoring Options

A variety of standard tools are available publically.

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

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:

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:

```
$ mount -t lustre -L testfs-MDT0000 /mnt/mdt
```

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

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:

```
mount -t lustre uml1@tcp0:/shortfs /mnt/<long-file_system-name>
```

13.2 Starting Lustre

The startup order of Lustre components depends on whether you have a combined MGS/MDT or these components are separate.

- If you have a combined MGS/MDT, the recommended startup order is OSTs, then the MGS/MDT, and then clients.
- If the MGS and MDT are separate, the recommended startup order is: MGS, then OSTs, then the MDT, and then clients.

Note – If an OST is added to a Lustre file system with a combined MGS/MDT, then the startup order changes slightly; the MGS must be started first because the OST needs to write its configuration data to it. In this scenario, the startup order is MGS/MDT, then OSTs, then the clients.

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. RedHat, SuSE, Debian (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 ends in FFFF, meaning that it has yet to be assigned. The assignment takes place when the server is first started, and the disk label is updated.

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

Lustre uses two modes, failout and failover, to handle an OST that has become unreachable because it fails, is taken off the network, is unmounted, etc.

- 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, run this command:

```
$ mkfs.lustre --fsname=<fsname> --ost --mgsnode=<MGS node NID>
--param="failover.mode=failout" <block device name>
```

In this example, failout mode is specified for the OSTs on MGS `uml1`, file system `testfs`.

```
$ mkfs.lustre --fsname=testfs --ost --mgsnode=uml1 --param=
"failover.mode=failout" /dev/sdb
```

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

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

```
$ tunefs.lustre --param failover.mode=failout <OST partition>
```

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

There may be situations in which you want to run multiple file systems. This is doable, as long as you follow specific naming conventions.

By default, the `mkfs.lustre` command creates a file system named `lustre`. To specify a different file system name (limited to 8 characters), run this command:

```
mkfs.lustre --fsname=<new file system name>
```

Note – The MDT, OSTs and clients in the new file system must share the same 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:

```
mount -t lustre mgsnode:/<new fsname> <mountpoint>
```

For example, to mount a client on file system `foo` at mount point `/mnt/lustre1`, run:

```
mount -t lustre mgsnode:/foo /mnt/lustre1
```

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 – The MGS is universal; there is only one MGS per Lustre installation, not per file system.

Note – There is only one file system per MDT. Therefore, specify `--mdt --mgs` on one file system and `--mdt --mgsnode=<MGS node NID>` on the other file systems.

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/lustre1 and /mnt/lustre2.

```
mgsnode# mkfs.lustre --mgs /mnt/lustre1

mdtfoonode# mkfs.lustre --fsname=foo --mdt \
--mgsnode=mgsnode@tcp0 /mnt/lustre1

ossfoonode# mkfs.lustre --fsname=foo --ost \
--mgsnode=mgsnode@tcp0 /mnt/lustre1

ossfoonode# mkfs.lustre --fsname=foo --ost \
--mgsnode=mgsnode@tcp0 /mnt/lustre2

mdtbarnode# mkfs.lustre --fsname=bar --mdt \
--mgsnode=mgsnode@tcp0 /mnt/lustre1

ossbarnode# mkfs.lustre --fsname=bar --ost \
--mgsnode=mgsnode@tcp0 /mnt/lustre1

ossbarnode# mkfs.lustre --fsname=bar --ost \
--mgsnode=mgsnode@tcp0 /mnt/lustre2
```

To mount a client on file system foo at mount point /mnt/lustre1, run:

```
mount -t lustre mgsnode@tcp0:/foo /mnt/lustre1
```

To mount a client on file system bar at mount point /mnt/lustre2, run:

```
mount -t lustre mgsnode@tcp0:/bar /mnt/lustre2
```

13.8 Setting and Retrieving Lustre Parameters

Several options are available for setting parameters in Lustre:

- When creating a file system, use mkfs.lustre. See Section 13.8.1, “Setting Parameters with mkfs.lustre” on page 13-9 below.
- When a server is stopped, use tunefs.lustre. See Section 13.8.2, “Setting Parameters with tunefs.lustre” on page 13-9 below.
- When the file system is running, use lctl to set or retrieve Lustre parameters. See Section 13.8.3, “Setting Parameters with lctl” on page 13-9 and Section 13.8.3.4, “Reporting Current Parameter Values” on page 13-11 below.

13.8.1 Setting Parameters with mkfs.lustre

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

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

For more details about creating a file system,see Chapter 10: Configuring Lustre. For more details about `mkfs.lustre`, see Chapter 36: System Configuration Utilities.

13.8.2 Setting Parameters with tunefs.lustre

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

```
$ 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:

```
$ 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.group_upcall=NONE /dev/sda1
```

For more details about `tunefs.lustre`, see Chapter 36: System Configuration Utilities.

13.8.3 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 13.8.3.3, “Listing Parameters” on page 13-11.

For more details about the `lctl` command, see the examples in the sections below and Chapter 36: System Configuration Utilities.

13.8.3.1 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
```

13.8.3.2 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):

```
<obd|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.group_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.

13.8.3.3 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:

```
$ lctl list_param obdfilter.lustre-OST0000
```

13.8.3.4 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.

```
$ 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 number of inodes available on each OST.

```
# lctl get_param osc.*.filesfree

osc.myth-OST0000-osc-ffff88006dd20000.filesfree=217623
osc.myth-OST0001-osc-ffff88006dd20000.filesfree=5075042
osc.myth-OST0002-osc-ffff88006dd20000.filesfree=3762034
osc.myth-OST0003-osc-ffff88006dd20000.filesfree=91052
osc.myth-OST0004-osc-ffff88006dd20000.filesfree=129651
```

13.9 Specifying NIDs and Failover

If a node has multiple network interfaces, it may have multiple NIDs. When a node is specified, all of its NIDs must be listed, delimited by commas (,) so other nodes can choose the NID that is appropriate for their network interfaces. When failover nodes are specified, they are delimited by a colon (:) or by repeating a keyword (`--mgsnode=` or `--failnode=`). To obtain all NIDs from a node (while LNET is running), run:

```
lctl list_nids
```

This displays the server's NIDs (networks configured to work with Lustre).

This example has a combined MGS/MDT failover pair on uml1 and uml2, and a OST failover pair on uml3 and uml4. There are corresponding Elan addresses on uml1 and uml2.

```
uml1> mkfs.lustre --fsname=testfs --mdt --mgs \
--failnode=uml2,2@elan /dev/sda1
uml1> mount -t lustre /dev/sda1 /mnt/test/mdt
uml3> mkfs.lustre --fsname=testfs --ost --failnode=uml4 \
--mgsnode=uml1,1@elan --mgsnode=uml2,2@elan /dev/sdb
uml3> mount -t lustre /dev/sdb /mnt/test/ost0
client> mount -t lustre uml1,1@elan:uml2,2@elan:/testfs /mnt/testfs
uml1> umount /mnt/mdt
uml2> mount -t lustre /dev/sda1 /mnt/test/mdt
uml2> cat /proc/fs/lustre/mds/testfs-MDT0000/recovery_status
```

Where multiple NIDs are specified, comma-separation (for example, `uml2,2@elan`) means that the two NIDs refer to the same host, and that Lustre needs to choose the "best" one for communication. Colon-separation (for example, `uml1:uml2`) means that the two NIDs refer to two different hosts, and should be treated as failover locations (Lustre tries the first one, and if that fails, it tries the second one.)

Note – If you have an MGS or MDT configured for failover, perform these steps:

1. On the OST, list the NIDs of all MGS nodes at mkfs time.

```
OST# mkfs.lustre --fsname sunfs --ost --mgsnode=10.0.0.1
--mgsnode=10.0.0.2 /dev/{device}
```

2. On the client, mount the file system.

```
client# mount -t lustre 10.0.0.1:10.0.0.2:/sunfs /cfs/client/
```

13.10 Erasing a File System

If you want to erase a 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. **Erase the file system and, presumably, replace it with another file system, run:**

   ```
   $ mkfs.lustre –reformat --fsname spfs --mdt --mgs /dev/sda
   ```

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 --mdt \
   --mgs /dev/sda
   ```

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.

13.11 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 for the root user. In order to reclaim this space, run the following command on your OSSs:

```
tune2fs [-m reserved_blocks_percent] [device]
```

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

13.12 Replacing an Existing OST or MDS

To copy the contents of an existing OST to a new OST (or an old MDS to a new MDS), use one of these methods:

- Connect the old OST disk and new OST disk to a single machine, mount both, and use rsync to copy all data between the OST file systems.

  For example:

  ```
  mount -t ldiskfs /dev/old /mnt/ost_old
  mount -t ldiskfs /dev/new /mnt/ost_new
  rsync -aSv /mnt/ost_old/ /mnt/ost_new
  # note trailing slash on ost_old/
  ```

- If you are unable to connect both sets of disk to the same computer, use `rsync` to copy over the network using `rsh` (or `ssh` with `-e ssh`):

  ```
  rsync -aSvz /mnt/ost_old/ new_ost_node:/mnt/ost_new
  ```

- Use the same procedure for the MDS, with one additional step:

  ```
  cd /mnt/mds_old; getfattr -R -e base64 -d . > /tmp/mdsea; \
  <copy all MDS files as above>; cd /mnt/mds_new; setfattr \
  --restore=/tmp/mdsea
  ```

13.13 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 1.41.5.sun2 (23-Apr-2009)
   /dev/lustre/ost_test2: catastrophic mode - not reading inode or
   group bitmaps
   Inode: 352365   Type: regular    Mode:  0666   Flags: 0x80000
   Generation: 1574463214    Version: 0xea020000:00000000
   User:   500   Group:   500   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 = "e2 00 11 00 00 00 00 00 25 43 c1 87 00 00 00 00 a0 88 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 " (32)
   BLOCKS:
   (0-63):47968-48031
   TOTAL: 64
   ```

2. **Note the FID's EA and apply it to the osd_inode_id mapping.**

   In this example, the FID's EA is:

   ```
   e20011000000000002543c18700000000a088000000000000000000000000000000
   ```

   ```
   struct osd_inode_id {

     __u64 oii_ino; /* inode number */
     __u32 oii_gen; /* inode generation */
     __u32 oii_pad; /* alignment padding */
   };
   ```

   After swapping, you get an inode number of 0x001100e2 and generation of 0.

3. **On the MDT (as root), use debugfs to find the file associated with the inode.**

   ```
   # debugfs -c -R "ncheck 0x001100e2" /dev/lustre/mdt_test
   ```

   Here is the command output:

   ```
   debugfs 1.41.5.sun2 (23-Apr-2009)
   /dev/lustre/mdt_test: catastrophic mode - not reading inode or group bitmaps
   Inode    Pathname
   1114338  /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:

- Working with Inactive OSTs
- Finding Nodes in the Lustre File System
- Mounting a Server Without Lustre Service
- Regenerating Lustre Configuration Logs
- Changing a Server NID
- Adding a New OST to a Lustre File System
- Removing and Restoring OSTs
- Aborting Recovery
- Determining Which Machine is Serving an OST
- Changing the Address of a Failover Node

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 `lustre` has three nodes, `lustre-MDT0000`, `lustre-OST0000`, and `lustre-OST0001`.

```
cfs21:/tmp# cat /proc/fs/lustre/mgs/MGS/live/*
fsname: lustre
flags: 0x0     gen: 26
lustre-MDT0000
lustre-OST0000
lustre-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, lustre-OST0000 and lustre-OST0001, which are both active.

```
cfs21:/tmp# cat /proc/fs/lustre/lov/lustre-mdtlov/target_obd
0: lustre-OST0000_UUID ACTIVE
1: lustre-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 <MDT partition> -o nosvc <mount point>
```

The <MDT partition> variable is the combined MGS/MDT.

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 system's 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's 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 node>$ tunefs.lustre --writeconf <device>
   ```

   b. **On each OST, run:**

   ```
   <ost node>$ tunefs.lustre --writeconf <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

If you need to change the NID on the MDT or an OST, run the `writeconf` command to erase Lustre configuration information (including server NIDs), and then re-generate the system configuration using updated server NIDs.

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 (`lctl list_nids`) is correct.**

   The `lctl list_nids` command indicates which network(s) are configured to work with Lustre.

2. **Shut down the file system in this order.**

   a. **Unmount the clients.**

   b. **Unmount the MDT.**

   c. **Unmount all OSTs.**

3. **Run the** `writeconf` **command on all servers.**

   Run `writeconf` on the MDT first, and then the OSTs.

   a. **On the MDT, run:**

   ```
   <mdt node>$ tunefs.lustre --writeconf <device>
   ```

   b. **On each OST, run:**

   ```
   <ost node>$ tunefs.lustre --writeconf <device>
   ```

   c. **If the NID on the MGS was changed, communicate the new MGS location to each server. Run:**

   ```
   tunefs.lustre --erase-param --mgsnode=<new_nid(s)> --writeconf /dev/..
   ```

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, and server NIDs in the updated `list_nids` file are used.

14.6 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:**

```
$ mkfs.lustre --fsname=spfs --ost --mgsnode=mds16@tcp0 /dev/sda
$ mkdir -p /mnt/test/ost0
$ mount -t lustre /dev/sda /mnt/test/ost0
```

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:

   ```
   lfs_migrate /mnt/lustre/file
   ```

   To migrate files within the `/test` filesystem on OST0004 that are larger than 4GB in size, enter:

   ```
   lfs find /test -obd test-OST0004 -size +4G | lfs_migrate -y
   ```

   See Section 32.2, “lfs_migrate” on page 32-13 for more details.

14.7 Removing and Restoring OSTs

OSTs can be removed from and restored to a Lustre file system. Currently in Lustre, removing an OST really means that the OST is ‘deactivated’ in the file system, not permanently removed. 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

14.7.1 Removing an OST from the File System

OSTs can be removed from a Lustre file system. Currently in Lustre, removing an OST actually means that the OST is 'deactivated' from the file system, not permanently removed. A removed OST still appears in the device listing; you should not normally create a new OST with the same name.

You may want to deactivate an OST and prevent new files from being written to it in several situations:

- OST is nearing its space capacity
- Hard drive has failed and a RAID resync/rebuild is underway
- OST storage has failed permanently

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 "
   ```

   This is sample `lctl dl | grep " osc "` output:

   ```
   11 UP osc lustre-OST-0000-osc-cac94211 4ea5b30f-6a8e-55a0-7519-2f20318ebdb4 5
   12 UP osc lustre-OST-0001-osc-cac94211 4ea5b30f-6a8e-55a0-7519-2f20318ebdb4 5
   13 IN osc lustre-OST-0000-osc lustre-MDT0000-mdtlov_UUID 5
   14 UP osc lustre-OST-0001-osc lustre-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:**

   ```
   $ mdt> lctl --device <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:

   ```
   $ mdt> 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 UUID> <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>-<OST name>-*.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.

---

b. To *permanently disable* the deactivated OST, enter:

```
[mgs]# lctl conf_param {OST name}.osc.active=0
```

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:

Note – A removed OST still appears in the file system; do not create a new OST with the same name.

14.7.2 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 filesystem.

```
[oss]# mkdir -p /mnt/ost
[oss]# mount -t ldiskfs {ostdev} /mnt/ost
```

2. **Back up the OST configuration files.**

```
[oss]# tar cvf {ostname}.tar -C /mnt/ost last_rcvd \
  CONFIGS/ O/0/LAST_ID
```

3. **Unmount the OST filesystem.**

```
[oss]# umount /mnt/ost
```

14.7.3 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)" on page 17-6, or Section 17.3, "Making a File-Level Backup of an OST File System" on page 17-7 and Section 17.4, "Restoring a File-Level Backup" on page 17-9.

To replace an OST that was removed from service due to corruption or hardware failure, the file system needs to be formatted for Lustre, and the Lustre 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 {OST index} {other options} \
  {newdev}
```

2. **Mount the OST filesystem.**

```
[oss]# mkdir /mnt/ost
[oss]# mount -t ldiskfs {newdev} /mnt/ost
```

3. **Restore the OST configuration files, if available.**

```
[oss]# tar xvf {ostname}.tar -C /mnt/ost
```

4. **Recreate the OST configuration files, if unavailable.**

   Follow the procedure in Section 26.3.4, "Fixing a Bad LAST_ID on an OST" on page 26-8 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):

```
[oss2]# debugfs -c -R "dump CONFIGS/mountdata /tmp/ldd" \
  {other_osdev}
[oss2]# scp /tmp/ldd oss:/tmp/ldd
[oss]# dd if=/tmp/ldd of=/mnt/ost/CONFIGS/mountdata bs=4 count=1 \
  seek=5 skip=5
```

5. Unmount the OST filesystem.

```
[oss]# umount /mnt/ost
```

14.7.4 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 <devno> activate
[client]# lctl set_param osc.<fsname>-<OST name>-*.active=0
```

14.8 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:

```
$ mount -t lustre -L <MDT name> -o abort_recov <mount point>
```

Note – The recovery process is blocked until all OSTs are available.

14.9 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 Lustre 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.${fsname}-${OSTname}*.ost_conn_uuid
```

For example:

```
client$ lctl get_param osc.*-OST0000*.ost_conn_uuid
osc.lustre-OST0000-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
```

- OR -

```
client$ lctl get_param osc.*.ost_conn_uuid
osc.lustre-OST0000-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.lustre-OST0001-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.lustre-OST0002-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.lustre-OST0003-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
osc.lustre-OST0004-osc-f1579000.ost_conn_uuid=192.168.20.1@tcp
```

14.10 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:

```
tunefs.lustre --erase-params --failnode=<NID> <device>
```

CHAPTER 15

Managing Lustre Networking (LNET)

This chapter describes some tools for managing Lustre Networking (LNET) and includes the following sections:

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

Lustre 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 Lustre.

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 Lustre

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 <NID list>
```

where `<NID list>` 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 Lustre 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 are stopped prior to unloading the modules. Be extremely careful using `rmmod -f`.

To unconfigure the LNET network, run:

```
modprobe -r <any lnd and the lnet modules>
```

Tip – To remove all Lustre modules, run:

```
$ lctl modules | awk '{print $2}' | xargs 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. Network interface failover is planned as an upcoming Lustre feature.
- 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:
  - Fewest hops (to minimize routing), and
  - Appears first in the "networks" or "ip2nets" LNET configuration strings

1. Multi-rail configurations are only supported by o2iblnd; other IB LNDs do not support multiple interfaces.

15.4 Load Balancing with InfiniBand

A Lustre file system contains OSSs with two InfiniBand HCAs. Lustre clients have only one InfiniBand HCA using OFED Infiniband "o2ib" drivers. Load balancing between the HCAs on the OSS is accomplished through LNET.

15.4.1 Setting Up `modprobe.conf` for Load Balancing

To configure LNET for load balancing on clients and servers:

1. **Set the modprobe.conf options.**

   Depending on your configuration, set modprobe.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 the MGS/MDT file system and mount the servers (MGS/MDT and OSS).

   ```
   modprobe lnet

   $ mkfs.lustre --fsname lustre --mgs --mdt <block device name>
   $ mkdir -p <mount point>
   $ mount -t lustre <block device> <mount point>
   $ mount -t lustre <block device> <mount point>

   $ mkfs.lustre --fsname lustre --mgs --mdt <block device name>
   $ mkdir -p <mount point>
   $ mount -t lustre <block device> <mount point>
   $ mount -t lustre <block device> <mount point>
   ```

For example:

```
modprobe lnet
```

```
$ mkfs.lustre --fsname lustre --mdt --mgs /dev/sda
$ mkdir -p /mnt/test/mdt
$ mount -t lustre /dev/sda /mnt/test/mdt
$ mount -t lustre mgs@o2ib0:/lustre /mnt/mdt
```

```
$ mkfs.lustre --fsname lustre --ost --mgsnode=mds@o2ib0 /dev/sda
$ mkdir -p /mnt/test/mdt
$ mount -t lustre /dev/sda /mnt/test/ost
$ mount -t lustre mgs@o2ib0:/lustre /mnt/ost
```

3. **Mount the clients.**

```
mount -t lustre <MGS node>:/<fsname> <mount point>
```

This example shows an IB client being mounted.

```
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 OFA stack (OFED) 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 the actual bottleneck.

```
ip2nets="o2ib0(ib0),  o2ib1(ib1)192.168.[0-1].*           #all servers;\
                      o2ib0(ib0) 192.168.[2-253].[0-252/2]#even clients;\
                      o2ib1(ib1) 192.168.[2-253].[1-253/2]#odd clients"
```

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=”  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 Lustre's simplistic NID selection algorithm. 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.

CHAPTER 16

Upgrading Lustre

This chapter describes Lustre interoperability and how to upgrade to Lustre 2.0, and includes the following sections:

16.1 Lustre Interoperability

Lustre 2.0 is built on a new architectural code base, which is different than the one used with Lustre 1.8. These architectural changes require existing Lustre 1.8.x users to follow a slightly different procedure to upgrade to Lustre 2.0 - requiring clients to be unmounted and the file system be shut down. Once the servers are upgraded and restarted, then the clients can be remounted. After the upgrade, Lustre 2.0 servers can interoperate with compatible 1.8 clients and servers. Lustre 2.0 does *not* support 2.0 clients interoperating with 1.8 servers.

Note – Lustre 1.8 clients support a mix of 1.8 and 2.0 OSTs, not all OSSs need to be upgraded at the same time.

Note – Lustre 2.0 is compatible with version 1.8.4 and above. If you are planning a heterogeneous environment (mixed 1.8 and 2.0 servers), make sure that version 1.8.4 is installed on the client and server nodes that are not upgraded to 2.0.

16.2 Upgrading Lustre 1.8.x to 2.0

Upgrading to Lustre 2.0 involves shutting down the file system and upgrading servers, and optionally clients, all at the same time. Lustre 2.0 does not support a rolling upgrade in which the file system operates continuously while individual servers (or their failover partners) and clients are upgraded one at a time.

Note – Although the Lustre 1.8 to 2.0 upgrade path has been tested, for best results we recommend performing a fresh Lustre 2.0 installation, rather than upgrading from 1.8 to 2.0.

16.2.1 Performing a File System Upgrade

This procedure describes a file system upgrade in which Lustre 2.0 packages are installed on multiple 1.8.x servers and, optionally, clients, requiring a file system shut down. You can choose to upgrade the entire Lustre file system to 2.0 or just upgrade the Lustre servers to 2.0 and leave the clients at 1.8.x. Lustre 2.0 servers can interoperate with compatible 1.8 clients and servers.

Tip – In a Lustre upgrade, the package install and file system unmount steps are reversible; you can do either step first. To minimize downtime, this procedure first performs the 2.0 package installation, and then unmounts the file system.

---

1. **Make a complete, restorable file system backup before upgrading Lustre.**

2. **If any Lustre nodes will not be upgraded to 2.0, make sure that these client and server nodes are at version 1.8.4.**

   Lustre 2.0 is compatible with version 1.8.4 and above. If you are planning a heterogeneous environment (mixed 1.8 and 2.0 clients and servers), make sure that version 1.8.4 is installed on nodes that are not upgraded to 2.0.

3. **Install the 2.0 packages on the Lustre servers and, optionally, the clients.**

   Some or all servers can be upgraded. Some or all clients can be upgraded.

   For help determining where to install a specific package, see TABLE 8-1 (Lustre packages, descriptions and installation guidance).

   a. **Install the kernel, modules and ldiskfs packages. For example:**

   ```
   $ rpm -ivh
   kernel-lustre-smp-<ver> \
   kernel-ib-<ver> \
   lustre-modules-<ver> \
   lustre-ldiskfs-<ver>
   ```

   b. **Upgrade the utilities/userspace packages. For example:**

   ```
   $ rpm -Uvh lustre-<ver>
   ```

   c. **If a new e2fsprogs package is available, upgrade it. For example:**

   ```
   $ rpm -Uvh e2fsprogs-<ver>
   ```

   Use e2fsprogs-1.41-10 or later, available at:

   http://downloads.lustre.org/public/tools/e2fsprogs/

   d. **(Optional) If you want to add optional packages to your Lustre system, install them now.**

4. **Shut down the file system.**

   Shut down the components in this order: clients, then the MDT, then OSTs. Unmounting a block device causes Lustre to be shut down on that node.

   a. **Unmount the clients. On each client node, run:**

      ```
      umount <mount point>
      ```

   b. **Unmount the MDT. On the MDS node, run:**

      ```
      umount <mount point>
      ```

   c. **Unmount the OSTs (be sure to unmount all OSTs). On each OSS node, run:**

      ```
      umount <mount point>
      ```

5. **Unload the old Lustre modules by rebooting the node or manually removing the Lustre modules.**

   Run `lustre_rmmod` several times and use `lsmod` to check the currently loaded modules.

6. **Start the upgraded file system.**

   Start the components in this order: OSTs, then the MDT, then clients.

   a. **Mount the OSTs (be sure to mount all OSTs). On each OSS node, run:**

      ```
      mount -t lustre <block device name> <mount point>
      ```

   b. **Mount the MDT. On the MDS node, run:**

      ```
      mount -t lustre <block device name> <mount point>
      ```

   c. **Mount the file system on the clients. On each client node, run:**

      ```
      mount -t lustre <MGS node>:/<fsname> <mount point>
      ```

If you have a problem upgrading Lustre, contact us via the Bugzilla bug tracker.

CHAPTER 17

Backing Up and Restoring a File System

Lustre provides backups at the file system-level, device-level and file-level. This chapter describes how to backup and restore on Lustre, and includes the following sections:

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 Lustre to scale the filesystem namespace for future applications, Lustre 2.x internally uses a 128-bit file identifier for all files. To interface with user applications, Lustre 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 filesystems (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 filesystem 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 recommended 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 36.13, "lustre_rsync" on page 36-23. 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 changelog_register parameter in the Section 36.3, "lctl" on page 36-4.

- 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=<user id>`	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 Section 36.3, "lctl" on page 36-4. 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 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 overriden. 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. lustre-MDT0000).

```
# lctl --device lustre-MDT0000 changelog_register lustre-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=lustre-MDT0000 --user=cl1 \
             --statuslog sync.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 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: lustre
MDT device: lustre-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=lustre-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.

Note – In Lustre 2.0 and 2.1 the only correct way to perform an MDT backup and restore is to do a device-level backup as is described in this section. The ability to do MDT file-level backups is not functional in these releases because of the inability to restore the Object Index (OI) file correctly (see bug 22741 for details).

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/{new} 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/{new} 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 -f` on the new device.

17.3 Making a File-Level Backup of an OST File System

This procedure provides another way to backup or migrate the data of an OST at the file level, so that the unused space of the OST does not need to be backed up. 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. 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 – In Lustre 2.0 and 2.1 the only correct way to perform an MDT backup and restore is to do a device-level backup as is described in this section. The ability to do MDT file-level backups is not functional in these releases because of the inability to restore the Object Index (OI) file correctly (see bug 22741 for details).

---

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 it 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 hae 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 --sparse .
```

Note – In Lustre 1.6.7 and later, the `--sparse` option reduces the size of the backup file. Be sure to use it so the `tar` command does not mistakenly create an archive full of zeros.

---

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 Section 14.5, “Changing a Server NID” on page 14-5.

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} \
     {newdev}
   ```

2. **Mount the file system.**

   ```
   [oss]# mount -t ldiskfs {newdev} /mnt/ost
   ```

3. **Change to the new file system mount point.**

   ```
   [oss]# cd /mnt/ost
   ```

4. **Restore the file system backup.**

   ```
   [oss]# tar xzvpf {backup file} --sparse
   ```

5. **Restore the file system extended attributes.**

   ```
   [oss]# setfattr --restore=ea-${date}.bak
   ```

6. **Verify that the extended attributes were restored.**

   ```
   [oss]# getfattr -d -m ".*" -e hex O/0/d0/100992 trusted.fid= \
   0x0d822200000000004a8a73e500000000808a0100000000000000000000000000
   ```

7. **Change directory out of the file system.**

   ```
   [oss]# cd -
   ```

8. **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 lfsck tool (part of Lustre e2fsprogs) can optionally 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.

17.5 Using LVM Snapshots with Lustre

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 volgroup /dev/sda1
    Volume group "volgroup" successfully created
cfs21:~# lvcreate -L200M -nMDT volgroup
    Logical volume "MDT" created
cfs21:~# lvcreate -L200M -nOST0 volgroup
    Logical volume "OST0" created
cfs21:~# lvscan
    ACTIVE           '/dev/volgroup/MDT' [200.00 MB] inherit
    ACTIVE           '/dev/volgroup/OST0' [200.00 MB] 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 --mdt --fsname=main /dev/volgroup/MDT
 No management node specified, adding MGS to this MDT.
    Permanent disk data:
 Target:     main-MDTffff
 Index:      unassigned
 Lustre FS:  main
 Mount type: ldiskfs
 Flags:      0x75
               (MDT MGS needs_index first_time update )
 Persistent mount opts: errors=remount-ro,iopen_nopriv,user_xattr
 Parameters:
checking for existing Lustre data
 device size = 200MB
 formatting backing filesystem ldiskfs on /dev/volgroup/MDT
         target name  main-MDTffff
         4k blocks     0
         options        -i 4096 -I 512 -q -O dir_index -F
 mkfs_cmd = mkfs.ext2 -j -b 4096 -L main-MDTffff  -i 4096 -I 512 -q
-O dir_index -F /dev/volgroup/MDT
 Writing CONFIGS/mountdata
cfs21:~# mkfs.lustre --ost --mgsnode=cfs21 --fsname=main
/dev/volgroup/OST0
    Permanent disk data:
 Target:     main-OSTffff
Index:      unassigned
 Lustre FS:  main
 Mount type: ldiskfs
 Flags:      0x72
               (OST needs_index 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 = 200MB
 formatting backing filesystem ldiskfs on /dev/volgroup/OST0
         target name  main-OSTffff
         4k blocks     0
         options        -I 256 -q -O dir_index -F
```

```
 mkfs_cmd = mkfs.ext2 -j -b 4096 -L main-OSTffff  -I 256 -q -O
dir_index -F /dev/ volgroup/OST0
 Writing CONFIGS/mountdata
cfs21:~# mount -t lustre /dev/volgroup/MDT /mnt/mdt
cfs21:~# mount -t lustre /dev/volgroup/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 MDTb1 /dev/volgroup/MDT
   Rounding up size to full physical extent 52.00 MB
   Logical volume "MDTb1" created
cfs21:~# lvcreate -L50M -s -n OSTb1 /dev/volgroup/OST0
   Rounding up size to full physical extent 52.00 MB
   Logical volume "OSTb1" 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/volgroup/MDTb1
 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/volgroup/OSTb1
 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 an FS, we must also erase the last_rcvd file from the
snapshots
cfs21:~# mount -t ldiskfs /dev/volgroup/MDTb1 /mnt/mdtback
 cfs21:~# rm /mnt/mdtback/last_rcvd
 cfs21:~# umount /mnt/mdtback
 cfs21:~# mount -t ldiskfs /dev/volgroup/OSTb1 /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-tlustre/dev/volgroup/MDTb1/mnt/mdtback
 cfs21:~# mount -t lustre /dev/volgroup/OSTb1 /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/volgroup/MDTb1
```

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/volgroup/MDTb1
```

Note – Extending snapshots seems to be broken in older LVM. It is working in LVM v2.02.01.

CHAPTER 18

Managing File Striping and Free Space

This chapter describes file striping and I/O options, and includes the following sections:

18.1 How Lustre Striping Works

Lustre uses a round-robin algorithm for selecting the next OST to which a stripe is to be written. 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.

The MDS allocates objects on seqential OSTs. Periodically, it will adjust the striping layout to eliminate some degenerated cases where applications that create very regular file layouts (striping patterns) would preferentially use a particular OST in the sequence.

Stripes are written to sequential OSTs until free space across the OSTs differs by more than 20%. The MDS will then use 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 to within 20% again.

For a more detailed description of stripe assignments, see Section 18.5, "Managing Free Space" on page 18-9.

18.2 Lustre File Striping Considerations

Whether you should set up file striping and what parameter values you select depends on your need. 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 – more bandwidth than can be provided by a single OSS. For example, scientific applications that write 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 160 OSSs, 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 also 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. In this case, there is needless contention.
- **Increased risk** - When a file is 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, you lose fewer files, but you lose them 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 small balancing act, but there are reasonable defaults.

- **The stripe size must be a multiple of the page size.** Lustre's 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 Lustre 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 on shared file access.

- **The maximum stripe size is 4GB.** 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, it can be counterproductive in some cases if it does not match your I/O pattern.
- **Choose a stripe pattern that takes into account your application's write patterns.** 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 very consistent and aligned way, make the stripe size a multiple of the write() size.
- **The choice of stripe size has no effect on a single-stripe file.**

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, and 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. Otherwise, the file starts on the specified OST index. The numbering of the OSTs starts at 0.

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:

```
lfs setstripe -c <stripe_count> <file>
```

pool_name

Specify the OST pool on 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 Section 19.2, "Creating and Managing OST Pools" on page 19-6.

18.3.1 Using a Specific Striping Pattern/File Layout 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
```

As can be seen, 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" on page 18-5 which shows only a single object for the file.

18.3.2 Changing Striping 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), Lustre uses those striping parameters instead of the file system default for the new file.

To change the striping pattern (file layout) 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 Changing Striping for a File System

Change the striping on the file system `root` will change the striping for all newly created files that would otherwise have a striping parameter from the parent directory or explicitly on the command line.

Note – Striping of new files and sub-directories is done per the striping parameter settings of the root directory. Once you set striping on the `root` directory, then, by default, it applies to any new child directories created in that `root` directory (unless they have their own striping settings).

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 "bob" will be created on the first OST (id 0).

```
$ lfs setstripe --count 1 --index 0 bob
$ dd if=/dev/zero of=bob count=1 bs=100M
1+0 records in
1+0 records out
$ lfs getstripe bob
```

OBDS:

```
0: home-OST0000_UUID ACTIVE
[...]
bob
          obdidx       objid        objid            group
          0            33459243     0x1fe8c2b        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 printed.

18.4.1 Displaying the Current Stripe Size

To see the current stripe size, use the `lfs getstripe` command on a Lustre file or directory. For example:

```
[client]# lfs getstripe /mnt/lustre
```

This command produces output similar to this:

```
/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 OSTs), the default stripe size is 1 MB, and objects are created over all available OSTs.

18.4.2 Inspecting the File Tree

To inspect an entire tree of files, use the `lfs find` command:

```
lfs find [--recursive | -r] <file or directory> ...
```

You can also use `ls -l /proc/<pid>/fd/` to find open files using Lustre. For example:

```
$ lfs getstripe $(readlink /proc/$(pidof cat)/fd/1)
```

Typical output is:

```
/mnt/lustre/foo
obdidx           objid        objid        group
2                835487       0xcbf9f      0
```

In this example, the file lives on `obdidx 2`, which is `lustre-OST0002`. To see which node is serving that OST, run:

```
$ lctl get_param osc.lustre-OST0002-osc.ost_conn_uuid
```

Typical output is:

```
osc.lustre-OST0002-osc.ost_conn_uuid=192.168.20.1@tcp
```

18.5 Managing Free Space

The MDT assigns file stripes to OSTs based on location (which OSS) and size considerations (free space) to optimize file system performance. Emptier OSTs are preferentially selected for stripes, and stripes are preferentially spread out between OSSs to increase network bandwidth utilization. The weighting factor between these two optimizations can be adjusted by the user.

18.5.1 Checking File System Free Space

Free space is an important consideration in assigning file stripes. The `lfs df` command shows 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.

Option	Description
-h	Human-readable print 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. Depending on the configuration, it may be possible to create more inodes than initially reported by `df -i`. Later, `df -i` operations will show the current, estimated free inode count.

If the underlying file system has fewer free blocks than inodes, then the total inode count for the file system reports only as many inodes as there are free blocks. This is done because Lustre may need to store an external attribute for each new inode, and it is better to report a free inode count that is the guaranteed, minimum number of inodes that can be created.

Examples

```
[lin-cli1] $ 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

[lin-cli1] $ 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

[lin-cli1] $ 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 Using Stripe Allocations

Two stripe allocation methods are provided: *round-robin* and *weighted*. By default, the allocation method is determined by the amount of free-space imbalance on the OSTs. The weighted allocator is used when any two OSTs are imbalanced by more than 20%. Otherwise, the faster round-robin allocator is used. (The round-robin order maximizes network balancing.)

- **Round-robin allocator** - When OSTs have approximately the same amount of free space (within 20%), an efficient round-robin allocator is used. 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. Here are several 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 is significant (by default, 20% of the free space), then a weighting algorithm is used to influence OST ordering based on size and location. Note that these are weightings for a random algorithm, so the OST with the most free space is not necessarily chosen each time. On average, the weighted allocator fills the emptier OSTs faster.

18.5.3 Adjusting the Weighting Between Free Space and Location

The weighting priority can be adjusted in the `/proc` file `/proc/fs/lustre/lov/lustre-mdtlov/qos_prio_free proc`. The default value is 90%. Use this command on the MGS to permanently change this weighting:

```
lctl conf_param <fsname>-MDT0000.lov.qos_prio_free=90
```

Increasing this value puts more weighting on free space. When the free space priority is set to 100%, then location is no longer used in stripe-ordering calculations and weighting is based entirely on free space.

Note – Setting the priority to 100% means that OSS distribution does not count in the weighting, but the stripe assignment is still done via a weighting. For example, if OST2 has twice as much free space as OST1, then OST2 is twice as likely to be used, but it is not guaranteed to be used.

CHAPTER 19

Managing the File System and I/O

This chapter describes file striping and I/O options, and includes the following sections:

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:

```
root@LustreClient01 ~]# 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:

```
[root@LustreClient01 ~]# lfs setstripe /mnt/lustre 4M 0 -1
[root@LustreClient01 ~]# 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:**

```
[root@LustreClient01 ~]# 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:**

```
[root@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:**

```
[root@mds ~]# lctl --device 7 deactivate
```

4. **Display the status of the file system components:**

```
[root@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 32.2, "lfs_migrate" on page 32-13). 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     objidobjidgroup
        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_UUID} -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 19.1.1, "Checking OST Space Usage" on page 19-2.

```
[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: 11.8G7.3G3.9G 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:

```
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:

```
lctl pool_add <fsname>.<poolname> <ost_list>
```

Where:

- `<ost_list>` is `<fsname->OST<index_range>[_UUID]`
- `<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 (`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:

```
lctl pool_remove <fsname>.<poolname> <ost_list>
```

To destroy a pool, run:

```
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:

```
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:

```
lfs setstripe --pool|-p pool_name <filename|dirname>
```

To set striping patterns, run:

```
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:**

```
$ mkfs.lustre --fsname=spfs --ost --mgsnode=mds16@tcp0 /dev/sda
$ mkdir -p /mnt/test/ost0
$ mount -t lustre /dev/sda /mnt/test/ost0
```

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 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

Lustre 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.

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 [102400-106495]
```

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:

```
echo 1 > /proc/fs/lustre/llite/<fsname>/checksum_pages
```

To disable both types of checksums (in-memory and wire), run:

```
echo 0 > /proc/fs/lustre/llite/<fsname>/checksum_pages
```

To check the status of a wire checksum, run:

```
lctl get_param osc.*.checksums
```

19.5.1.1 Changing Checksum Algorithms

By default, Lustre uses the adler32 checksum algorithm, because it is robust and has a lower impact on performance than crc32. The Lustre administrator can change the checksum algorithm via /proc, depending on what is supported in the kernel.

To check which checksum algorithm is being used by Lustre, run:

```
$ cat /proc/fs/lustre/osc/<fsname>-OST<index>-osc-*/checksum_type
```

To change the wire checksum algorithm used by Lustre, run:

```
$ echo <algorithm name> /proc/fs/lustre/osc/<fsname>-OST<index>- \
osc-*/checksum_type
```

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 `cat` command is used to determine that Lustre is using the adler32 checksum algorithm. Then the `echo` command is used to change the checksum algorithm to crc32. A second `cat` command confirms that the crc32 checksum algorithm is now in use.

```
$ cat /proc/fs/lustre/osc/lustre-OST0000-osc- \
ffff81012b2c48e0/checksum_type

crc32 [adler]

$ echo crc32 > /proc/fs/lustre/osc/lustre-OST0000-osc- \
ffff81012b2c48e0/checksum_type

$ cat /proc/fs/lustre/osc/lustre-OST0000-osc- \
ffff81012b2c48e0/checksum_type

[crc32] adler
```

CHAPTER 20

Managing Failover

This chapter describes failover in a Lustre system and includes the following sections:

- Lustre Failover and Multiple-Mount Protection

Note – For information about high availability(HA) management software, see the Lustre wiki topic Using Red Hat Cluster Manager with Lustre or the Lustre wiki topic Using Pacemaker with Lustre.

20.1 Lustre Failover and Multiple-Mount Protection

The failover functionality in Lustre is implemented with the multiple-mount protection (MMP) feature, which protects the 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 OSTs share a partition).

Lustre's 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 >= 2.6.9.

20.1.1 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 administrator can manually enable MMP when the file system is unmounted.

Use the following commands to determine whether MMP is running in Lustre and to enable or disable the MMP feature.

To determine if MMP is enabled, run:

```
dumpe2fs -h <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 <device>
```

To manually enable MMP, run:

```
tune2fs -O mmp <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:

21.1 Working with Quotas

Quotas allow a system administrator to limit the amount of disk space a user or group can use in a directory. 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` command (post-mount).
- Quotas are distributed (as Lustre 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. When quota is enabled, it is enabled for all clients of the file system; started automatically using `quota_type` or started manually with `lfs quotaon`.

Caution – Although quotas are available in Lustre, 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

Use this procedure to enable (configure) disk quotas in Lustre.

1. **If you have re-complied your Linux kernel, be sure that CONFIG_QUOTA and CONFIG_QUOTACTL are enabled. Also, verify that CONFIG_QFMT_V1 and/or CONFIG_QFMT_V2 are enabled.**

   Quota is enabled in all Linux 2.6 kernels supplied for Lustre.

2. **Start the server.**

3. **Mount the Lustre file system on the client and verify that the lquota module has loaded properly by using the `lsmod` command.**

```
$ lsmod
[root@oss161 ~]# lsmod
Module            Size      Used by
obdfilter         220532    1
fsfilt_ldiskfs    52228     1
ost               96712     1
mgc               60384     1
ldiskfs           186896    2 fsfilt_ldiskfs
lustre            401744    0
lov               289064    1 lustre
lquota            107048    4 obdfilter
mdc               95016     1 lustre
ksocklnd          111812    1
```

The Lustre `mount` command no longer recognizes the `usrquota` and `grpquota` options. If they were previously specified, remove them from `/etc/fstab`.

When quota is enabled, it is enabled for all file system clients (started automatically using `quota_type` or manually with `lfs quotaon`).

Note – Lustre with the Linux kernel 2.4 does *not* support quotas.

To enable quotas automatically when the file system is started, you must set the `mdt.quota_type` and `ost.quota_type` parameters, respectively, on the MDT and OSTs. The parameters can be set to the string `u` (user), `g` (group) or `ug` for both users and groups.

You can enable quotas at mkfs time (`mkfs.lustre --param mdt.quota_type=ug`) or with tunefs.lustre. As an example:

```
tunefs.lustre --param ost.quota_type=ug $ost_dev
```

Caution – If you are using `mkfs.lustre --param mdt.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.

21.2.0.1 Administrative and Operational Quotas

Lustre has two kinds of quota files:

- Administrative quotas (for the MDT), which contain limits for users/groups for the entire cluster.
- Operational quotas (for the MDT and OSTs), which contain quota information dedicated to a cluster node.

Lustre 1.6.5 introduced the v2 file format for administrative quota files, with continued support for the old file format (v1). The `mdt.quota_type` parameter also handles '1' and '2' options, to specify the Lustre quota versions that will be used. For example:

```
--param mdt.quota_type=ug1
--param mdt.quota_type=u2
```

Lustre 1.6.6 introduced the v2 file format for operational quotas, with continued support for the old file format (v1). The `ost.quota_type` parameter handles '1' and '2' options, to specify the Lustre quota versions that will be used. For example:

```
--param ost.quota_type=ug2
--param ost.quota_type=u1
```

For more information about the v1 and v2 formats, see Section 21.5.3, "Quota File Formats" on page 21-12.

21.3 Creating Quota Files and Quota Administration

Once each quota-enabled file system is remounted, it is capable of working with disk quotas. However, the file system is not yet ready to support quotas. If umount has been done regularly, run the `lfs` command with the `quotaon` option. If umount has not been done, perform these steps:

1. **Take Lustre "offline".**

   That is, verify that no write operations (append, write, truncate, create or delete) are being performed (preparing to run `lfs quotacheck`). Operations that do not change Lustre files (such as read or mount) are okay to run.

Caution – When `lfs quotacheck` is run, Lustre must NOT be performing any write operations. Failure to follow this caution may cause the statistic information of quota to be inaccurate. For example, the number of blocks used by OSTs for users or groups will be inaccurate, which can cause unexpected quota problems.

---

2. **Run the `lfs` command with the `quotacheck` option:**

   ```
   # lfs quotacheck -ug /mnt/lustre
   ```

   By default, quota is turned on after `quotacheck` completes. Available options are:

   - `u` — checks the user disk quota information
   - `g` — checks the group disk quota information

   The `lfs quotacheck` command checks all objects on all OSTs and the MDS to sum up for every UID/GID. It reads all Lustre metadata and re-computes the number of blocks/inodes that each UID/GID has used. If there are many files in Lustre, it may take a long time to complete.

Note – User and group quotas are separate. If either quota limit is reached, a process with the corresponding UID/GID cannot allocate more space on the file system.

Note – When `lfs quotacheck` runs, it creates a quota file -- a sparse file with a size proportional to the highest UID in use and UID/GID distribution. As a general rule, if the highest UID in use is large, then the sparse file will be large, which may affect functions such as creating a snapshot.

Note – For Lustre 1.6 releases before version 1.6.5, and 1.4 releases before version 1.4.12, if the underlying ldiskfs file system has not unmounted gracefully (due to a crash, for example), re-run `quotacheck` to obtain accurate quota information. Lustre 1.6.5 and 1.4.12 use journaled quota, so it is not necessary to run `quotacheck` after an unclean shutdown.

In certain failure situations (e.g., when a broken Lustre installation or build is used), re-run `quotacheck` after checking the server kernel logs and fixing the root problem.

The `lfs` command includes several command options to work with quotas:

- `quotaon` — enables disk quotas on the specified file system. The file system quota files must be present in the root directory of the file system.
- `quotaoff` — disables disk quotas on the specified file system.
- `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 quotaon [-ugf] <filesystem>

lfs quotaoff [-ug] <filesystem>

lfs quota [-q] [-v] [-o obd_uuid] [-u|-g <uname>|uid|gname|gid>]
<filesystem>

lfs quota -t <-u|-g> <filesystem>

lfs setquota <-u|--user|-g|--group> <username|groupname>
[-b <block-softlimit>] [-B <block-hardlimit>] [-i <inode-softlimit>]
[-I <inode-hardlimit>] <filesystem>
```

Examples:

In all of the examples below, the file system is /mnt lustre.

To turn on user and group quotas, run:

```
$ lfs quotaon -ug /mnt/lustre
```

To turn off user and group quotas, run:

```
$ lfs quotaoff -ug /mnt/lustre
```

To display general quota information (disk usage and limits) for the user running the command and his primary group, run:

```
$ lfs quota /mnt/lustre
```

To display general quota information for a specific user ("bob" in this example), run:

```
$ lfs quota -u bob /mnt/lustre
```

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/lustre
```

To display general quota information for a specific group ("eng" in this example), run:

```
$ lfs quota -g eng /mnt/lustre
```

To display block and inode grace times for user quotas, run:

```
$ lfs quota -t -u /mnt/lustre
```

To set user and group quotas for a specific user ("bob" in this example), run:

```
$ lfs setquota -u bob 307200 309200 10000 11000 /mnt/lustre
```

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.

Note – For the Lustre command `$ lfs setquota/quota` ... the qunit for block is KB (1024) and the qunit for inode is 1.

The `quota` command displays the quota allocated and consumed for each Lustre device. Using the previous `setquota` example, running this lfs `quota` command:

```
$ lfs quota -u bob -v /mnt/lustre
```

displays this command output:

```
Disk quotas for user bob (uid 6000):
Filesystem           kbytes  quotalimitgracefilesquotalimitgrace
/mnt/lustre          0       3072030920-  0    1000011000-
lustre-MDT0000_UUID  0       -   16384-   0    -   2560-
lustre-OST0000_UUID  0       -   16384-   0    -   0-
lustre-OST0001_UUID  0       -   16384-   0    -   0-
```

21.4 Quota Allocation

In Lustre, 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. By default, Lustre supports both user and group quotas to limit disk usage and file counts.

The quota system in Lustre is completely compatible with the quota systems used on other file systems. The Lustre quota system distributes quotas from the quota master. Generally, the MDS is the quota master for both inodes and blocks. All OSTs and the MDS are quota slaves to the OSS nodes. To reduce quota requests and get reasonably accurate quota distribution, the transfer quota unit (qunit) between quota master and quota slaves is changed dynamically by the lquota module. The default minimum value of qunit is 1 MB for blocks and 2 for inodes. The proc entries to set these values are: `/proc/fs/lustre/mds/lustre-MDT*/quota_least_bunit` and `/proc/fs/lustre/mds/lustre-MDT*/quota_least_iunit`. The default maximum value of qunit is 128 MB for blocks and 5120 for inodes. The proc entries to set these values are `quota_bunit_sz` and `quota_iunit_sz` in the MDT and OSTs.

Note – In general, the `quota_bunit_sz` value should be larger than 1 MB. For testing purposes, it can be set to 4 KB, if necessary.

The file system block quota is divided up among the OSTs and the MDS within the file system. Only the MDS uses the file system inode quota.

This means that the minimum quota for block is 1 MB* (the number of OSTs + the number of MDSs), which is 1 MB* (number of OSTs + 1). If you attempt to assign a smaller quota, users maybe not be able to create files. As noted, the default minimum quota for inodes is 2. The default is established at file system creation time, but can be tuned via /proc values (described below). The inode quota is also allocated in a quantized manner on the MDS.

If we look at the setquota example again, running this `lfs quota` command:

```
# lfs quota -u bob -v /mnt/lustre
```

displays this command output:

```
Disk quotas for user bob (uid 500):
Filesystem           kbytes  quotalimitgracefilesquotalimitgrace
/mnt/lustre          30720*  30720309206d23h56m44s10101*10000110006d23h59m50s
lustre-MDT0000_UUID  0       -   1024-    10101-   10240
lustre-OST0000_UUID  0       -   1024-    -    -   -
lustre-OST0001_UUID  30720*  -   28872-   -    -   -
```

The total quota limit of 30,920 is allotted to user `bob`, which is further distributed to two OSTs and one MDS.

Note – Values appended with "*" show the limit that has been over-used (exceeding the quota), and receives this message `Disk quota exceeded`. For example:

```
\
$ cp: writing `/mnt/lustre/var/cache/fontconfig/
beeeeb3dfe132a8a0633a017c99ce0-x86.cache': Disk quota exceeded.
```

The requested quota of 300 MB is divided across the OSTs.

Note – It is very important to note that the block quota is consumed per OST and the MDS per block and inode (there is only one MDS for inodes). Therefore, when the quota is consumed on one OST, the client may not be able to create files regardless of the quota available on other OSTs.

Additional information:

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 periods are applied to all users. The user block soft limit is for all users who are using a blocks quota.

Soft limit — Once you are beyond the soft limit, the quota module begins to time, but you still can write block and inode. When you are always beyond the soft limit and use up your grace time, you get the same result as the hard limit. For inodes and blocks, it is the same. Usually, the soft limit MUST be less than the hard limit; if not, the quota module never triggers the timing. If the soft limit is not needed, leave it as zero (0).

Hard limit — When you are beyond the hard limit, you get -EQUOTA and cannot write inode/block any more. The hard limit is the absolute limit. When a grace period is set, you can exceed the soft limit within the grace period if are under the hard limits.

Lustre quota allocation is controlled by two variables, `quota_bunit_sz` and `quota_iunit_sz` referring to KBs and inodes, respectively. These values can be accessed on the MDS as `/proc/fs/lustre/mds/*/quota_*` and on the OST as `/proc/fs/lustre/obdfilter/*/quota_*`. The `quota_bunit_sz` and `quota_iunit_sz` variables are the maximum qunit values for blocks and inodes, respectively. At any time, module lquota chooses a reasonable qunit between the minimum and maximum values.

The `/proc` values are bounded by two other variables `quota_btune_sz` and `quota_itune_sz`. By default, the `*tune_sz` variables are set at 1/2 the `*unit_sz` variables, and you cannot set `*tune_sz` larger than `*unit_sz`. You must set `bunit_sz` first if it is increasing by more than 2x, and `btune_sz` first if it is decreasing by more than 2x.

Total number of inodes — To determine the total number of inodes, use `lfs df -i` (and also `/proc/fs/lustre/*/*/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" on page 18-9.

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 a file system's total inode count. Instead, you should know (accurately), the free inode count and the used inode count for a file system. Lustre manipulates the total inode count in order to accurately report the other two values.

The values set for the MDS must match the values set on the OSTs.

The `quota_bunit_sz` parameter displays bytes, however `lfs setquota` uses KBs. The `quota_bunit_sz` parameter must be a multiple of 1024. A proper minimum KB size for `lfs setquota` can be calculated as:

Size in KBs = minimum_quota_bunit_sz * (number of OSTS + 1) = 1024 * (number of OSTs +1)

We add one (1) to the number of OSTs as the MDS also consumes KBs. As inodes are only consumed on the MDS, the minimum inode size for `lfs setquota` is equal to `quota_iunit_sz`.

Note – Setting the quota below this limit may prevent the user from all file creation.

21.5 Known Issues with Quotas

Using quotas in Lustre can be complex and there are several known issues.

21.5.1 Granted Cache and Quota Limits

In Lustre, granted cache does not respect quota limits. In this situation, OSTs grant cache to 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 Lustre.**
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 `max_dirty_buffer` in the clients (can be set from 0 to 512). To set `max_dirty_buffer` to 0:

* In releases after Lustre 1.6.5, `lctl set_param osc.*.max_dirty_mb=0`.

* In releases before Lustre 1.6.5, `proc/fs/lustre/osc/*/max_dirty_mb; do echo 512 > $O`

21.5.2 Quota Limits

Available quota limits depend on the Lustre version you are using.

- Lustre version 1.4.11 and earlier (for 1.4.x releases) and Lustre version 1.6.4 and earlier (for 1.6.x releases) support quota limits less than 4 TB.
- Lustre versions 1.4.12, 1.6.5 and later support quota limits of 4 TB and greater in Lustre configurations with OST storage limits of 4 TB and less.
- Future Lustre versions are expected to `support` quota limits of 4 TB and greater with no OST storage limits.

Lustre Version	Quota Limit Per User/Per Group	OST Storage Limit
1.4.11 and earlier	< 4TB	n/a
1.4.12	=> 4TB	<= 4TB of storage
1.6.4 and earlier	< 4TB	n/a
1.6.5	=> 4TB	<= 4TB of storage
Future Lustre versions	=> 4TB	No storage limit

21.5.3 Quota File Formats

Lustre 1.6.5 introduced the v2 file format for administrative quotas, with 64-bit limits that support large-limits handling. The old quota file format (v1), with 32-bit limits, is also supported. Lustre 1.6.6 introduced the v2 file format for operational quotas. A few notes regarding the current quota file formats:

Lustre 1.6.5 and later use `mdt.quota_type` to force a specific administrative quota version (v2 or v1).

- For the v2 quota file format, (OBJECTS/admin_quotafile_v2.{usr,grp})
- For the v1 quota file format, (OBJECTS/admin_quotafile.{usr,grp})

Lustre 1.6.6 and later use `ost.quota_type` to force a specific operational quota version (v2 or v1).

- For the v2 quota file format, (lquota_v2.{user,group})
- For the v1 quota file format, (lquota.{user,group})

The `quota_type` specifier can be used to set different combinations of administrative/operational quota file versions on a Lustre node:

- "1" - v1 (32-bit) administrative quota file, v1 (32-bit) operational quota file (default in releases before Lustre 1.6.5)
- "2" - v2 (64-bit) administrative quota file, v1 (32-bit) operational quota file (default in Lustre 1.6.5)
- "3" - v2 (64-bit) administrative quota file, v2 (64-bit) operational quota file (default in releases after Lustre 1.6.5)

If quotas do not exist or look broken, then `quotacheck` creates quota files of a required name and format.

If Lustre is using the v2 quota file format when only v1 quota files exist, then `quotacheck` converts old v1 quota files to new v2 quota files. This conversion is triggered automatically, and is transparent to users. If an old quota file does not exist or looks broken, then the new v2 quota file will be empty. In case of an error, details can be found in the kernel log of the corresponding MDS/OST. During conversion of a v1 quota file to a v2 quota file, the v2 quota file is marked as broken, to avoid it being used if a crash occurs. The quota module does not use broken quota files (keeping quota off).

In most situations, Lustre administrators do not need to set specific versioning options. Upgrading Lustre without using `quota_type` to force specific quota file versions results in quota files being upgraded automatically to the latest version. The option ensures backward compatibility, preventing a quota file upgrade to a version which is not supported by earlier Lustre versions.

21.6 Lustre Quota Statistics

Lustre 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.
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.

Quota Event	Description
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.

21.6.1 Interpreting Quota Statistics

Quota statistics are an important measure of a Lustre file system's performance. 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:

```
cat /proc/fs/lustre/lquota/lustre-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).

CHAPTER 22

Managing Lustre Security

This chapter describes Lustre security and includes the following sections:

22.1 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.

22.1.1 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 Linux, 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. Lustre 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.

22.1.2 Using ACLs with Lustre

POSIX Access Control Lists (ACLs) can be used with Lustre. 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 Lustre on a system-wide basis; either all clients enable ACLs or none do. Activating ACLs is controlled by MDS mount options `acl` / `noacl` (enable/disableACLs). 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.

22.1.3 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-heade rain
user::rwx
user:chirag:rwx
group::r-x
mask::rwx
other::---
```

22.2 Using Root Squash

Lustre 1.6 introduced root squash functionality, a security feature which controls super user access rights to an Lustre file system. Before the root squash feature was added, Lustre users could run `rm -rf *` as root, and remove data which should not be deleted. Using the root squash feature prevents this outcome.

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 administrator to specify a set of client for which UID/GID re-mapping does not apply.

22.2.1 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 22.2.2, "Enabling and Tuning Root Squash" on page 22-6). 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.

22.2.2 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:

```
mkfs.lustre --reformat --fsname=Lustre --mdt --mgs \
            --param "mds.root_squash=500:501" \
            --param "mds.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 "mds.root_squash=65534:65534"  \
--param "mds.nosquash_nids=192.168.0.13@tcp0" /dev/sda1
```

Root squash parameters can also be changed with the lctl conf_param command. For example:

```
lctl conf_param Lustre.mds.root_squash="1000:100"
lctl conf_param Lustre.mds.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:

```
lctl conf_param Lustre.mds.nosquash_nids="NONE"
```

- OR -

```
lctl conf_param Lustre.mds.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:

```
mkfs.lustre ... --param "mds.nosquash_nids='0@elan1 1@elan2'" /dev/sda1
lctl conf_param Lustre.mds.nosquash_nids="24@elan 15@elan1"
```

These are examples of incorrect syntax:

```
mkfs.lustre ... --param "mds.nosquash_nids=0@elan1 1@elan2" /dev/sda1
lctl conf_param Lustre.mds.nosquash_nids=24@elan 15@elan1
```

To check root squash parameters, use the `lctl get_param` command:

```
lctl get_param mds.Lustre-MDT0000.root_squash
lctl get_param mds.Lustre-MDT000*.nosquash_nids
```

Note – An empty `nosquash_nids` list is reported as NONE.

22.2.3 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 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 Lustre 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 Performance (Lustre I/O Kit)

Lustre Tuning

CHAPTER 23

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:

23.1 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 25: Lustre Tuning 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 Lustre.

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 user can invite a kernel test node to join the test session by running `lst add_group NID`, but the console 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 is running `lstclient` to connect to the console.

23.1.1 Prerequisites

To run LNET self-test, these modules must be loaded:

- `libcfs`
- `net`
- `lnet_selftest`
- One of the `klnds` (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 and test nodes require all the prerequisite modules to be loaded, userspace test nodes do not require these modules.

23.2 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.

23.2.1 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 23.3.1, “Session Commands” on page 23-7.

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
```

23.2.2 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 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:

```
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 user 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. However, the console user 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`).

23.2.3 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 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 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 (`brw read`) or data is transferred from the source to the target (`brw write`). 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 23.3.3, "Batch and Test Commands" on page 23-11.

23.2.4 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).

23.3 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.

23.3.1 Session Commands

This section describes `lst session` commands.

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] NAME

Creates a new session.

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
```

23.3.2 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
```

23.3.3 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 <#>*]* [--concurrency <#>] [--distribute <#:#>] --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 <#>`	Number of times to run the test.
`--concurrency <#>`	The number of requests that are active at one time.
`--distribute <#:#>`	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).

Parameter	Description	
ping	Sends a small request message, resulting in a small reply message. For more details, see Section 23.2.3, "Defining and Running the Tests" on page 23-5	
brw	Sends a small request message followed by a bulk data transfer, resulting in a small reply message. Section 23.2.3, "Defining and Running the Tests" on page 23-5. Options are:	
	read | write	Read or write. The default is read.
	size=<#> | <#>K | <#>M	I/O size in bytes, KB or MB (i.e., size=1024, size=4K, size=1M). The default is 4K bytes.
	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->S2), (C5,C6->S3)
--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. IIf 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_batch
bulkperf
$ 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 <#>] [--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.
`--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
```

23.3.4 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 clients
clients
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 24

Benchmarking Lustre 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. It includes:

24.1 Using Lustre I/O Kit Tools

The tools in the Lustre I/O Kit are used to benchmark Lustre 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.

24.1.1 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 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, Lustre should deliver 85-90% of the raw device performance.

A utility `state-collect` is also provided to collect application profiling information from Lustre clients and servers. See Section 24.5, "Collecting Application Profiling Information (stats-collect)" on page 24-14 for more information.

24.1.2 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 23: 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.lustre.org/public/tools/lustre-iokit/

24.2 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.

Prequisites:

- `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 that shipped with RHEL4U4.)

24.2.1 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
```

24.2.2 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 prefixdefined 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 disjount 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.

24.3 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 24.3.1, "Testing Local Disk Performance" on page 24-7.

- **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 24.3.2, "Testing Network Performance" on page 24-9

- **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 24.3.3, "Testing Remote Disk Performance" on page 24-10.

Caution – The `obdfilter_survey` script is destructive and should not be run on devices that containing existing data that needs to be preserved. Thus, tests using `obdfilter_survey` should be run before the Lustre file system is placed in production.

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.

24.3.1 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 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
   ```

24.3.2 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 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="oss1 oss2" case=network \
     sh obdfilter-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.

24.3.3 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 seperated by spaces using the format *<fsname>-<OST_name>*-osc-*<OSC_number>* (for example, lustre-OST0000-osc-ffff88007754bc00). You do not have to specify an MDS or LOV.

5. **Run the o**bdfilter_survey **script with the** target= parameter **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
```

24.3.4 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.

24.3.4.1 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.

24.3.4.2 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/*/brw_stats` (see Section 31.2.5, “Watching the OST Block I/O Stream” on page 31-17 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.

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`.

24.4 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 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
```

24.5 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 to be installed and set up on your cluster
- SSH and SCP access to these nodes without requiring a password

24.5.1 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 block read/write statistics (`brw_stats`)
- `SDIO` - SCSI disk IO statistics (`sd_iostats`)
- `MBALLOC` - ldiskfs block allocation statistics
- `IO` - Lustre target operations statistics
- `JBD` - `ldisfs` journal statistics
- `CLIENT` - Lustre OSC request statistics

To collect profile information:

1. **Begin collecting statistics on each node specified in the** `config.sh` **script.**

   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 25

Lustre Tuning

This chapter contains information about tuning Lustre for better performance and includes the following sections:

- Optimizing the Number of Service Threads
- Tuning LNET Parameters
- Lockless I/O Tunables
- Improving Lustre Performance When Working with Small Files
- Understanding Why Write Performance Is Better Than Read Performance

Note – Many options in Lustre are set by means of kernel module parameters. These parameters are contained in the `modprobe.conf` file.

25.1 Optimizing the Number of Service Threads

An OSS can have a minimum of 2 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.

25.1.1 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}
```

For details, see Section 31.2.12, “Setting MDS and OSS Thread Counts” on page 31-27.

25.1.2 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 31.2.12, “Setting MDS and OSS Thread Counts” on page 31-27.

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.

25.2 Tuning LNET Parameters

This section describes LNET tunables. that may be necessary on some systems to improve performance. To test the performance of your Lustre network, see Chapter 23: Testing Lustre Network Performance (LNET Self-Test).

25.2.1 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.

25.2.2 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 10Gb 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.

25.3 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.

25.4 Improving Lustre Performance When Working with Small Files

A Lustre environment where an application writes small file chunks from many clients to a single file will result in bad I/O performance. To improve Lustre's performance with small files:

- Have the application aggregate writes some amount before submitting them to Lustre. By default, Lustre 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.

25.5 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 Lustre

Part V provides information about troubleshooting a Lustre file system. You will find information in this section about:

Lustre Troubleshooting

Troubleshooting Recovery

Lustre Debugging

CHAPTER 26

Lustre Troubleshooting

This chapter provides information to troubleshoot Lustre, submit a Lustre bug, and Lustre performance tips. It includes the following sections:

- Lustre Error Messages
- Reporting a Lustre Bug
- Common Lustre Problems

26.1 Lustre Error Messages

Several resources are available to help troubleshoot Lustre. This section describes error numbers, error messages and logs.

26.1.1 Error Numbers

Error numbers for Lustre come from the Linux `errno.h`, and are located in `/usr/include/asm/errno.h`. Lustre 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 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 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.

26.1.2 Viewing Error Messages

As Lustre 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 Lustre action for a short period of time which, in turn, depends on the processes on the node to use Lustre. 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.

26.2 Reporting a Lustre Bug

If, after troubleshooting your Lustre system, you cannot resolve the problem, consider reporting a Lustre bug. The process for reporting a bug is described in the Lustre wiki topic Reporting Bugs.

You can also post a question to the lustre-discuss mailing list or search the lustre-discuss Archives for information about your issue.

A Lustre diagnostics tool is available for downloading at:
http://downloads.lustre.org/public/tools/lustre-diagnostics/

You can run this tool to capture diagnostics output to include in the reported bug. To run this tool, enter one of these commands:

```
# lustre-diagnostics -t <bugzilla bug #>

# lustre-diagnostics.
```

Output is sent directly to the terminal. Use normal file redirection to send the output to a file, and then manually attach the file to the bug you are submitting.

26.3 Common Lustre Problems

This section describes how to address common issues encountered with Lustre.

26.3.1 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 report this condition to the Lustre group, and we will investigate.

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.

26.3.2 OSTs Become Read-Only

If the SCSI devices are inaccessible to Lustre 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.

26.3.3 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 <OST device name or 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 <OST device name or 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.

26.3.4 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 Lustre 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 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.

---

1. The contents of the LAST_ID file must be accurate regarding the actual objects that exist on the OST.

For hex <–> 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/<mdsdev> /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
   ```

26.3.5 Handling/Debugging "Bind: Address already in use" Error

During startup, Lustre 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) which starts before Lustre 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 Lustre before starting any service that uses sunrpc.
- Use a port other than 988 for Lustre. This is configured in `/etc/modprobe.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 Lustre 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.

26.3.6 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, Lustre 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" on page 18-9.

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`.

26.3.7 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]
```

26.3.8 Handling Timeouts on Initial Lustre Setup

If you come across timeouts or hangs on the initial setup of your Lustre system, verify that name resolution for servers and clients is working correctly. Some distributions configure `/etc/hosts sts` 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)
```

26.3.9 Handling/Debugging "LustreError: xxx went back in time"

Each time Lustre 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 Lustre.
- As per the Lustre requirement, 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 Lustre 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.

26.3.10 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.

26.3.11 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.

26.3.12 Slowdown Occurs During Lustre Startup

When Lustre starts, the Lustre file system 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 Lustre 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.

26.3.13 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 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” on page 5-11.

26.3.14 Setting SCSI I/O Sizes

Some SCSI drivers default to a maximum I/O size that is too small for good Lustre performance. we have fixed quite a few drivers, but you may still find that some drivers give unsatisfactory performance with Lustre. 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 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 Lustre, you may want to examine the SCSI driver code.

CHAPTER 27

Troubleshooting Recovery

This chapter describes what to do if something goes wrong during recovery. It describes:

- Recovering from Errors or Corruption on a Backing File System
- Recovering from Corruption in the Lustre File System
- Recovering from an Unavailable OST

Note – For a description of how recovery is implemented in Lustre, see Chapter 30: Troubleshooting Recovery.

27.1 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 coherent. The backing file systems are never accessed directly from the client, so client crashes are not relevant.

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, Lustre can cope with any inconsistencies it finds on the disk and between other devices in the file system.

Note – `lfsck` is rarely required for Lustre 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 filesystem directly on the node with Lustre stopped (NOT via Lustre), 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 for Lustre). 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 filesystem using "prudent" answers
(usually 'y')
```

In addition, the e2fsprogs package contains the lfsck tool, which does distributed coherency checking for the Lustre file system after e2fsck has been run. Running lfsck is NOT required in a large majority of cases, at a small risk of having some leaked space in the file system. To avoid a lengthy downtime, it can be run (with care) after Lustre is started.

27.2 Recovering from Corruption in the Lustre File System

In cases where the MDS or an OST becomes corrupt, you can run a distributed check on the file system to determine what sort of problems exist. Use `lfsck` to correct any defects found.

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.
3. **Run a full e2fsck of the MDS to create a database for lfsck. You *must* use the `-n` option for a mounted file system, otherwise you will corrupt the file system.**

   ```
   e2fsck -n -v --mdsdb /tmp/mdsdb /dev/{mdsdev}
   ```

   The mdsdb file can grow fairly large, depending on the number of files in the file system (10 GB or more for millions of files, though the actual file size is larger because the file is sparse). It is quicker to write the file to a local file system due to seeking and small writes. Depending on the number of files, this step can take several hours to complete.

   Example

   ```
   e2fsck -n -v --mdsdb /tmp/mdsdb /dev/sdb
   e2fsck 1.39.cfs1 (29-May-2006)
   Warning: skipping journal recovery because doing a read-only
   filesystem check.
   lustre-MDT0000 contains a file system with errors, check forced.
   Pass 1: Checking inodes, blocks, and sizes
   MDS: ost_idx 0 max_id 288
   MDS: got 8 bytes = 1 entries in lov_objids
   MDS: max_files = 13
   MDS: num_osts = 1
   mds info db file written
   Pass 2: Checking directory structure
   Pass 3: Checking directory connectivity
   Pass 4: Checking reference counts
   Pass 5: Checking group summary information
   Free blocks count wrong (656160, counted=656058).
   Fix? no
   ```

```
Free inodes count wrong (786419, counted=786036).
Fix? no

Pass 6: Acquiring information for lfsck
MDS: max_files = 13
MDS: num_osts = 1
MDS: 'lustre-MDT0000_UUID' mdt idx 0: compat 0x4 rocomp 0x1 incomp
0x4
lustre-MDT0000: ******* WARNING: Filesystem still has errors
*******
           13 inodes used (0%)
              2 non-contiguous inodes (15.4%)
                  # of inodes with ind/dind/tind blocks: 0/0/0
130272 blocks used (16%)
           0 bad blocks
           1 large file
           296 regular files
           91 directories
           0 character device files
           0 block device files
           0 fifos
           0 links
           0 symbolic links (0 fast symbolic links)
           0 sockets
           --------
           387 files
```

4. **Make this file accessible on all OSTs, either by using a shared file system or copying the file to the OSTs. The `pdcp` command is useful here.**

   The `pdcp` command (installed with pdsh), can be used to copy files to groups of hosts. `Pdcp` is available here:

   http://sourceforge.net/projects/pdsh

5. **Run a similar e2fsck step on the OSTs. The `e2fsck --ostdb` command can be run in parallel on all OSTs.**

   ```
   e2fsck -n -v --mdsdb /tmp/mdsdb --ostdb /tmp/{ostNdb} \
   /dev/{ostNdev}
   ```

   The mdsdb file is read-only in this step; a single copy can be shared by all OSTs.

Note – If the OSTs do not have shared file system access to the MDS, a stub mdsdb file, {mdsdb}.mdshdr, is generated. This can be used instead of the full mdsdb file.

Example:

```
[root@oss161 ~]# e2fsck -n -v --mdsdb /tmp/mdsdb --ostdb \
/tmp/ostdb /dev/sda
e2fsck 1.39.cfs1 (29-May-2006)
Warning: skipping journal recovery because doing a read-only
filesystem check.
lustre-OST0000 contains a file system with errors, check forced.
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
Free blocks count wrong (989015, counted=817968).
Fix? no

Free inodes count wrong (262088, counted=261767).
Fix? no

Pass 6: Acquiring information for lfsck
OST: 'lustre-OST0000_UUID' ost idx 0: compat 0x2 rocomp 0 incomp 0x2
OST: num files = 321
OST: last_id = 321

lustre-OST0000: ******* WARNING: Filesystem still has errors
*******

           56 inodes used (0%)
           27 non-contiguous inodes (48.2%)
              # of inodes with ind/dind/tind blocks: 13/0/0
59561 blocks used (5%)
           0 bad blocks
           1 large file
           329 regular files
           39 directories
           0 character device files
           0 block device files
           0 fifos
           0 links
           0 symbolic links (0 fast symbolic links)
           0 sockets
           --------
           368 files
```

6. **Make the mdsdb file and all ostdb files available on a mounted client and run `lfsck` to examine the file system. Optionally, correct the defects found by lfsck.**

```
script /root/lfsck.lustre.log
lfsck -n -v --mdsdb /tmp/mdsdb --ostdb /tmp/{ost1db} /tmp/{ost2db}
... /lustre/mount/point
```

Example:

```
script /root/lfsck.lustre.log
lfsck -n -v --mdsdb /home/mdsdb --ostdb /home/{ost1db} \
/mnt/lustre/client/
MDSDB: /home/mdsdb
OSTDB[0]: /home/ostdb
MOUNTPOINT: /mnt/lustre/client/
MDS: max_id 288 OST: max_id 321
lfsck: ost_idx 0: pass1: check for duplicate objects
lfsck: ost_idx 0: pass1 OK (287 files total)
lfsck: ost_idx 0: pass2: check for missing inode objects
lfsck: ost_idx 0: pass2 OK (287 objects)
lfsck: ost_idx 0: pass3: check for orphan objects
[0] uuid lustre-OST0000_UUID
[0] last_id 288
[0] zero-length orphan objid 1
lfsck: ost_idx 0: pass3 OK (321 files total)
lfsck: pass4: check for duplicate object references
lfsck: pass4 OK (no duplicates)
lfsck: fixed 0 errors
```

By default, `lfsck` reports errors, but it does not repair any inconsistencies found. lfsck checks for three kinds of inconsistencies:

- Inode exists but has missing objects (dangling inode). This normally happens if there was a problem with an OST.
- Inode is missing but OST has unreferenced objects (orphan object). Normally, this happens if there was a problem with the MDS.
- Multiple inodes reference the same objects. This can happen if the MDS is corrupted or if the MDS storage is cached and loses some, but not all, writes.

If the file system is in use and being modified while the `--mdsdb` and `--ostdb` steps are running, `lfsck` may report inconsistencies where none exist due to files and objects being created/removed after the database files were collected. Examine the lfsck results closely. You may want to re-run the test.

27.2.1 Working with Orphaned Objects

The easiest problem to resolve is that of orphaned objects. When the `-l` option for `lfsck` is used, these objects are linked to new files and put into lost+found in the Lustre file system, where they can be examined and saved or deleted as necessary. If you are certain the objects are not useful, run `lfsck` with the `-d` option to delete orphaned objects and free up any space they are using.

To fix dangling inodes, use `lfsck` with the `-c` option to create new, zero-length objects on the OSTs. These files read back with binary zeros for stripes that had objects re-created. Even without `lfsck` repair, these files can be read by entering:

```
dd if=/lustre/bad/file of=/new/file bs=4k conv=sync,noerror
```

Because it is rarely useful to have files with large holes in them, most users delete these files after reading them (if useful) and/or restoring them from backup.

Note – You cannot write to the holes of such files without having `lfsck` re-create the objects. Generally, it is easier to delete these files and restore them from backup.

To fix inodes with duplicate objects, use `lfsck` with the `-c` option to copy the duplicate object to a new object and assign it to a file. One file will be okay and the duplicate will likely contain garbage. By itself, `lfsck` cannot tell which file is the usable one.

27.3 Recovering from an Unavailable OST

One of the most common problems encountered in a Lustre environment is when an OST becomes unavailable, because of 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, Lustre 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. The timeout length is determined by the obd_timeout parameter.

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 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 Section 30.4, "Version-based Recovery" on page 30-13.

CHAPTER 28

Lustre Debugging

This chapter describes tips and information to debug Lustre, and includes the following sections:

28.1 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.

28.1.1 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 28.2.2, "Using the `lctl` Tool to View Debug Messages" on page 28-7 and Section 36.3, "lctl" on page 36-4.
- **Lustre subsystem asserts** - A panic-style assertion (LBUG) in the kernel causes Lustre 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 26.1.2, "Viewing Error Messages" on page 26-3.
- **lfs** - This utility provides access to the extended attributes (EAs) of a Lustre file (along with other information). For more inforamtion about `lfs`, see Section 32.1, "lfs" on page 32-2.

28.1.2 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 28.2, “Lustre Debugging Procedures” on page 28-5 and Section 28.3, “Lustre Debugging for Developers” on page 28-14.

28.1.2.1 Tools for Administrators and Developers

Some general debugging tools provided as a part of the standard Linux distro 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. Lustre 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?. To download kdump, go to the Fedora Project 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 RHEL 5. For more information about netdump, see Red Hat, Inc.'s Network Console and Crash Dump Facility.

28.1.2.2 Tools for Developers

The tools described in this section may be useful for debugging Lustre in a development environment.

Of general interest is:

- **leak_finder.pl**. This program provided with Lustre 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.
- **VMware Server**. Virtualization platform available as free introductory software at Download VMware Server.
- **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.

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 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
  - Crash Usage: A Case Study from the white paper *Red Hat Crash Utility* by David Anderson
  - Kernel Trap forum entry: Linux: Kernel Crash Dumps
  - White paper: A Quick Overview of Linux Kernel Crash Dump Analysis

28.2 Lustre Debugging Procedures

The procedures below may be useful to administrators or developers debugging a Lustre files system.

28.2.1 Understanding the Lustre Debug Messaging Format

Lustre debug messages are categorized by originating sybsystem, message type, and locaton in the source code. For a list of subsystems and message types, see Section 28.2.1.1, "Lustre Debug Messages" on page 28-5.

Note – For a current list of subsystems and debug message types, see `lnet/include/libcfs/libcfs.h` in the Lustre tree

---

The lements of a Lustre debug message are described in Section 28.2.1.2, "Format of Lustre Debug Messages" on page 28-6.

28.2.1.1 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 in Lustre 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

Types	Description
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

28.2.1.2 Format of Lustre Debug Messages

Lustre uses the CDEBUG and CERROR macros to print the debug or error messages. To print the message, the CDEBUG macro uses `portals_debug_msg` (`portals/linux/oslib/debug.c`). The message format is described below, along with an example.

Parameter	Description
subsystem	800000
debug mask	000010
smp_processor_id	0
sec.used	10818808 47.677302
stack size	1204:
pid	2973:

Parameter	Description
host pid (if uml) or zero	31070:
(file:line #:functional())	(as_dev.c:144:create_write_buffers())
debug message	kmalloced '*obj': 24 at a375571c (tot 17447717)

28.2.1.3 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 (`/proc/sys/lnet/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.

28.2.2 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 36.3, “lctl” on page 36-4.

You can use `lctl` to:

- Obtain a list of all the types and subsystems:

```
lctl > debug_list <subs | 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 filename> [output filename]
  ```

  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.

28.2.2.1 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.
```

28.2.3 Dumping the Buffer to a File (debug_daemon)

The debug_daemon option is used by lctl to control the dumping of the debug_kernel buffer to a user-specified file. This functionality uses a kernel thread on top of debug_kernel, which works in parallel with the debug_daemon command.

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 CDEBUG 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 lctl control to start or stop the Lustre daemon from dumping the debug_buffer to a file. Users can also temporarily hold daemon from dumping the file. Use of the debug_daemon sub-command to lctl can provide the same function.

28.2.3.1 lctl debug_daemon Commands

This section describes lctl debug_daemon commands.

To initiate the debug_daemon to start dumping debug_buffer into a file., enter

```
$ lctl debug_daemon start [{file} {megabytes}]
```

The file can be a system default file, as shown in /proc/sys/lnet/debug_path. After Lustre starts, the default path is /tmp/lustre-log-$HOSTNAME. Users can specify a new filename for debug_daemon to output debug_buffer. The new file name shows up in /proc/sys/lnet/debug_path. Megabytes is the limitation of the file size in MBs.

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 order the log entries by time, run:

```
lctl debug_file {file} > {newfile}
```

The output is internally sorted by the lctl command using quicksort.

To completely shut down the debug_daemon operation and flush the file output, enter:

```
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 10 MB file.

```
#~/utils/lctl
```

To start the daemon to dump debug_buffer into a 40 MB /tmp/dump file, enter:

```
lctl > debug_daemon start /trace/log 40
```

To completely shut down the daemon, enter:

```
lctl > debug_daemon stop
```

To start another daemon with an unlimited file size, enter:

```
lctl > debug_daemon start /tmp/unlimited
```

The text message *** End of debug_daemon trace log *** appears at the end of each output file.

28.2.4 Controlling Information Written to the Kernel Debug Log

Masks are provided in `/proc/sys/lnet/subsystem_debug` and `/proc/sys/lnet/debug` to be used with the `systctl` command to determine what information is to be written to the debug log. The `subsystem_debug` mask determines the information written to the log based on the subsystem (such as iobdfilter, net, portals, or OSC). The debug mask controls information based on debug type (such as info, error, trace, or alloc).

To turn off Lustre debugging completely:

```
sysctl -w lnet.debug=0
```

To turn on full Lustre debugging:

```
sysctl -w lnet.debug=-1
```

To turn on logging of messages related to network communications:

```
sysctl -w lnet.debug=net
```

To turn on logging of messages related to network communications and existing debug flags:

```
sysctl -w lnet.debug=+net
```

To turn off network logging with changing existing flags:

```
sysctl -w lnet.debug=-net
```

The various options available to print to kernel debug logs are listed in `lnet/include/libcfs/libcfs.h`

28.2.5 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, aruguments, and return values.

To invoke `strace` on a program, enter:

```
$ strace <program> <args>
```

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> <args>
```

To redirect the strace output to a file, enter:

```
$ strace -o <filename> <program> <args>
```

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.

If the debugging is done in UML, save the traces on the host machine. In this example, hostfs is mounted on /r:

```
$ strace -o /r/tmp/vi.strace
```

28.2.6 Looking at Disk Content

In Lustre, 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 utility is written in C; it 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 Lustre file system, run:

```
$ lfs getstripe /mnt/lustre/frog
$
   obdix          objid
   0              17
   1              4
```

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 Lustre, 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 /tmp/ost1
   debugfs: cd O
   debugfs: cd 0                  /* for files in group 0 */
   debugfs: cd d<objid % 32>
   debugfs: stat <objid>          /* for getattr on object */
```

```
   debugfs: quit
## Suppose object id is 36, then follow the steps below:
   $ debugfs /tmp/ost1
   debugfs: cd O
   debugfs: cd 0
   debugfs: cd d4                /* objid % 32 */
   debugfs: stat 36              /* for getattr on obj 4*/
   debugfs: dump 36 /tmp/obj.36  /* dump contents of obj 4 */
   debugfs: quit
```

28.2.7 Finding the Lustre UUID of an OST

To determine the Lustre UUID of an obdfilter 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), use `debugfs` to get the `last_rcvd` file.

28.2.8 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:

```
sysctl -w lnet.printk=-1
```

This slows down the system dramatically. It is also possible to selectively enable or disable this capability for particular flags using:

```
sysctl -w lnet.printk=+vfstrace
sysctl -w lnet.printk=-vfstrace
```

It is possible to disable `warning`, `error` , and `console` messages, though it is strongly recommended to have something like `lctl debug_daemon` runing to capture this data to a local file system for debugging purposes.

28.2.9 Tracing Lock Traffic

Lustre has a specific debug type category for tracing lock traffic. Use:

```
lctl> filter all_types
lctl> show dlmtrace
lctl> debug_kernel [filename]
```

28.3 Lustre Debugging for Developers

The procedures in this section may be useful to developers debugging Lustre code.

28.3.1 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 descritions is provided in the table below.

Macro	Description
LBUG	A panic-style assertion in the kernel which causes Lustre 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, "This is my debug message: the number is %d\n", number).
CERROR	Behaves similarly to CDEBUG, but unconditionally prints the message in the debug log and to the console. This is appropriate for serious errors or fatal conditions: CERROR("Something very bad has happened, and the return code is %d.\n", rc);

Macro	Description
ENTRY and EXIT	Add messages to aid in call tracing (takes no arguments). When using these macros, cover all exit conditions to avoid confusion when the debug log reports that a function was entered, but never exited.
LDLM_DEBUG and LDLM_DEBUG_NOLOCK	Used when tracing MDS and VFS operations for locking. These macros build a thin trace that shows the protocol exchanges between nodes.
DEBUG_REQ	Prints information about the given ptlrpc_request structure.
OBD_FAIL_CHECK	Allows insertion of failure points into the Lustre code. This is useful to generate regression tests that can hit a very specific sequence of events. This works in conjunction with "sysctl -w lustre.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 lustre.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 "sysctl -w lustre.fail_loc=0".
OBD_FAIL_RAND	Has OBD_FAIL_CHECK fail randomly; on average every (1 / lustre.fail_val) times.
OBD_FAIL_SKIP	Has OBD_FAIL_CHECK succeed lustre.fail_val times, and then fail permanently or once with OBD_FAIL_ONCE.
OBD_FAIL_SOME	Has OBD_FAIL_CHECK fail lustre.fail_val times, and then succeed.

28.3.2 Accessing a 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.

A prlrpc 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's request buffer history if it has grown above req_buffer_history_max and its reqs are removed from the service's 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:

```
<seq>:<target NID>:<client ID>:<xid>:<length>:<phase> <svc specific>
```

Parameter	Description
seq	Request sequence number
target NID	Destination NID of the incoming request
client ID	Client PID and NID
xid	rq_xid

Parameter	Description
length	Size of the request message
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

28.3.3 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:

```
sysctl -w lnet.debug=+malloc
```

Then complete the following steps:

1. **Dump the log into a user-specified log file using `lctl` (see** Section 28.2.2, "Using the `lctl` Tool to View Debug Messages" on page 28-7**).**
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 241)
```

PART VI Reference

Part VI includes reference information on Lustre user utilities, configuration files and module parameters, programming interfaces, system configuration utilities, and system limits. You will find information in this section about:

CHAPTER 29

Installing Lustre from Source Code

If you need to build a customized Lustre server kernel or are using a Linux kernel that has not been tested with the version of Lustre you are installing, you may need to build and install Lustre from source code. This chapter describes:

29.1 Overview and Prerequisites

Lustre can be installed from either pre-built binary packages (RPMs) or freely-available source code. Installing from the package release is recommended unless you need to customize the Lustre server kernel or will be using an Linux kernel that has not been tested with Lustre. For a list of supported Linux distributions and architectures, see the topic Lustre_2.0 on the Lustre wiki. The procedure for installing Lustre from RPMs is describe in Chapter 8: Installing the Lustre Software.

To install Lustre from source code, the following are required:

- Linux kernel patched with Lustre-specific patches
- Lustre modules compiled for the Linux kernel
- Lustre utilities required for Lustre configuration

The installation procedure involves several steps:

- Patching the core kernel
- Configuring the kernel to work with Lustre
- Creating Lustre and kernel RPMs from source code.

Note – When using third-party network hardware with Lustre, the third-party modules (typically, the drivers) must be linked against the Linux kernel. The LNET modules in Lustre also need these references. To meet these requirements, a specific process must be followed to install and recompile Lustre. See Section 29.4, "Installing Lustre with a Third-Party Network Stack" on page 29-9, for an example showing how to install Lustre 1.6.6 using the Myricom MX 1.2.7 driver. The same process can be used for other third-party network stacks.

29.2 Patching the Kernel

If you are using non-standard hardware, plan to apply a Lustre patch, or have another reason not to use packaged Lustre binaries, you have to apply several Lustre patches to the core kernel and run the Lustre configure script against the kernel.

29.2.1 Introducing the Quilt Utility

To simplify the process of applying Lustre patches to the kernel, we recommend that you use the Quilt utility.

Quilt manages a stack of patches on a single source tree. A series file lists the patch files and the order in which they are applied. Patches are applied, incrementally, on the base tree and all preceding patches. You can:

- Apply patches from the stack (`quilt push`)
- Remove patches from the stack (`quilt pop`)
- Query the contents of the series file (`quilt series`), the contents of the stack (`quilt applied`, `quilt previous`, `quilt top`), and the patches that are not applied at a particular moment (`quilt next`, `quilt unapplied`).
- Edit and refresh (`update`) patches with Quilt, as well as revert inadvertent changes, and fork or clone patches and show the diffs before and after work.

A variety of Quilt packages (RPMs, SRPMs and tarballs) are available from various sources. Use the most recent version you can find. Quilt depends on several other utilities, e.g., the `coreutils` RPM that is only available in RedHat 9. For other RedHat kernels, you have to get the required packages to successfully install Quilt. If you cannot locate a Quilt package or fulfill its dependencies, you can build Quilt from a tarball, available at the Quilt project website:

http://savannah.nongnu.org/projects/quilt

For additional information on using Quilt, including its commands, see Introduction to Quilt and the quilt(1) man page.

29.2.2 Get the Lustre Source and Unpatched Kernel

The Lustre Engineering Team has targeted several Linux kernels for use with Lustre servers (MDS/OSS) and provides a series of patches for each one. The Lustre patches are maintained in the kernel_patch directory bundled with the Lustre source code.

Caution – Lustre contains kernel modifications which interact with storage devices and may introduce security issues and data loss if not installed, configured and administered correctly. Before installing Lustre, be cautious and back up ALL data.

Note – Each patch series has been tailored to a specific kernel version, and may or may not apply cleanly to other versions of the kernel.

To obtain the Lustre source and unpatched kernel:

1. **Verify that all of the Lustre installation requirements have been met.**

   For more information on these prerequisites, see:

   - Hardware requirements in Chapter 5: Setting Up a Lustre File System
   - Software and environmental requirements in Section 8.1, "Preparing to Install the Lustre Software" on page 8-2

2. **Download the Lustre source code.**

   On the Lustre download site, select a version of Lustre to download and then select "Source" as the platform.

3. **Download the unpatched kernel.**

   For convenience, Oracle maintains an archive of unpatched kernel sources at:

   http://downloads.lustre.org/public/kernels/

29.2.3 Patch the Kernel

This procedure describes how to use Quilt to apply the Lustre patches to the kernel. To illustrate the steps in this procedure, a RHEL 5 kernel is patched for Lustre 1.6.5.1.

1. **Unpack the Lustre source and kernel to separate source trees.**

   a. **Unpack the Lustre source.**

      For this procedure, we assume that the resulting source tree is in `/tmp/lustre-1.6.5.1`

   b. **Unpack the kernel.**

      For this procedure, we assume that the resulting source tree (also known as the destination tree) is in `/tmp/kernels/linux-2.6.18`

2. **Select a config file for your kernel, located in the `kernel_configs` directory (`lustre/kernel_patches/kernel_config`).**

   The `kernel_config` directory contains the `.config` files, which are named to indicate the kernel and architecture with which they are associated. For example, the configuration file for the 2.6.18 kernel shipped with RHEL 5 (suitable for i686 SMP systems) is `kernel-2.6.18-2.6-rhel5-i686-smp.config`.

3. **Select the `series` file for your kernel, located in the `series` directory (`lustre/kernel_patches/series`).**

   The `series` file contains the patches that need to be applied to the kernel.

4. **Set up the necessary symlinks between the kernel patches and the Lustre source.**

   This example assumes that the Lustre source files are unpacked under `/tmp/lustre-1.6.5.1` and you have chosen the `2.6-rhel5.series` file). Run:

   ```
   $ cd /tmp/kernels/linux-2.6.18
   $ rm -f patches series
   $ ln -s /tmp/lustre-1.6.5.1/lustre/kernel_patches/series/2.6-\
   rhel5.series ./series
   $ ln -s /tmp/lustre-1.6.5.1/lustre/kernel_patches/patches .
   ```

5. **Use Quilt to apply the patches in the selected `series` file to the unpatched kernel. Run:**

   ```
   $ cd /tmp/kernels/linux-2.6.18
   $ quilt push -av
   ```

   The patched destination tree acts as a base Linux source tree for Lustre.

29.3 Creating and Installing the Lustre Packages

After patching the kernel, configure it to work with Lustre, create the Lustre packages (RPMs) and install them.

1. **Configure the patched kernel to run with Lustre. Run:**

```
$ cd <path to kernel tree>
$ cp /boot/config-`uname -r` .config
$ make oldconfig || make menuconfig
$ make include/asm
$ make include/linux/version.h
$ make SUBDIRS=scripts
$ make include/linux/utsrelease.h
```

2. **Run the Lustre configure script against the patched kernel and create the Lustre packages.**

```
$ cd <path to lustre source tree>
$ ./configure --with-linux=<path to kernel tree>
$ make rpms
```

This creates a set of .rpms in `/usr/src/redhat/RPMS/<arch>` with an appended date-stamp. The SuSE path is `/usr/src/packages`.

Note – You do not need to run the Lustre configure script against an unpatched kernel.

Example set of RPMs:

```
lustre-1.6.5.1-\
2.6.18_53.xx.xx.el5_lustre.1.6.5.1.custom_20081021.i686.rpm

lustre-debuginfo-1.6.5.1-\
2.6.18_53.xx.xx.el5_lustre.1.6.5.1.custom_20081021.i686.rpm

lustre-modules-1.6.5.1-\
2.6.18_53.xx.xxel5_lustre.1.6.5.1.custom_20081021.i686.rpm

lustre-source-1.6.5.1-\
2.6.18_53.xx.xx.el5_lustre.1.6.5.1.custom_20081021.i686.rpm
```

Note – If the steps to create the RPMs fail, contact Lustre `Support` by reporting a bug. See Section 26.2, "Reporting a Lustre Bug" on page 26-4.

Note – Several features and packages are available that extend the core functionality of Lustre. These features/packages can be enabled at the build time by issuing appropriate arguments to the `configure` command. For a list of these features and packages, run `./configure –help` in the Lustre source tree. The `configs/` directory of the kernel source contains the config files matching each the kernel version. Copy one to `.config` at the root of the kernel tree.

3. **Create the kernel package. Navigate to the kernel source directory and run:**

   ```
   $ make rpm
   ```

   Example result:

   ```
   kernel-2.6.95.0.3.EL_lustre.1.6.5.1custom-1.i686.rpm
   ```

Note – Step 3 is only valid for RedHat and SuSE kernels. If you are using a stock Linux kernel, you need to get a script to create the kernel RPM.

4. **Install the Lustre packages.**

   Some Lustre packages are installed on servers (MDS and OSSs), and others are installed on Lustre clients. For guidance on where to install specific packages, see TABLE 8-1 in Section 8.1, "Preparing to Install the Lustre Software" on page 8-2. which lists required packages and for each package, where to install it. Depending on the selected platform, not all of the packages listed in TABLE 8-1 need to be installed.

Note – Running the patched server kernel on the clients is optional. It is not necessary unless the clients will be used for multiple purposes, for example, to run as a client and an OST.

Lustre packages should be installed in this order:

a. **Install the kernel, modules and ldiskfs packages.**

   Navigate to the directory where the RPMs are stored, and use the `rpm -ivh` command to install the `kernel`, `module` and `ldiskfs` packages.

   ```
   $ rpm -ivh kernel-lustre-smp-<ver> \
   kernel-ib-<ver> \
   lustre-modules-<ver> \
   lustre-ldiskfs-<ver>
   ```

b. Install the utilities/userspace packages.

Use the `rpm -ivh` command to install the utilities packages. For example:

```
$ rpm -ivh lustre-<ver>
```

c. Install the e2fsprogs package.

Make sure the e2fsprogs package is unpacked, and use the `rpm -i` command to install it. For example:

```
$ rpm -i e2fsprogs-<ver>
```

d. (Optional) If you want to add optional packages to your Lustre system, install them now.

5. **Verify that the boot loader (grub.conf or lilo.conf) has been updated to load the patched kernel.**

6. **Reboot the patched clients and the servers.**

   a. If you applied the patched kernel to any clients, reboot them.

   Unpatched clients do not need to be rebooted.

   b. Reboot the servers.

Once all the machines have rebooted, the next steps are to configure Lustre Networking (LNET) and the Lustre file system. See Chapter 10: Configuring Lustre.

29.4 Installing Lustre with a Third-Party Network Stack

When using third-party network hardware, you must follow a specific process to install and recompile Lustre. This section provides an installation example, describing how to install Lustre 1.6.6 while using the Myricom MX 1.2.7 driver. The same process is used for other third-party network stacks, by replacing MX-specific references in Step 2 with the stack-specific build and using the proper `--with` option when configuring the Lustre source code.

1. **Compile and install the Lustre kernel.**

   a. **Install the necessary build tools.**

   GCC and related tools must be installed. For more information, see Section 8.1.1, "Required Software" on page 8-2.

   ```
   $ yum install rpm-build redhat-rpm-config
   $ mkdir -p rpmbuild/{BUILD,RPMS,SOURCES,SPECS,SRPMS}
   $ echo '%_topdir %(echo $HOME)/rpmbuild' > .rpmmacros
   ```

   b. **Install the patched Lustre source code.**

   This RPM is available at the Lustre download site.

   ```
   $ rpm -ivh \
     kernel-lustre-source-2.6.18-92.1.10.el5_lustre.1.6.6.x86_64.rpm
   ```

   a. **Build the Linux kernel RPM.**

   ```
   $ cd /usr/src/linux-2.6.18-92.1.10.el5_lustre.1.6.6
   $ make distclean
   $ make oldconfig dep bzImage modules
   $ cp /boot/config-`uname -r` .config
   $ make oldconfig || make menuconfig
   $ make include/asm
   $ make include/linux/version.h
   $ make SUBDIRS=scripts
   $ make rpm
   ```

b. Install the Linux kernel RPM.

If you are building a set of RPMs for a cluster installation, this step is not necessary. Source RPMs are only needed on the build machine.

```
$ rpm -ivh \
  ~/rpmbuild/kernel-lustre-2.6.18-92.1.10.el5_\
  lustre.1.6.6.x86_64.rpm
$ mkinitrd /boot/2.6.18-92.1.10.el5_lustre.1.6.6
```

c. Update the boot loader (/etc/grub.conf) with the new kernel boot information.

```
$ /sbin/shutdown 0 -r
```

2. Compile and install the MX stack.

```
$ cd /usr/src/
$ gunzip mx_1.2.7.tar.gz (can be obtained from www.myri.com/scs/)
$ tar -xvf mx_1.2.7.tar
$ cd mx-1.2.7
$ ln -s common include
$ ./configure --with-kernel-lib
$ make
$ make install
```

3. Compile and install the Lustre source code.

a. Install the Lustre source (this can be done via RPM or tarball). The source file is available at the Lustre download site**. This example shows installation via the tarball.**

```
$ cd /usr/src/
$ gunzip lustre-1.6.6.tar.gz
$ tar -xvf lustre-1.6.6.tar
```

b. Configure and build the Lustre source code.

The `./configure --help` command shows a list of all of the `--with` options. All third-party network stacks are built in this manner.

```
$ cd lustre-1.6.6
$ ./configure --with-linux=/usr/src/linux \
  --with-mx=/usr/src/mx-1.2.7
$ make
$ make rpms
```

The `make rpms` command output shows the location of the generated RPMs

4. **Use the `rpm -ivh` command to install the RPMS.**

```
$ rpm -ivh \
  lustre-1.6.6-2.6.18_92.1.10.el5_lustre.1.6.6smp.x86_64.rpm
$ rpm -ivh \
  lustre-modules-1.6.6-2.6.18_92.1.10.el5_lustre.1.6.6\
  smp.x86_64.rpm
$ rpm -ivh \
  lustre-ldiskfs-3.0.6-2.6.18_92.1.10.el5_lustre.1.6.6\
  smp.x86_64.rpm
```

5. **Add the following lines to the /etc/modprobe.conf file.**

```
options kmxlnd hosts=/etc/hosts.mxlnd
options lnet networks=mx0(myri0),tcp0(eth0)
```

6. **Populate the myri0 configuration with the proper IP addresses.**

```
vim /etc/sysconfig/network-scripts/myri0
```

7. **Add the following line to the /etc/hosts.mxlnd file.**

```
$ IP HOST BOARD EP_ID
```

8. **Start Lustre.**

Once all the machines have rebooted, the next steps are to configure Lustre Networking (LNET) and the Lustre file system. See Chapter 10: Installing Lustre from Source Code.

CHAPTER 30

Lustre Recovery

This chapter describes how recovery is implemented in Lustre and includes the following sections:

- Recovery Overview
- Metadata Replay
- Reply Reconstruction
- Version-based Recovery
- Commit on Share

Note – Usually the Lustre recovery process is transparent. For information about troubleshooting recovery when something goes wrong, see Chapter 27: Lustre Recovery.

30.1 Recovery Overview

Lustre's recovery feature is responsible for dealing with node or network failure and returning the cluster to a consistent, performant state. Because Lustre 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

Currently, all Lustre 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.

For information on Lustre recovery, see Section 30.2, “Metadata Replay” on page 30-6. For information on recovering from a corrupt file system, see Section 30.5, “Commit on Share” on page 30-15. For information on resolving orphaned objects, a common issue after recovery, see Section 27.2.1, “Working with Orphaned Objects” on page 27-8.

30.1.1 Client Failure

Recovery from client failure in Lustre 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 30.1.5, “Network Partition” on page 30-5 describes this case in more detail.

30.1.2 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).

30.1.3 MDS Failure (Failover)

Highly-available (HA) Lustre 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 clients detect an MDS failure (either by timeouts of in-flight requests or idle-time ping messages), they 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 30.2, “Metadata Replay” on page 30-6.

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 filesystem 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.

30.1.4 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 Section 27.2.1, “Working with Orphaned Objects” on page 27-8.

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 very new versions of Lustre), 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:

```
lctl --device <OST 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.

30.1.5 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.

30.1.6 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 30.1.2, "Client Eviction" on page 30-3, 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

30.2 Metadata Replay

Highly available Lustre 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.

30.2.1 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.

30.2.2 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, monontonically 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 Lustre 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.

30.2.3 Replay and Resend

Lustre 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.

30.2.4 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.

30.2.5 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.

30.2.6 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.

30.2.7 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 30.4, “Version-based Recovery” on page 30-13.

30.2.8 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, Lustre does not do lock verification and just trusts clients to present an accurate lock state. This does not impart any security concerns since Lustre 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.

30.2.9 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.

30.3 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.

30.3.1 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.

30.3.2 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.

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.

Finding the Resource/fid

The file handle contains the resource/fid.

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.

30.4 Version-based Recovery

The Version-based Recovery (VBR) feature improves Lustre reliability in cases where client requests (RPCs) fail to replay during recovery[1].

In pre-VBR versions of Lustre, 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.

---

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.

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.
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.

30.4.1 VBR Messages

The VBR feature is built into the Lustre 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.

30.4.2 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.

30.5 Commit on Share

The commit-on-share (COS) feature makes Lustre 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.

30.5.1 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 30.4, “Version-based Recovery” on page 30-13 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.

30.5.2 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.

CHAPTER 31

LustreProc

The /proc file system acts as an interface to internal data structures in the kernel. The /proc variables can be used to control aspects of Lustre performance and provide information.

This chapter describes Lustre `/proc` entries and includes the following sections:

31.1 Proc Entries for Lustre

This section describes `/proc` entries for Lustre.

31.1.1 Locating Lustre File Systems and Servers

Use the proc files on the MGS to locate the following:

- All known file systems

```
# cat /proc/fs/lustre/mgs/MGS/filesystems
spfs
lustre
```

- The server names participating in a file system (for each file system that has at least one server running)

```
# cat /proc/fs/lustre/mgs/MGS/live/spfs
fsname: spfs
flags: 0x0     gen: 7
spfs-MDT0000
spfs-OST0000
```

All servers are named according to this convention: *<fsname>-<MDT|OST><XXXX>* This can be shown for live servers under `/proc/fs/lustre/devices`:

```
# cat /proc/fs/lustre/devices
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 lustre-mdtlov lustre-mdtlov_UUID 4
4 UP mds lustre-MDT0000 lustre-MDT0000_UUID 7
5 UP osc lustre-OST0000-osc lustre-mdtlov_UUID 5
6 UP osc lustre-OST0001-osc lustre-mdtlov_UUID 5
7 UP lov lustre-clilov-ce63ca00 08ac6584-6c4a-3536-2c6d-b36cf9cbdaa04
8 UP mdc lustre-MDT0000-mdc-ce63ca00
08ac6584-6c4a-3536-2c6d-b36cf9cbdaa05
9 UP osc lustre-OST0000-osc-ce63ca00
08ac6584-6c4a-3536-2c6d-b36cf9cbdaa05
10 UP osc lustre-OST0001-osc-ce63ca00
08ac6584-6c4a-3536-2c6d-b36cf9cbdaa05
```

Or from the device label at any time:

```
# e2label /dev/sda
lustre-MDT0000
```

31.1.2 Lustre Timeouts

Lustre uses two types of timeouts.

- LND timeouts that ensure point-to-point communications complete in finite time in the presence of failures. These timeouts are logged with the S_LND flag set. They may *not* be printed as console messages, so you should check the Lustre log for D_NETERROR messages, or enable printing of D_NETERROR messages to the console (echo + neterror > /proc/sys/lnet/printk).

  Congested routers can be a source of spurious LND timeouts. To avoid this, increase the number of LNET router buffers to reduce back-pressure and/or increase LND timeouts on all nodes on all connected networks. You should 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 that ensure Lustre RPCs complete in finite time in the presence of failures. These timeouts should *always* be printed as console messages. If Lustre timeouts are not accompanied by LNET timeouts, then you need to increase the lustre timeout on both servers and clients.

Specific Lustre timeouts are described below.

/proc/sys/lustre/timeout

This is the time period that a client waits for a server to complete an RPC (default is 100s). Servers wait half of 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 1 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."

Note – Lustre sends periodic 'PING' messages to servers with which it had no communication for a specified period of time. Any network activity on the file system that triggers network traffic toward servers also works as a health check.

/proc/sys/lustre/ldlm_timeout

This is the time period for which a server will wait for a client to reply to an initial AST (lock cancellation request) where 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 of the client timeout) to flush any dirty data and release the lock.

/proc/sys/lustre/fail_loc

This is the internal debugging failure hook.

See `lustre/include/linux/obd_support.h` for the definitions of individual failure locations. The default value is 0 (zero).

```
sysctl -w lustre.fail_loc=0x80000122 # drop a single reply
```

/proc/sys/lustre/dump_on_timeout

This triggers dumps of the Lustre debug log when timeouts occur. The default value is 0 (zero).

/proc/sys/lustre/dump_on_eviction

This triggers dumps of the Lustre debug log when an eviction occurs. The default value is 0 (zero). By default, debug logs are dumped to the `/tmp` folder; this location can be changed via `/proc`.

31.1.3 Adaptive Timeouts

Lustre offers an adaptive mechanism to set RPC timeouts. The adaptive timeouts feature (enabled, by default) causes servers to track actual RPC completion times, and to report estimated completion times for future RPCs back to clients. The clients use these estimates to set their future RPC timeout values. If server request processing slows down for any reason, the RPC completion estimates increase, and the clients allow more time for RPC completion.

If RPCs queued on the server approach their timeouts, then the server sends an early reply to the client, telling the client to allow more time. In this manner, clients avoid RPC timeouts and disconnect/reconnect cycles. Conversely, as a server speeds up, RPC timeout values decrease, allowing faster detection of non-responsive servers and faster attempts to reconnect to a server's failover partner.

In previous Lustre versions, the static `obd_timeout` (/proc/sys/lustre/timeout) value was used as the maximum completion time for all RPCs; this value also affected the client-server ping interval and initial recovery timer. Now, with adaptive timeouts, `obd_timeout` is only used for the ping interval and initial recovery estimate. When a client reconnects during recovery, the server uses the client's timeout value to reset the recovery wait period; i.e., the server learns how long the client had been willing to wait, and takes this into account when adjusting the recovery period.

31.1.3.1 Configuring Adaptive Timeouts

One of the goals of adaptive timeouts is to relieve users from having to tune the `obd_timeout` value. In general, `obd_timeout` should no longer need to be changed. However, there are several parameters related to adaptive timeouts that users can set. In most situations, the default values should be used.

The following parameters can be set persistently system-wide using `lctl conf_param` on the MGS. For example, `lctl conf_param work1.sys.at_max= 1500` sets the `at_max` value for all servers and clients using the `work1` file system.

Note – Nodes using multiple Lustre file systems must use the same at_* values for all file systems.)

Parameter	Description
at_min	Sets the minimum adaptive timeout (in seconds). Default value is 0. The `at_min` parameter is the minimum processing time that a server will report. Clients base their timeouts on this value, but they do not use this value directly. If you experience cases in which, for unknown reasons, the adaptive timeout value is too short and clients time out their RPCs (usually due to temporary network outages), then you can increase the `at_min` value to compensate for this. Ideally, users should leave `at_min` set to its default.
at_max	Sets the maximum adaptive timeout (in seconds). The `at_max` parameter is an upper-limit on the service time estimate, and is used as a 'failsafe' in case of rogue/bad/buggy code that would lead to never-ending estimate increases. If `at_max` is reached, an RPC request is considered 'broken' and should time out. Setting at_max to 0 causes adaptive timeouts to be disabled and the old fixed-timeout method (`obd_timeout`) to be used. This is the default value in Lustre 1.6.5. **NOTE**: It is possible that slow hardware might validly cause the service estimate to increase beyond the default value of `at_max`. In this case, you should increase `at_max` to the maximum time you are willing to wait for an RPC completion.
at_history	Sets a time period (in seconds) within which adaptive timeouts remember the slowest event that occurred. Default value is 600.

Parameter	Description
at_early_margin	Sets how far before the deadline Lustre sends an early reply. Default value is 5*.
at_extra	Sets the incremental amount of time that a server asks for, with each early reply. The server does not know how much time the RPC will take, so it asks for a fixed value. Default value is 30†. 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 client enters recovery status and reconnects to restore it to normal status. If you see multiple early replies for the same RPC asking for multiple 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	Sets the minimum lock enqueue time. Default value is 100. The `ldlm_enqueue` time is the maximum of the measured enqueue estimate (influenced by `at_min` and `at_max` parameters), multiplied by a weighting factor, and the `ldlm_enqueue_min` setting. LDLM lock enqueues were based on the `obd_timeout` value; now they have a dedicated minimum value. Lock enqueues increase as the measured enqueue times increase (similar to adaptive timeouts).

* This default was chosen as a reasonable time in which to send a reply from the point at which it was sent.

† This default was chosen as a balance between sending too many early replies for the same RPC and overestimating the actual completion time.

Adaptive timeouts are enabled, by default. To disable adaptive timeouts, at run time, set `at_max` to 0. On the MGS, run:

```
$ lctl conf_param <fsname>.sys.at_max=0
```

Note – Changing adaptive timeouts status at runtime may cause transient timeout, reconnect, recovery, etc.

31.1.3.2 Interpreting Adaptive Timeouts Information

Adaptive timeouts information can be read from `/proc/fs/lustre/*/timeouts` files (for each service and client) or with the `lctl` command.

This is an example from the `/proc/fs/lustre/*/timeouts` files:

```
cfs21:~# cat /proc/fs/lustre/ost/OSS/ost_io/timeouts
```

This is an example using the `lctl` command:

```
$ lctl get_param -n ost.*.ost_io.timeouts
```

This is the sample output:

```
service : cur 33  worst 34 (at 1193427052, 0d0h26m40s ago) 1 1 33 2
```

The `ost_io` service on this node is currently reporting an estimate of 33 seconds. The worst RPC service time was 34 seconds, and it happened 26 minutes ago.

The output also provides a history of service times. In the example, there are 4 "bins" of `adaptive_timeout_history`, with the maximum RPC time in each bin reported. In 0-150 seconds, the maximum RPC time was 1, with the same result in 150-300 seconds. From 300-450 seconds, the worst (maximum) RPC time was 33 seconds, and from 450-600s the worst time was 2 seconds. The current estimated service time is the maximum value of the 4 bins (33 seconds in this example).

Service times (as reported by the servers) are also tracked in the client OBDs:

```
cfs21:# 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 case, RPCs to portal 6, the OST_IO_PORTAL (see `lustre/include/lustre/lustre_idl.h`), shows the history of what the `ost_io` portal has reported as the service estimate.

Server statistic files also show the range of estimates in the normal min/max/sum/sumsq manner.

```
cfs21:~# lctl get_param mdt.*.mdt.stats
...
req_timeout               6 samples [sec] 1 10 15 105
...
```

31.1.4 LNET Information

This section describes `/proc` entries for LNET information.

/proc/sys/lnet/peers

Shows all NIDs known to this node and also gives information on the queue state.

```
# cat /proc/sys/lnet/peers
nid              refs   state  max    rtr    min    tx     min queue
0@lo             1      ~rtr   0      0      0      0      0   0
192.168.10.35@tcp1      ~rtr   8      8      8      8      6   0
192.168.10.36@tcp1      ~rtr   8      8      8      8      6   0
192.168.10.37@tcp1      ~rtr   8      8      8      8      6   0
```

The fields are explained below:

Field	Description
refs	A reference count (principally used for debugging)
state	Only valid to refer to routers. Possible values: • ~ rtr (indicates this node is not a router) • up/down (indicates this node is a router) • auto_fail must be enabled
max	Maximum number of concurrent sends from this peer
rtr	Routing buffer credits.
min	Minimum routing buffer credits seen.
tx	Send credits.
min	Minimum send credits seen.
queue	Total bytes in active/queued sends.

Credits work like a semaphore. At start they are initialized to allow a certain number of operations (8 in this example). LNET keeps a track of the minimum value so that you can see how congested a resource was.

If `rtr/tx` is less than `max`, there are operations in progress. The number of operations is equal to `rtr` or `tx` subtracted from `max`.

If `rtr/tx` is greater that `max`, there are operations blocking.

LNET also limits concurrent sends and router buffers allocated to a single peer so that no peer can occupy all these resources.

/proc/sys/lnet/nis

```
# cat /proc/sys/lnet/nis
nid                  refs   peer   max    tx     min
0@lo                 3      0      0      0      0
192.168.10.34@tcp    4      8      256    256    252
```

Shows the current queue health on this node. The fields are explained 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.

Subtracting `max - tx` yields the number of sends currently active. A large or increasing number of active sends may indicate a problem.

```
# cat /proc/sys/lnet/nis
nid                  refs   peer   max    tx     min
0@lo                 2      0      0      0      0
10.67.73.173@tcp     4      8      256    256    253
```

31.1.5 Free Space Distribution

Free-space stripe weighting, as set, gives a priority of "0" to free space (versus trying to place the stripes "widely" -- nicely distributed across OSSs and OSTs to maximize network balancing). To adjust this priority (as a percentage), use the `qos_prio_free` proc tunable:

```
$ cat /proc/fs/lustre/lov/<fsname>-mdtlov/qos_prio_free
```

Currently, the default is 90%. You can permanently set this value by running this command on the MGS:

```
$ lctl conf_param <fsname>-MDT0000.lov.qos_prio_free=90
```

Setting the priority to 100% means that OSS distribution does not count in the weighting, but the stripe assignment is still done via weighting. If OST 2 has twice as much free space as OST 1, it is twice as likely to be used, but it is NOT guaranteed to be used.

Also note that free-space stripe weighting does not activate until two OSTs are imbalanced by more than 20%. Until then, a faster round-robin stripe allocator is used. (The new round-robin order also maximizes network balancing.)

31.1.5.1 Managing Stripe Allocation

The MDS uses two methods to manage stripe allocation and determine which OSTs to use for file object storage:

- **QOS**

  Quality of Service (QOS) considers an OST's available blocks, speed, and the number of existing objects, etc. Using these criteria, the MDS selects OSTs with more free space more often than OSTs with less free space.

- **RR**

  Round-Robin (RR) allocates objects evenly across all OSTs. The RR stripe allocator is faster than QOS, and used often because it distributes space usage/load best in most situations, maximizing network balancing and improving performance.

Whether QOS or RR is used depends on the setting of the `qos_threshold_rr` proc tunable. The `qos_threshold_rr` variable specifies a percentage threshold where the use of QOS or RR becomes more/less likely. The `qos_threshold_rr` tunable can be set as an integer, from 0 to 100, and results in this stripe allocation behavior:

- If `qos_threshold_rr` is set to 0, then QOS is always used
- If `qos_threshold_rr` is set to 100, then RR is always used
- The larger the `qos_threshold_rr` setting, the greater the possibility that RR is used instead of QOS

31.2 Lustre I/O Tunables

The section describes I/O tunables.

/proc/fs/lustre/llite/<fsname>-<uid>/max_cache_mb

```
# cat /proc/fs/lustre/llite/lustre-ce63ca00/max_cached_mb 128
```

This tunable is the maximum amount of inactive data cached by the client (default is 3/4 of RAM).

31.2.1 Client I/O RPC Stream Tunables

The Lustre engine always attempts to pack an optimal amount of data into each I/O RPC and attempts to keep a consistent number of issued RPCs in progress at a time. Lustre exposes several tuning variables to adjust behavior according to network conditions and cluster size. Each OSC has its own tree of these tunables. For example:

```
$ ls -d /proc/fs/lustre/osc/OSC_client_ost1_MNT_client_2 /localhost
/proc/fs/lustre/osc/OSC_uml0_ost1_MNT_localhost
/proc/fs/lustre/osc/OSC_uml0_ost2_MNT_localhost
/proc/fs/lustre/osc/OSC_uml0_ost3_MNT_localhost
$ ls /proc/fs/lustre/osc/OSC_uml0_ost1_MNT_localhost
blocksizefilesfreemax_dirty_mb ost_server_uuid stats
```

... and so on.

RPC stream tunables are described below.

/proc/fs/lustre/osc/<object name>/max_dirty_mb

This tunable 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 512 are allowable. If 0 is given, no writes are cached. Performance suffers noticeably unless you use large writes (1 MB or more).

/proc/fs/lustre/osc/<object name>/cur_dirty_bytes

This tunable is a read-only value that returns the current amount of bytes written and cached on this OSC.

/proc/fs/lustre/osc/<object name>/max_pages_per_rpc

This tunable is the maximum number of pages that will undergo I/O in a single RPC to the OST. The minimum is a single page and the maximum for this setting is platform dependent (256 for i386/x86_64, possibly less for ia64/PPC with larger PAGE_SIZE), though generally amounts to a total of 1 MB in the RPC.

/proc/fs/lustre/osc/<object name>/max_rpcs_in_flight

This tunable is 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 32. If you are looking to improve small file I/O performance, increase the max_rpcs_in_flight value.

To maximize performance, the value for `max_dirty_mb` is recommended to be 4 * max_pages_per_rpc * `max_rpcs_in_flight`.

Note – The *<object name>* varies depending on the specific Lustre configuration. For *<object name>* examples, refer to the sample command output.

31.2.2 Watching the Client RPC Stream

The same directory contains a rpc_stats file with a histogram showing the composition of previous RPCs. The histogram can be cleared by writing any value into the rpc_stats file.

```
# cat /proc/fs/lustre/osc/spfs-OST0000-osc-c45f9c00/rpc_stats
snapshot_time:         1174867307.156604 (secs.usecs)
read RPCs in flight:   0
write RPCs in flight:  0
pending write pages:   0
pending read pages:    0
                    read                    write
pages per rpc       rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0

                    read                    write
rpcs in flight      rpcs   %  cum %   |  rpcs   %  cum %
0:                  0      0  0       |  0      0  0

                    read                    write
offset              rpcs   %  cum %   |  rpcs   %  cum %
0:                  0      0  0       |  0      0  0
```

Where:

Field	Description
{read,write} RPCs in flight	Number of read/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.
pending {read,write} pages	Number of pending read/write pages that have been queued for I/O in the OSC.
pages per RPC	When an RPC is sent, the number of pages it consists of is recorded (in order). A single page RPC increments the 0: row.
RPCs in flight	When an RPC is sent, the number of other RPCs that are pending is recorded. When the first RPC is sent, the 0: row is incremented. If the first RPC is sent while another is pending, the 1: row is incremented and so on. As each RPC *completes*, the number of pending RPCs is not tabulated. This table is a good 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.
offset	

31.2.3 Client Read-Write Offset Survey

The `offset_stats` parameter maintains statistics for occurrences where a series of read or write calls from a process did not access the next sequential location. The offset field is reset to 0 (zero) whenever a different file is read/written.

Read/write offset statistics are off, by default. The statistics can be activated by writing anything into the `offset_stats` file.

Example:

```
# cat /proc/fs/lustre/llite/lustre-f57dee00/rw_offset_stats
snapshot_time: 1155748884.591028 (secs.usecs)
R/W  PID  RANGE STARTRANGE ENDSMALLEST EXTENTLARGEST EXTENTOFFSET
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
```

Where:

Field	Description
R/W	Whether the non-sequential call was a read or write
PID	Process ID which made the read/write call.
Range Start/Range End	Range in which the read/write calls were sequential.
Smallest Extent	Smallest extent (single read/write) in the corresponding range.
Largest Extent	Largest extent (single read/write) in the corresponding range.
Offset	Difference from the previous range end to the current range start. For example, `Smallest-Extent` indicates that the writes in the range 100 to 1110 were sequential, with a minimum write of 10 and a maximum write of 500. This range was started with an offset of -150. That means this is the difference between the last entry's range-end and this entry's range-start for the same file. The `rw_offset_stats` file can be cleared by writing to it: `echo > /proc/fs/lustre/llite/lustre-f57dee00/rw_offset_stats`

31.2.4 Client Read-Write Extents Survey

Client-Based I/O Extent Size Survey

The `rw_extent_stats` histogram in the llite directory shows you the statistics for the sizes of the read-write I/O extents. This file does not maintain the per-process statistics.

Example:

```
$ cat /proc/fs/lustre/llite/lustre-ee5af200/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
```

The file can be cleared by issuing the following command:

```
$ echo > cat /proc/fs/lustre/llite/lustre-ee5af200/extents_stats
```

Per-Process Client I/O Statistics

The `extents_stats_per_process` file maintains the I/O extent size statistics on a per-process basis. So you can track the per-process statistics for the last `MAX_PER_PROCESS_HIST` processes.

Example:

```
$ cat /proc/fs/lustre/llite/lustre-ee5af200/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
```

31.2.5 Watching the OST Block I/O Stream

Similarly, there is a `brw_stats` histogram in the obdfilter directory which shows you the statistics for number of I/O requests sent to the disk, their size and whether they are contiguous on the disk or not.

```
cat /proc/fs/lustre/obdfilter/lustre-OST0000/brw_stats
snapshot_time:         1174875636.764630 (secs:usecs)
                              read                write
pages per brw       brws   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
discont pages       rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
discont blocks      rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
dio frags           rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
disk ios in flight  rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
io time (1/1000s)   rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
disk io size        rpcs   %  cum %   |  rpcs   %  cum %
1:                  0      0  0       |  0      0  0
                              read                write
```

The fields are explained below:

Field	Description
pages per brw	Number of pages per RPC request, which should match aggregate client `rpc_stats`.
discont pages	Number of discontinuities in the logical file offset of each page in a single RPC.
discont blocks	Number of discontinuities in the physical block allocation in the file system for a single RPC.

For each Lustre service, the following information is provided:

- Number of requests
- Request wait time (avg, min, max and std dev)
- Service idle time (% of elapsed time)

Additionally, data on each Lustre service is provided by service type:

- Number of requests of this type
- Request service time (avg, min, max and std dev)

31.2.6 Using File Readahead and Directory Statahead

Lustre 1.6.5.1 introduced file readahead and directory statahead functionality that read data into memory in anticipation of a process actually requesting the data. File readahead functionality reads file content data into memory. Directory statahead functionality reads metadata into memory. When readahead and/or statahead work well, a data-consuming process finds that the information it needs is available when requested, and it is unnecessary to wait for network I/O.

31.2.6.1 Tuning File Readahead

File readahead is triggered when two or more sequential reads by an application fail to be satisfied by 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.

/proc/fs/lustre/llite/<fsname>-<uid>/max_read_ahead_mb

This tunable controls the maximum amount of data readahead on a file. Files are read ahead in RPC-sized chunks (1 MB or the size of 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 there are sequential reads again. To disable readahead, set this tunable to 0. The default value is 40 MB.

/proc/fs/lustre/llite/<fsname>-<uid>/max_read_ahead_whole_mb

This tunable controls the maximum size of a file that is read in its entirety, regardless of the size of the read().

31.2.6.2 Tuning Directory Statahead

When the `ls -l` process opens a directory, its process ID is recorded. When the first directory entry is "stated" with this recorded process ID, a statahead thread is triggered which stats ahead all of the directory entries, in order. The `ls -l` process can use the stated directory entries directly, improving performance.

/proc/fs/lustre/llite/*/statahead_max

This tunable controls whether directory statahead is enabled and the maximum statahead count. By default, statahead is active.

To disable statahead, set this tunable to:

```
echo 0 > /proc/fs/lustre/llite/*/statahead_max
```

To set the maximum statahead count (`n`), set this tunable to:

```
echo n > /proc/fs/lustre/llite/*/statahead_max
```

The maximum value of `n` is 8192.

/proc/fs/lustre/llite/*/statahead_status

This is a read-only interface that indicates the current statahead status.

31.2.7 OSS Read Cache

The OSS read cache feature provides read-only caching of data on an OSS. This functionality uses the regular Linux page cache to store the data. Just like caching from a regular filesytem in Linux, OSS read cache uses as much physical memory as is allocated.

OSS read cache improves Lustre performance in these situations:

- Many clients are accessing the same data set (as in HPC applications and when diskless clients boot from Lustre)
- One client is storing data while another client is reading it (essentially 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)

31.2.7.1 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, you should use I/O patterns to determine the appropriate amount of memory for the cache; if the data is mostly reads, then more cache is required than for writes.

OSS read cache is enabled, by default, and managed by 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 requests. If read cache is enabled, this data stays in memory after the client's request is finished, and the OSS skips reading data from disk when subsequent read requests for the same are received. The read cache is managed by the Linux kernel globally across all OSTs on that OSS, and the least recently used cache pages will be dropped from memory when the amount of free memory is running low.

  If read cache is disabled (`read_cache_enable` = 0), then the OSS will discard the data after the client's read requests are serviced and, for subsequent read requests, the OSS must read the data from disk.

  To disable read cache on all 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, 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 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 writethrough cache is disabled (`writethrough_cache_enabled` = 0), then the OSS discards the data after the client's write request is completed, and for subsequent read request, or partial-page write request, 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 this 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 writethrough cache is advisable in the case where 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 writethrough cache on all OSTs of an OSS, run:

```
root@oss1# lctl set_param obdfilter.*.writethrough_cache_enable=0
```

To re-enable writethrough cache on one OST, run:

```
root@oss1# lctl set_param \
obdfilter.{OST_name}.writethrough_cache_enable=1
```

To check if writethrough cache is

```
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.

  This can be very 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**ilobytes, **M**egabytes, **G**igabytes, **T**erabytes, or **P**etabytes.

  To limit the maximum cached file size to 32MB 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
  ```

31.2.8 OSS Asynchronous Journal Commit

The OSS asynchronous journal commit feature synchronously 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 O_DIRECT writes, a journal flush is still forced.

When asynchronous journal commit 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 >=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 (jbd commit time 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 the intermediate data is lost. However, new OSS recovery functionality (introduced in 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.

To enable asynchronous journal commit, set the `sync_journal` parameter to zero (`sync_journal=0`):

```
$ lctl set_param obdfilter.*.sync_journal=0
obdfilter.lol-OST0001.sync_journal=0
```

By default, `sync_journal` is disabled (`sync_journal=1`), which forces a journal flush after every bulk write.

When asynchronous journal commit is used, clients keep a page reference until the journal transaction commits. This can cause problems when a client receives a blocking callback, because pages need to be removed from the page cache, but they cannot be removed because of the extra page reference.

This problem is solved by forcing a journal flush on lock cancellation. When this happens, the client is granted the metadata blocks that have hit the disk, and it can safely release the page reference before processing the blocking callback. The parameter which controls this action is `sync_on_lock_cancel`, which can be set to the following values:

`always`: Always force a journal flush on lock cancellation

`blocking`: Force a journal flush only when the local cancellation is due to a blocking callback

`never`: Do not force any journal flush

Here is an example of sync_on_lock_cancel being set not to force a journal flush:

```
$ lctl get_param obdfilter.*.sync_on_lock_cancel
obdfilter.lol-OST0001.sync_on_lock_cancel=never
```

By default, sync_on_lock_cancel is set to never, because asynchronous journal commit is disabled by default.

When asynchronous journal commit is enabled (sync_journal=0), sync_on_lock_cancel is automatically set to always, if it was previously set to never.

Similarly, when asynchronous journal commit is disabled, (sync_journal=1), sync_on_lock_cancel is enforced to never.

31.2.9 mballoc History

/proc/fs/ldiskfs/sda/mb_history

Multi-Block-Allocate (mballoc), enables Lustre to ask ldiskfs to allocate multiple blocks with a single request to the block allocator. Typically, an ldiskfs file system allocates only one block per time. Each mballoc-enabled partition has this file. This is sample output:

```
pid   inode    goal        result      found   grpscr  \   merge   tailbroken
2838  139267   17/12288/1  17/12288/1  1       0   0   \   M       1   8192
2838  139267   17/12289/1  17/12289/1  1       0   0   \   M       0   0
2838  139267   17/12290/1  17/12290/1  1       0   0   \   M       1   2
2838  24577    3/12288/1   3/12288/1   1       0   0   \   M       1   8192
2838  24578    3/12288/1   3/771/1     1       1   1   \           0   0
2838  32769    4/12288/1   4/12288/1   1       0   0   \   M       1   8192
2838  32770    4/12288/1   4/12289/1   13      1   1   \           0   0
2838  32771    4/12288/1   5/771/1     26      2   1   \           0   0
2838  32772    4/12288/1   5/896/1     31      2   1   \           1   128
2838  32773    4/12288/1   5/897/1     31      2   1   \           0   0
2828  32774    4/12288/1   5/898/1     31      2   1   \           1   2
2838  32775    4/12288/1   5/899/1     31      2   1   \           0   0
2838  32776    4/12288/1   5/900/1     31      2   1   \           1   4
2838  32777    4/12288/1   5/901/1     31      2   1   \           0   0
2838  32778    4/12288/1   5/902/1     31      2   1   \           1   2
```

The parameters are described below:

Parameter	Description
pid	Process that made the allocation.
inode	inode number allocated blocks
goal	Initial request that came to mballoc (group/block-in-group/number-of-blocks)
result	What mballoc actually found for this request.
found	Number of free chunks mballoc found and measured before the final decision.
grps	Number of groups mballoc scanned to satisfy the request.
cr	Stage at which mballoc found the result: **0** - best in terms of resource allocation. The request was 1MB or larger and was satisfied directly via the kernel buddy allocator. **1** - regular stage (good at resource consumption) **2** - fs is quite fragmented (not that bad at resource consumption) **3** - fs is very fragmented (worst at resource consumption)
queue	Total bytes in active/queued sends.
merge	Whether the request hit the goal. This is good as extents code can now merge new blocks to existing extent, eliminating the need for extents tree growth.
tail	Number of blocks left free after the allocation breaks large free chunks.
broken	How large the broken chunk was.

Most customers are probably interested in `found/cr`. If `cr` is 0 1 and `found` is less than 100, then mballoc is doing quite well.

Also, number-of-blocks-in-request (third number in the goal triple) can tell the number of blocks requested by the obdfilter. If the obdfilter is doing a lot of small requests (just few blocks), then either the client is processing input/output to a lot of small files, or something may be wrong with the client (because it is better if client sends large input/output requests). This can be investigated with the OSC `rpc_stats` or OST `brw_stats` mentioned above.

Number of groups scanned (grps column) should be small. If it reaches a few dozen often, then either your disk file system is pretty fragmented or mballoc is doing something wrong in the group selection part.

31.2.10 mballoc3 Tunables

Lustre version 1.6.1 and later includes mballoc3, which was built on top of mballoc2. By default, mballoc3 is enabled, and adds these features:

- Pre-allocation for single files (helps to resist fragmentation)
- Pre-allocation for a group of files (helps to pack small files into large, contiguous chunks)
- Stream allocation (helps to decrease the seek rate)

The following mballoc3 tunables are available:

Field	Description
stats	Enables/disables the collection of statistics. Collected statistics can be found in /proc/fs/ldiskfs2/<dev>/mb_history.
max_to_scan	Maximum number of free chunks that mballoc finds before a final decision to avoid livelock.
min_to_scan	Minimum number of free chunks that mballoc finds before a final decision. This is useful for a very small request, to resist fragmentation of big free chunks.
order2_req	For requests equal to 2^N (where N >= order2_req), a very fast search via buddy structures is used.
stream_req	Requests smaller or equal to this value are packed together to form large write I/Os.

The following tunables, providing more control over allocation policy, will be available in the next version:

Field	Description
stats	Enables/disables the collection of statistics. Collected statistics can be found in /proc/fs/ldiskfs2/<dev>/mb_history.
max_to_scan	Maximum number of free chunks that mballoc finds before a final decision to avoid livelock.
min_to_scan	Minimum number of free chunks that mballoc finds before a final decision. This is useful for a very small request, to resist fragmentation of big free chunks.
order2_req	For requests equal to 2^N (where N >= order2_req), a very fast search via buddy structures is used.

Field	Description
small_req **large_req**	All requests are divided into 3 categories: < small_req (packed together to form large, aggregated requests) < large_req (allocated mostly in linearly) > large_req (very large requests so the arm seek does not matter) The idea is that we try to pack small requests to form large requests, and then place all large requests (including compound from the small ones) close to one another, causing as few arm seeks as possible.
prealloc_table	The amount of space to preallocate depends on the current file size. The idea is that for small files we do not need 1 MB preallocations and for large files, 1 MB preallocations are not large enough; it is better to preallocate 4 MB.
group_prealloc	The amount of space preallocated for small requests to be grouped.

31.2.11 Locking

/proc/fs/lustre/ldlm/ldlm/namespaces/<OSC name|MDCname>/lru_size

The lru_size parameter is used to control the number of client-side locks in an LRU queue. LRU size is dynamic, based on load. This optimizes 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's RAM. The default limit is 50 locks/1 MB of RAM. If there is too much memory pressure, then the LRU size is shrunk. The number of locks on the server is limited to {number of OST/MDT on node} * {number of clients} * {client lru_size}.

- 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. (In Lustre 1.6.5.1 and later, LRU sizing is enabled, by default.)
- To specify a maximum number of locks, set the lru_size parameter to a value > 0 (former numbers are okay, 100 * CPU_NR). We recommend 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:

```
$ lctl set_param ldlm.namespaces.<osc_name|mdc_name>.lru_size=clear
```

If you shrink the LRU size below the number of existing unused locks, then the unused locks are canceled immediately. Use echo clear to cancel all locks without changing the value.

Note – Currently, the `lru_size` parameter can only be set temporarily with `lctl set_param`; it cannot be set permanently.

To disable LRU sizing, run this command on the Lustre clients:

```
$ lctl set_param ldlm.namespaces.*osc*.lru_size=$((NR_CPU*100))
```

Replace `NR_CPU` value with the number of CPUs on the node.

To determine the number of locks being granted:

```
$ lctl get_param ldlm.namespaces.*.pool.limit
```

31.2.12 Setting MDS and OSS Thread Counts

MDS and OSS thread counts (minimum and maximum) can be set via the `{min,max}_thread_count` tunable. For each service, a new `/proc/fs/lustre/{service}/*/thread_{min,max,started}` entry is created. The tunable, `{service}.thread_{min,max,started}`, can be used to set the minimum and maximum thread counts or get the current number of running threads for the following services.

Service	Description
mdt.MDS.mds	normal metadata ops
mdt.MDS.mds_readpage	metadata readdir
mdt.MDS.mds_setattr	metadata setattr
ost.OSS.ost	normal data
ost.OSS.ost_io	bulk data IO
ost.OSS.ost_create	OST object pre-creation service
ldlm.services.ldlm_canceld	DLM lock cancel
ldlm.services.ldlm_cbd	DLM lock grant

- To temporarily set this tunable, run:

  ```
  # lctl {get,set}_param {service}.thread_{min,max,started}
  ```

- To permanently set this tunable, run:

  ```
  # lctl conf_param {service}.thread_{min,max,started}
  ```

The following examples show how to set thread counts and get the number of running threads for the `ost_io` service.

- To get the number of running threads, run:

  ```
  # lctl get_param ost.OSS.ost_io.threads_started
  ```

  The command output will be similar to this:

  ```
  ost.OSS.ost_io.threads_started=128
  ```

- To set the maximum number of threads (512), run:

  ```
  # lctl get_param ost.OSS.ost_io.threads_max
  ```

  The command output will be:

  ```
  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
  ```

  The command output will be:

  ```
  ost.OSS.ost_io.threads_max=256
  ```

- To check if the new `threads_max` setting is active, run:

  ```
  # lctl get_param ost.OSS.ost_io.threads_max
  ```

  The command output will be similar to this:

  ```
  ost.OSS.ost_io.threads_max=256
  ```

Note – Currently, the maximum thread count setting is advisory because Lustre does not reduce the number of service threads in use, even if that number exceeds the `threads_max` value. Lustre does not stop service threads once they are started.

31.3 Debug

/proc/sys/lnet/debug

By default, Lustre generates a detailed log of all operations to aid in debugging. The level of debugging can affect the performance or speed you achieve with Lustre. Therefore, it is useful to reduce this overhead by turning down the debug level[1] to improve performance. Raise the debug level when you need to collect the logs for debugging problems. The debugging mask can be set with "symbolic names" instead of the numerical values that were used in prior releases. The new symbolic format is shown in the examples below.

Note – All of the commands below must be run as root; note the # nomenclature.

To verify the debug level used by examining the sysctl that controls debugging, run:

```
# sysctl lnet.debug
lnet.debug = ioctl neterror warning error emerg ha config console
```

To turn off debugging (except for network error debugging), run this command on all concerned nodes:

```
# sysctl -w lnet.debug="neterror"
lnet.debug = neterror
```

To turn off debugging completely, run this command on all concerned nodes:

```
# 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 above collect enough high-level information to aid debugging, but they do not cause any serious performance impact.

To clear all flags and set new ones, run:

```
# sysctl -w lnet.debug="warning"
lnet.debug = warning
```

1. This controls the level of Lustre debugging kept in the internal log buffer. It does not alter the level of debugging that goes to syslog.

To add new flags to existing ones, prefix them with a "+":

```
# sysctl -w lnet.debug="+neterror +ha"
lnet.debug = +neterror +ha

# sysctl lnet.debug
lnet.debug = neterror warning ha
```

To remove flags, prefix them with a "-":

```
# sysctl -w lnet.debug="-ha"
lnet.debug = -ha

# sysctl lnet.debug
lnet.debug = neterror warning
```

You can verify and change the debug level using the /proc interface in Lustre. To use the flags with /proc, run:

```
# cat /proc/sys/lnet/debug
neterror warning

# echo "+ha" > /proc/sys/lnet/debug

# cat /proc/sys/lnet/debug
neterror warning ha

# echo "-warning" > /proc/sys/lnet/debug

# cat /proc/sys/lnet/debug
neterror ha
```

/proc/sys/lnet/subsystem_debug

This controls the debug logs[3] for subsystems (see S_* definitions).

/proc/sys/lnet/debug_path

This indicates the location where debugging symbols should be stored for gdb. The default is set to `/r/tmp/lustre-log-localhost.localdomain`.

These values can also be set via `sysctl -w lnet.debug={value}`

Note – The above entries only exist when Lustre has already been loaded.

/proc/sys/lnet/panic_on_lbug

This causes Lustre to call "panic" when it detects an internal problem (an LBUG); 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 Lustre.

/proc/sys/lnet/upcall

This allows you to specify the path to the binary which will be invoked when an LBUG is encountered. This binary is called with four parameters. The first one is the string "LBUG". The second one is the file where the LBUG occurred. The third one is the function name. The fourth one is the line number in the file.

31.3.1 RPC Information for Other OBD Devices

Some OBD devices maintain a count of the number of RPC events that they process. Sometimes these events are more specific to operations of the device, like *llite*, than actual raw RPC counts.

```
$ find /proc/fs/lustre/ -name stats
/proc/fs/lustre/osc/lustre-OST0001-osc-ce63ca00/stats
/proc/fs/lustre/osc/lustre-OST0000-osc-ce63ca00/stats
/proc/fs/lustre/osc/lustre-OST0001-osc/stats
/proc/fs/lustre/osc/lustre-OST0000-osc/stats
/proc/fs/lustre/mdt/MDS/mds_readpage/stats
/proc/fs/lustre/mdt/MDS/mds_setattr/stats
/proc/fs/lustre/mdt/MDS/mds/stats
/proc/fs/lustre/mds/lustre-MDT0000/exports/ab206805-0630-6647-8543-d
24265c91a3d/stats
/proc/fs/lustre/mds/lustre-MDT0000/exports/08ac6584-6c4a-3536-2c6d-b
36cf9cbdaa0/stats
/proc/fs/lustre/mds/lustre-MDT0000/stats
/proc/fs/lustre/ldlm/services/ldlm_canceld/stats
/proc/fs/lustre/ldlm/services/ldlm_cbd/stats
/proc/fs/lustre/llite/lustre-ce63ca00/stats
```

31.3.1.1 Interpreting OST Statistics

Note – See also Section 36.6, "llobdstat" on page 36-14 and Section 12.3, "CollectL" on page 12-8.

The OST .../stats files can be used to track client statistics (client activity) for each OST. It is possible to get a periodic dump of values from these file (for example, every 10 seconds), that show the RPC rates (similar to iostat) by using the llstat.pl tool:

```
# llstat /proc/fs/lustre/osc/lustre-OST0000-osc/stats

/usr/bin/llstat: STATS on 09/14/07
/proc/fs/lustre/osc/lustre-OST0000-osc/stats on 192.168.10.34@tcp
snapshot_time           1189732762.835363
ost_create              1
ost_get_info            1
ost_connect             1
ost_set_info            1
obd_ping                212
```

To clear the statistics, give the -c option to llstat.pl. To specify how frequently the statistics should be cleared (in seconds), use an integer for the -i option. This is sample output with -c and -i10 options used, providing statistics every 10s):

```
$ 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.CountCur.Rate#EventsUnit\  last    min avg     max     stddev
req_waittime8       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_avail8       0       8   [bufs]  511\    63  63.88   64      0.35
ost_write   8       0       8   [bytes]1697677\72914212209.6238757991874.29

/proc/fs/lustre/ost/OSS/ost_io/stats @ 1181074113.290180
Name        Cur.CountCur.Rate#EventsUnit \     lastmin avg max     stddev
req_waittime31      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_avail31      3       39  [bufs]  1977\       63  63.79   64      0.41
ost_write   30      3       38  [bytes]10284679\15019315325.16910694197776.51

/proc/fs/lustre/ost/OSS/ost_io/stats @ 1181074123.325560
Name        Cur.CountCur.Rate#Events Unit \    last minavgmax     stddev
req_waittime21          2       60  [usec] 14970\ 34784.32122451878.66
```

```
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_avail21         2       60  [bufs] 1341\ 6363.82   64  0.39
ost_write  21          2       59  [bytes]7648424\ 15019332725.08910694
180397.87
```

Where:

Parameter	Description
Cur. Count	Number of events of each type sent in the last interval (in this example, 10s)
Cur. Rate	Number of events per second in the last interval
#Events	Total number of such events since the system started
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 all cases)

The events common to all services are:

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:

Parameter	Description
ldlm_enqueue	Time it takes to enqueue a lock (this includes file open on the MDS)
mds_reint	Time it takes to process an MDS modification record (includes create, mkdir, unlink, rename and setattr)

31.3.1.2 Interpreting MDT Statistics

Note – See also Section 36.6, "llobdstat" on page 36-14 and Section 12.3, "CollectL" on page 12-8.

The MDT .../stats files can be used to track MDT statistics for the MDS. Here is sample output for an MDT stats file:

```
# cat /proc/fs/lustre/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 32

User Utilities

This chapter describes user utilities and includes the following sections:

Note – User utility man pages are found in the `lustre/doc` folder.

32.1 lfs

The `lfs` utility can be used for user configuration routines and monitoring.

Synopsis

```
lfs
lfs changelog [--follow] <mdtname> [startrec [endrec]]
lfs changelog_clear <mdtname> <id> <endrec>
lfs check <mds|osts|servers>
lfs df [-i] [-h] [--pool]-p <fsname>[.<pool>] [path]
lfs find [[!] --atime|-A [-+]N] [[!] --mtime|-M [-+]N]
   [[!] --ctime|-C [-+]N] [--maxdepth|-D N] [--name|-n <pattern>]
   [--print|-p] [--print0|-P] [[!] --obd|-O <uuid[s]>]
   [[!] --size|-S [+-]N[kMGTPE]] --type |-t {bcdflpsD}]
   [[!] --gid|-g|--group|-G <gname>|<gid>]
   [[!] --uid|-u|--user|-U <uname>|<uid>]
   <dirname|filename>
lfs osts [path]
lfs getstripe [--obd|-O <uuid>] [--quiet|-q] [--verbose|-v]
              [--count|-c] [--index|-i | --offset|-o]
              [--size|-s] [--pool|-p] [--directory|-d]
              [--recursive|-r] <dirname|filename> ...
lfs setstripe [--size|-s stripe_size] [--count|-c stripe_cnt]
              [--index|-i|--offset|-o start_ost_index]
              [--pool|-p <pool>]
              <dirname|filename>
lfs setstripe -d <dir>
lfs poollist <filesystem>[.<pool>]|<pathname>
lfs quota [-q] [-v] [-o obd_uuid|-I ost_idx|-i mdt_idx] [-u <uname>|
-u <uid>|-g <gname>| -g <gid>] <filesystem>
lfs quota -t <-u|-g> <filesystem>
lfs quotacheck [-ug] <filesystem>
lfs quotachown [-i] <filesystem>
lfs quotaon [-ugf] <filesystem>
lfs quotaoff [-ug] <filesystem>
lfs quotainv [-ug] [-f] <filesystem>
```

```
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>]
             <filesystem>
lfs setquota <-u|--user|-g|--group> <uname|uid|gname|gid>
             [-b <block-softlimit>] [-B <block-hardlimit>]
             [-i <inode-softlimit>] [-I <inode-hardlimit>]
             <filesystem>
lfs setquota -t <-u|-g>
             [--block-grace <block-grace>]
             [--inode-grace <inode-grace>]
             <filesystem>
lfs setquota -t <-u|-g>
             [-b <block-grace>] [-i <inode-grace>]
             <filesystem>
lfs help
```

Note – In the above example, the `<filesystem>` parameter refers to the mount point of the Lustre file system. The default mount point is `/mnt/lustre`.

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`.

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.

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 current last record. Changelog consumers must be registered on the MDT node using `lctl`.
check	
	Displays the status of the MDS or OSTs (as specified in the command) or all servers (MDS and OSTs).
df [-i] [-h] [--pool|-p <fsname>[.<pool>] [path]	
	Report file system disk space usage or inode usage (with **-i**) of each MDT/OST or a subset of OSTs if a pool is specified with **-p**. By default, prints the usage of all mounted Lustre file systems. Otherwise, if **path** is specified, prints only the usage of that file system. If **-h** is given, 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.
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, while - before a numeric value means files with the parameter OR LESS.

Option	Description
--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). Lustre 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.
--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 [Solaris]).
--uid	File has a specific numeric user ID.
--user	File is owned by a specific user (numeric user ID is 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.

Option		Description
	--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 Lustre-mounted file system is specified, then only OSTs belonging to this file system are displayed.
getstripe		
		Lists striping information for a given filename or directory. By default, the stripe count, 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.
	--obd <uuid>	
		Lists only files that have an object on a specific OST.
	--quiet	
		Lists details about the file's object ID information.
	--verbose	
		Prints additional striping information.
	--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`).

Option	Description
--recursive	Recurses into all sub-directories.
setstripe	Creates a new file or sets the directory default with specific striping parameters.†
--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, and normally results in a file with 80 stripes.
--size stripe_size*	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 36.3, "lctl" on page 36-4) 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 file system.pool.

Option	Description
quota [-q] [-v] [-o obd_uuid|-i mdt_idx|-I ost_idx] [-u|-g <uname>|<uid>|<gname>|<gid>] <filesystem>	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 the number of columns is consistent. The **-v** option provides more verbose (per-OBD statistics) output.
quota -t <-u|-g> <filesystem>	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] <filesystem>	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] <filesystem>	Turns on file system quotas. Options specify quota for users (-u), groups (-g), and force (-f).
quotaoff [-ugf] <filesystem>	Turns off file system quotas. Options specify quota for users (-u), groups (-g), and force (-f).
quotainv [-ug] [-f] <filesystem>	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.

Option	Description
setquota <-u|-g> <uname>|<uid>|<gname>|<gid> [--block-softlimit <block-softlimit>] [--block-hardlimit <block-hardlimit>] [--inode-softlimit <inode-softlimit>] [--inode-hardlimit <inode-hardlimit>] <filesystem>	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.‡ 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 Examples below.
setquota -t <-u|-g> [--block-grace <block-grace>] [--inode-grace <inode-grace>] <filesystem>	Sets file system quota grace times for users or groups. Grace time is specified in "XXwXXdXXhXXmXXs" format or as an integer seconds value. See Examples below.
help	Provides brief help on various lfs arguments.
exit/quit	Quits the interactive **lfs** session.

* The default stripe-size is 0. The default stripe-start is -1. Do NOT confuse them! If you set stripe-start to 0, all new file creations occur on OST 0 (seldom a good idea).

† The file cannot exist prior to using setstripe. A directory must exist prior to using setstripe.

‡ The old setquota interface is supported, but it may be removed in a future Lustre release.

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
```

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/lustre
```

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
```

See Also

`lctl` in Section 36.3, “lctl” on page 36-4

32.2 lfs_migrate

The `lfs_migrate` utility is a simple tool to migrate files between Lustre OSTs.

Synopsis

```
lfs_migrate [-c|-s] [-h] [-l] [-n] [-y] [file|directory ...]
```

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 that 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).

Options

Options supporting `lfs_migrate` are described below.

Option	Description
-c	Compares file data after migrate (default value, use **-s** to disable).
-s	Skips file data comparison after migrate (use **-c** to enable).
-h	Displays help information.
-l	Migrates 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 prints the names of files to be migrated.
-q	Runs quietly (does not print filenames or status).
-y	Answers 'y' to usage warning without prompting (for scripts).

Examples

```
$ lfs_migrate /mnt/lustre/file
```

Rebalances all files in `/mnt/lustre/dir`.

```
$ lfs find /test -obd test-OST004 -size +4G | lfs_migrate -y
```

Migrates files in `/test` filesystem on OST004 larger than 4 GB in size.

See Also

`lfs` in Section 32.1, “lfs” on page 32-2

32.3 lfsck

Lfsck ensures that objects are not referenced by multiple MDS files, that there are no orphan objects on the OSTs (objects that do not have any file on the MDS which references them), and that all of the objects referenced by the MDS exist. Under normal circumstances, Lustre maintains such coherency by distributed logging mechanisms, but under exceptional circumstances that may fail (e.g. disk failure, file system corruption leading to e2fsck repair). To avoid lengthy downtime, you can also run lfsck once Lustre is already started.

The e2fsck utility is run on each of the local MDS and OST device file systems and verifies that the underlying ldiskfs is consistent. After e2fsck is run, lfsck does distributed coherency checking for the Lustre file system. In most cases, e2fsck is sufficient to repair any file system issues and lfsck is not required.

Synopsis

```
lfsck [-c|--create] [-d|--delete] [-f|--force] [-h|--help]
[-l|--lostfound] [-n|--nofix] [-v|--verbose] --mdsdb
mds_database_file --ostdb ost1_database_file [ost2_database_file...]
<filesystem>
```

Note – As shown, the <filesystem> parameter refers to the Lustre file system mount point. The default mount point is `/mnt/lustre`.

Note – For `lfsck`, database filenames must be provided as absolute pathnames. Relative paths do not work, the databases cannot be properly opened.

Options

Options supporting `lfsck` are described below.

Option	Description
-c	Creates (empty) missing OST objects referenced by MDS inodes.
-d	Deletes orphaned objects from the file system. Since objects on the OST are often only one of several stripes of a file, it can be difficult to compile multiple objects together in a single, usable file.
-h	Prints a brief help message.
-l	Puts orphaned objects into a lost+found directory in the root of the file system.
-n	Performs a read-only check; does not repair the file system.
-v	Verbose operation - more verbosity by specifying the option multiple times.
--mdsdb mds_database_file	MDS database file created by running e2fsck --mdsdb mds_database_file <device> on the MDS backing device. This is required.
--ostdb ost1_database_file [ost2_database_file...]	OST database files created by running e2fsck --ostdb ost_database_file <device> on each of the OST backing devices. These are required unless an OST is unavailable, in which case all objects thereon are considered missing.

Description

The lfsck utility is used to check and repair the distributed coherency of a Lustre file system. If an MDS or an OST becomes corrupt, run a distributed check on the file system to determine what sort of problems exist. Use `lfsck` to correct any defects found.

For more information on using `e2fsck` and `lfsck`, including examples, see Section 30.5, “Commit on Share” on page 30-15. For information on resolving orphaned objects, see Section 27.2.1, “Working with Orphaned Objects” on page 27-8.

32.4 Filefrag

The e2fsprogs package contains the filefrag tool which reports the extent of file fragmentation.

Synopsis

```
filefrag [ -belsv ] [ files...  ]
```

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 – Lustre 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 Lustre, the extents are printed in device offset order, not logical offset order.

1. The default mode is faster than the verbose/extent mode.

Options

Options supporting filefrag are described below.

Option	Description
-b	Uses the 1024-byte blocksize for the output. By default, this blocksize is used by Lustre, 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.
-v	Uses the verbose mode when checking file fragmentation.

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:  start..end:length: device:flags:
0:     0..         49151:     212992..   262144:    49152:  0:      remote
1:     49152..     73727:     270336..   294912:    24576:  0:      remote
2:     73728..     76799:     24576..    27648:     3072:   0:      remote
3:     0..         57343:     196608..   253952:    57344:  1:      remote
4:     57344..     65535:     139264..   147456:    8192:   1:      remote
5:     65536..     76799:     163840..   175104:    11264:  1:      remote
/mnt/lustre/foo: 6 extents found
```

32.5 Mount

Lustre uses the standard mount(8) Linux command. When mounting a Lustre file system, mount(8) executes the `/sbin/mount.lustre` command to complete the mount. The `mount` command supports these Lustre-specific options:

Server options	Description
abort_recov	Aborts recovery when starting a target
nosvc	Starts only MGS/MGC servers
exclude	Starts with a dead OST

Client options	Description
flock	Enables/disables flock support
user_xattr/nouser_xattr	Enables/disables user-extended attributes
retry=	Number of times a client will retry to mount the file system

32.6 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 33

Lustre Programming Interfaces

This chapter describes public programming interfaces to control various aspects of Lustre from userspace. These interfaces are generally not guaranteed to remain unchanged over time, although we will make an effort to notify the user community well in advance of major changes. This chapter includes the following section:

- User/Group Cache Upcall
- `l_getgroups` Utility

Note – Lustre programming interface man pages are found in the `lustre/doc` folder.

33.1 User/Group Cache Upcall

This section describes user and group upcall.

Note – For information on a universal UID/GID, see Section 8.1.2, "Environmental Requirements" on page 8-5.

33.1.1 Name

Use `/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_upcall` to look up a given user's group membership.

33.1.2 Description

The group upcall file contains the path to an executable that, when installed, is invoked to resolve a numeric UID to a group membership list. This utility should complete the `mds_grp_downcall_data` data structure (see Section 33.1.4, "Data Structures" on page 33-4) and write it to the `/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_info` pseudo-file.

For a sample upcall program, see `lustre/utils/l_getgroups.c` in the Lustre source distribution.

33.1.2.1 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, supplementary groups will be added as supplied by the client (as they are now).
- 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 authentication configuration, and can be local /etc/passwd, NIS, LDAP, etc. 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 `echo {path} > /proc/fs/lustre/mds/{mdsname}/group_upcall`
- The Lustre administrator can specify permissions for a specific UID by configuring `/etc/lustre/perm.conf` on the MDS. As commented in `lustre/utils/l_getidentity.c`

```
/*
* permission file format is like this:
* {nid} {uid} {perms}
*
* '*' nid means any nid
* '*' uid means any uid
* the valid values for perms are:
* setuid/setgid/setgrp/rmtacl -- enable corresponding perm
* nosetuid/nosetgid/nosetgrp/normtacl -- disable corresponding perm
* they can be listed together, seperated by ',',
* when perm and noperm are in the same line (item), noperm is
preferential,
* when they are in different lines (items), the latter is
preferential,
* '*' nid is as default perm, and is not preferential.
*/
```

  Currently, rmtacl/normtacl can be ignored (part of security functionality), and used for remote clients. The `/usr/sbin/l_getidentity` utility can parse `/etc/lustre/perm.conf` to obtain permission mask for specified UID.

- To avoid repeated upcalls, the MDS caches supplemental group information. Use `/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_expire` to set the cache time (default is 600 seconds). The kernel waits for the upcall to complete (at most, 5 seconds) and takes the "failure" behavior as described. Set the wait time in `/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_acquire_expire` (default is 15 seconds). Cached entries are flushed by writing to `/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_flush`.

33.1.3 Parameters

- Name of the MDS service
- Numeric UID

33.1.4 Data Structures

```
struct identity_downcall_data {
     __u32    idd_magic;
     __u32    idd_err;
     __u32    idd_uid;
     __u32    idd_gid;
     __u32    idd_nperms;
     struct perm_downcall_data idd_perms[N_PERMS_MAX];
     __u32    idd_ngroups;
     __u32    idd_groups[0];
};
```

33.2 l_getgroups Utility

The l_getgroups utility handles Lustre user/group cache upcall.

Synopsis

```
l_getgroups [-v] [-d|mdsname] uid]
l_getgroups [-v] -s
```

Description

The group upcall file contains the path to an executable that, when properly installed, is invoked to resolve a numeric UID to a group membership list. This utility should complete the mds_grp_downcall_data data structure (see Data structures) and write it to the /proc/fs/lustre/mds/mds-service/group_info pseudo-file.

l_getgroups is the reference implementation of the user/group cache upcall.

Files

/proc/fs/lustre/mds/mds-service/group_upcall

CHAPTER 34

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:

- llapi_file_create
- llapi_file_get_stripe
- llapi_file_open
- llapi_quotactl
- llapi_path2fid
- Example Using the llapi Library

Note – Lustre programming interface man pages are found in the `lustre/doc` folder.

34.1 llapi_file_create

Use `llapi_file_create` to set Lustre properties for a new file.

Synopsis

```
#include <lustre/liblustreapi.h>
#include <lustre/lustre_user.h>

int llapi_file_create(char *name, long stripe_size,
int stripe_offset, int stripe_count, int stripe_pattern);
```

Description

The `llapi_file_create()` function sets a file descriptor's Lustre 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

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_pattern);
```

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;
}
```

34.2 llapi_file_get_stripe

Use `llapi_file_get_stripe` to get striping information for a file or directory on a Lustre file system.

Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <liblustre.h>
#include <lustre/lustre_idl.h>
#include <lustre/liblustreapi.h>
#include <lustre/lustre_user.h>

int llapi_file_get_stripe(const char *path, void *lum);
```

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 version.
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.

Option	Description

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;`
	`__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.

Return Values

`llapi_file_get_stripe()` returns:

0 On success

!= 0 On failure, *errno* is set appropriately

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

Examples

```
#include <sys/vfs.h>
#include <liblustre.h>
#include <lnet/lnetctl.h>
#include <obd.h>
#include <lustre_lib.h>
#include <lustre/liblustreapi.h>
#include <obd_lov.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;
}
```

34.3 llapi_file_open

The `llapi_file_open` command opens (or creates) a file or device on a Lustre filesystem.

Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <liblustre.h>
#include <lustre/lustre_idl.h>
#include <lustre/liblustreapi.h>
#include <lustre/lustre_user.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);
```

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 filesystem.

Option	Description
flags	Can be a combination of O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_NOCTTY, 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.

Option	Description
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 version of Lustre, only LOV_PATTERN_RAID0 is available. The default value is 0.

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

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 filesystem.

Example

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <liblustre.h>
#include <lustre/lustre_idl.h>
#include <lustre/liblustreapi.h>
#include <lustre/lustre_user.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;
}
```

34.4 llapi_quotactl

Use `llapi_quotactl` to manipulate disk quotas on a Lustre file system.

Synopsis

```
#include <liblustre.h>
#include <lustre/lustre_idl.h>
#include <lustre/liblustreapi.h>
#include <lustre/lustre_user.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];
};
```

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. *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.
LUSTRE_Q_QUOTAOFF	Turns off quotas for a Lustre file system. *qc_type* is USRQUOTA, GRPQUOTA or UGQUOTA (both user and group quota). This call is restricted to the super user privilege.
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.

Option	Description
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 Lustre version.
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 Lustre version and must be zeroed.

Return Values

`llapi_quotactl()` returns:

0 On success

-1 On failure and sets error number (errno) to indicate the error

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.

34.5 llapi_path2fid

Use `llapi_path2fid` to get the FID from the pathname.

Synopsis

```
#include <lustre/liblustreapi.h>
#include <lustre/lustre_user.h>

int llapi_path2fid(const char *path, unsigned long long *seq, unsigned
long *oid, unsigned long *ver)
```

Description

The `llapi_path2fid` function returns the FID (sequence : object ID : version) for the pathname.

Return Values

`llapi_path2fid` returns:

0	On success
non-zero value	On failure

34.6 Example Using the llapi Library

Use `llapi_file_create` to set Lustre properties for a new file. For a synopsis and description of `llapi_file_create` and examples of how to use it, see Chapter 34: Setting Lustre Properties in a C Program (llapi).

You can set striping from inside programs like ioctl. To compile the sample program, you need to download `libtest.c` and `liblustreapi.c` files from the Lustre source tree.

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 - simple code examples of liblustreapi functions
 */

#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <lustre/liblustreapi.h>
#include <lustre/lustre_user.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 liblustre 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 required */
#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(errno));
               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,
strerror(-rc));
                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,
strerror(errno));
              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(errno));
        }
        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(errno));
                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 object group %llu\n", lump->lmm_object_gr);
            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_idx, lump->lmm_objects[i].l_object_id);
        }
```

```
            free(lump);
            return rc;

}
/* Ping all OSTs that belong to this filesysem */

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);
            rintf("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 getsrtipe\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/
```

See Also

`llapi_file_create` in Section 34.1, “llapi_file_create” on page 34-2

`llapi_file_get_stripe` in Section 34.2, “llapi_file_get_stripe” on page 34-4

`llapi_file_open` in Section 34.3, “llapi_file_open” on page 34-9

`llapi_quotactl` in Section 34.4, “llapi_quotactl” on page 34-12

CHAPTER 35

Configuration Files and Module Parameters

This section describes configuration files and module parameters and includes the following sections:

35.1 Introduction

LNET network hardware and routing are now configured via module parameters. Parameters should be specified in the `/etc/modprobe.conf` file, for example:

```
alias lustre llite
options lnet networks=tcp0,elan0
```

The above option specifies that this node should use all the available TCP and Elan interfaces.

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).

Under Linux 2.6, LNET configuration parameters can be viewed under `/sys/module/`; generic and acceptor parameters under LNET, and LND-specific parameters under the name of the corresponding LND.

Under Linux 2.4, sysfs is not available, but the LND-specific parameters are accessible via equivalent paths under `/proc`.

Important: All old (pre v.1.4.6) Lustre configuration lines should be removed from the module configuration files and replaced with the following. Make sure that CONFIG_KMOD is set in your linux.config so LNET can load the following modules it needs. The basic module files are:

`modprobe.conf` (for Linux 2.6)

```
alias lustre llite
options lnet networks=tcp0,elan0
```

`modules.conf` (for Linux 2.4)

```
alias lustre llite
options lnet networks=tcp0,elan0
```

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.)

35.2 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 modprobe.conf.lnet included from modprobe.conf makes distributing the configuration much easier.
- If you set `config_on_load=1`, LNET starts at modprobe time rather than waiting for Lustre 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.

35.2.1 LNET Options

This section describes LNET options.

35.2.1.1 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 – Lustre ignores the loopback interface (lo0), but Lustre use 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="tcp198.129.135.* 192.128.88.98; \
             elan  198.128.88.98 198.129.135.3;" \
             routes="tcp  1022@elan# Elan NID of router;\
             elan  198.128.88.98@tcp # TCP NID of router  "
```

35.2.1.2 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.

35.2.1.3 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>{<w><nid>}
```

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@tcp1"
```

The hopcount is 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.

Equivalent entries are resolved in favor of the route with the shorter hopcount. 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.

35.2.1.4 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 (< 1023). • **all** - Accept connections from any TCP port. 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.

35.2.2 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.

Variable	Description
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.
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.

35.2.3 Portals LND (Linux)

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 a

(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.
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.

35.2.4 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 (< 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 > 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 Lustre. 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 36

System Configuration Utilities

This chapter includes system configuration utilities and includes the following sections:

Note – The system configuration utilities man pages are found in the `lustre/doc` folder.

36.1 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.lustre.org/public/tools/e2fsprogs/

Synopsis

```
e2scan [options] [-f file] block_device
```

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.

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.

36.2 l_getidentity

The l_getidentity utility handles Lustre user / group cache upcall.

Synopsis

```
l_getidentity {mdtname} {uid}
```

Description

The group upcall file contains the path to an executable file that, when properly installed, is invoked to resolve a numeric UID to a group membership list. This utility should complete the mds_grp_downcall_data structure and write it to the `/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_info` pseudo-file.

The `l_getidentity` utility is the reference implementation of the user or group cache upcall.

Options

Option	Description
mdtname	
	Metadata server target name
uid	
	User identifier

Files

The `l_getidentity` files are located at:

```
/proc/fs/lustre/mdt/${FSNAME}-MDT{xxxx}/identity_upcall
```

36.3 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.

Synopsis

```
lctl
lctl --device <devno> <command [args]>
```

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 nid
help
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, 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.

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 to set temporary parameters (mapping 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 ldlm.namespaces.*osc*.lru_size=$((NR_CPU*100))
```

Many permanent parameters can be set with `lctl conf_param`. In general, `lctl conf_param` 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:

```
<obd|fsname>.<obdtype>.<proc_file_name>=<value>)
```

For example:

```
$ lctl conf_param testfs-MDT0000.mdt.group_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.

To get current Lustre parameter settings, use the `lctl get_param` command with this syntax:

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

For example:

```
$ lctl get_param -n ost.*.ost_io.timeouts
```

To list Lustre parameters that are available to set, use the `lctl list_param` command, with this syntax:

```
lctl list_param [-n] <obdtype>.<obdname>
```

For example:

```
$ lctl list_param obdfilter.lustre-OST0000
```

For more informaiton on using `lctl` to set temporary and permanent parameters, see Section 13.8.3, “Setting Parameters with lctl” on page 13-9.

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*>	Check's 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.
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] <param_path ...>*	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] <param_path ...>*	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] <param_path=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. 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 13.8.3.2, "Setting Permanent Parameters" on page 13-10.

Option	Description
	-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}_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` 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 <*file name*> <*device node*>	Attaches a regular Lustre file to a block device. If the device node does not exist, **lctl** creates it. We recommend that you create the device node by **lctl** since the emulator uses a dynamical major number.
blockdev_detach <*device node*>	Detaches the virtual block device.
blockdev_info <*device node*>	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*> *[output]*	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 <*subs/types*>	Lists all subsystem and debug types.
modules <*path*>	Provides GDB-friendly module information.

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.

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
```

See Also

`mkfs.lustre` in Section 36.14, “mkfs.lustre” on page 36-28

`llobdstat` in Section 36.6, “llobdstat” on page 36-14

`lfs` in Section 32.1, “lfs” on page 32-2

36.4 ll_decode_filter_fid

The `ll_decode_filter_fid` utility displays the Lustre object ID and MDT parent FID.

Synopsis

```
ll_decode_filter_fid object_file [object_file ...]
```

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 filesystem 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.

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 hexdecimal 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.

See Also

`ll_recover_lost_found_objs` in Section 36.5, "ll_recover_lost_found_objs" on page 36-12

36.5 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.

Synopsis

```
$ ll_recover_lost_found_objs [-hv] -d directory
```

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.

Options

Option	Description
-h	Prints a help message
-v	Increases verbosity
-d *directory*	Sets the lost and found directory path

Example

```
ll_recover_lost_found_objs -d /mnt/ost/lost+found
```

36.6 llobdstat

The `llobdstat` utility displays OST statistics.

Synopsis

```
llobdstat ost_name [interval]
```

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.

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, punch: 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
```

Files

```
/proc/fs/lustre/obdfilter/<ostname>/stats
```

36.7 llog_reader

The `llog_reader` utility parses Lustre's on-disk configuration logs.

Synopsis

```
llog_reader filename
```

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.

See Also

`tunefs.lustre` in Section 36.18, "tunefs.lustre" on page 36-38

36.8 llstat

The `llstat` utility prints Lustre statistics.

Synopsis

```
llstat [-c] [-g] [-i interval] stats_file
```

Description

The `llstat` utility can display statistics from any of several Lustre statistics files that share a common format, updated every `interval` seconds. Use CTRL-C to stop statistics printing.

Option	Description
-c	Clears the statistics file first.
-i interval	Polling period (in seconds).
-g	Graphable output format.
-h	Displays help information.
stats_file	Either the full path to a statistics file or the shorthand: MDS or OST.

Example

Monitors statistics on /proc/fs/lustre/ost/OSS/ost/stats at one (1) second intervals:

```
llstat -i 1 ost
```

Files

```
/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
```

36.9 llverdev

The `llverdev` verifies a block device is functioning properly over its full size.

Synopsis

`llverdev [-c` *`chunksize`*`] [-f] [-h] [-o` *`offset`*`] [-l] [-p] [-r] [-t` *`timestamp`*`] [-v] [-w]` *`device`*

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.

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 filesystem (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).

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
```

36.10 lshowmount

The `lshowmount` utility shows Lustre exports.

Synopsis

```
lshowmount [-ehlv]
```

Description

The `lshowmount` utility shows the hosts that have Lustre mounted to a server. Ths utility looks for exports from the MGS, MDS, and obdfilter.

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.

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
```

36.11 lst

The `lst` utility starts LNET self-test.

Synopsis

```
lst
```

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.

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.

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
```

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
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
```

36.12 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.

36.13 lustre_rsync

The `lustre_rsync` utility synchronizes (replicates) a Lustre file system to a target file system.

Synopsis

```
lustre_rsync --source|-s <src> --target|-t <tgt>
          --mdt|-m <mdt> [--user|-u <user id>]
          [--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>
```

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.

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=<*user id*>	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 overriden. 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.

Option	Description
--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.

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
```

See Also

`lctl` in Section 36.3, “lctl” on page 36-4

`lfs` in Section 32.1, “lfs” on page 32-2

36.14 mkfs.lustre

The `mkfs.lustre` utility formats a disk for a Lustre service.

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.

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 13.8.1, “Setting Parameters with mkfs.lustre” on page 13-9.

Option	Description
--backfstype=*fstype*	Forces a particular format for the backing file system (ldiskfs).
--comment=*comment*	Sets a user comment about this disk, ignored by Lustre.
--device-size=*KB*	Sets the device size for loop devices.
--dryrun	Only prints what would be done; it does not affect the disk.
--failnode=*nid,...*	Sets the NID(s) of a failover partner. This option can be repeated as needed.
--fsname=*filesystem_name*	The Lustre file system of which this service/node will be a part. The default file system name is “lustre”. **NOTE**: The file system name is limited to 8 characters.
--index=*index*	Forces a particular OST or MDT index.
--mkfsoptions=*opts*	Formats options for the backing file system. For example, ldiskfs options could be set here.
--mountfsoptions=*opts*	

Option	Description
	Sets the mount options used when the backing file system is mounted. **CAUTION**: 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,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.
--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	

Option	Description
	Used to optimize the MDT's inode size.
--verbose	
	Prints more information.

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 --ost --mgsnode=cfs21@tcp0 /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
```

See Also

`llobdstat` in Section 36.6, "llobdstat" on page 36-14

`lfs` in Section 32.1, "lfs" on page 32-2

36.15 mount.lustre

The `mount.lustre` utility starts a Lustre client or target service.

Synopsis

```
mount -t lustre [-o options] directory
```

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(8)` 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
<mgsspec>:/<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 mounted client file system appears in fstab(5) and is usable, like any local file system, and provides a full POSIX-compliant interface.
<disk_device>	Starts the target service defined by the `mkfs.lustre` command on the physical disk *disk_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.

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 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.

In addition to the standard mount options and backing disk type (e.g. ldiskfs) options, Lustre understands the following server-specific options:

Option	Description
nosvc	Starts only the MGC (and MGS, if co-located) for a target service, not the actual service.
nomgs	Starts 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 900 seconds (15 minutes).

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
```

See Also

`mkfs.lustre` in Section 36.14, “mkfs.lustre” on page 36-28

`tunefs.lustre` in Section 36.18, “tunefs.lustre” on page 36-38

`lctl` in Section 36.3, “lctl” on page 36-4

`lfs` in Section 32.1, “lfs” on page 32-2

36.16 plot-llstat

The `plot-llstat` utility plots Lustre statistics.

Synopsis

```
plot-llstat results_filename [parameter_index]
```

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.

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

Example

```
llstat -i2 -g -c lustre-OST0000 > log
plot-llstat log 3
```

36.17 routerstat

The `routerstat` utility prints Lustre router statistics.

Synopsis

```
routerstat [interval]
```

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).

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

Files

The `routerstat` utility extracts statistics data from:

```
/proc/sys/lnet/stats
```

36.18 tunefs.lustre

The `tunefs.lustre` utility modifies configuration information on a Lustre target disk.

Synopsis

```
tunefs.lustre [options] device
```

Description

`tunefs.lustre` is used to modify configuration information on a Lustre target disk. This includes upgrading old (pre-Lustre 1.6) disks. 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 only affect a file system 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.group_upcall=NONE /dev/sda1
```

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.
--failnode=*nid,...*	Sets the NID(s) of a failover partner. This option can be repeated as needed.
--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. **CAUTION**: 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 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.

Option	Description
--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: * Unmount all clients on the file system * Unmount the MDT and all OSTs on the file system * Run **tunefs.lustre --writeconf <device>** on every server * Mount the MDT and OSTs * Mount the clients

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
```

See Also

`mkfs.lustre` in Section 36.14, "mkfs.lustre" on page 36-28

`mount.lustre` in Section 36.15, "mount.lustre" on page 36-32

`lctl` in Section 36.3, "lctl" on page 36-4

`lfs` in Section 32.1, "lfs" on page 32-2

36.19 Additional System Configuration Utilities

This section describes additional system configuration utilities for Lustre.

36.19.1 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 (improved in Lustre 1.6), handles a wider range of `/proc` 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.

36.19.2 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 1.6 included 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
  ```

36.19.3 Testing / Debugging Utilities

Lustre offers the following test and debugging utilities.

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.

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 a file can be written to (a write cache requirement), OSTs reserve ("grants"), chunks of space for each newly-created file. A grant may cause an OST to report it is out of space, even though there is enough space on the disk, because the space is "reserved" by other files. `Loadgen` estimates the number of simultaneous open files as disk size divided by grant size and reports that number when write tests start.

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

llog_reader

The `llog_reader` utility translates a Lustre configuration log into human-readable form.

Synopsis

```
llog_reader filename
```

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

`tunefs.lustre` in Section 36.18, "tunefs.lustre" on page 36-38

lr_reader

The `lr_reader` utility translates a last received (`last_rcvd`) file into human-readable form.

The following utilites are part of the Lustre I/O kit. For more information, see Chapter 24: Benchmarking Lustre Performance (Lustre I/O Kit).

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.

---

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.

ior-survey

The ior-`survey` utility is a script used to run the IOR benchmark. Lustre includes IOR version 2.8.6.

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.

stats-collect

The `stats-collect` utility contains scripts used to collect application profiling information from Lustre clients and servers.

36.19.4 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.

CAUTION: 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.

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 00
```

Glossary

A

ACL Access Control List - An extended attribute associated with a file which contains authorization directives.

Administrative OST failure A configuration directive given to a cluster to declare that an OST has failed, so errors can be immediately returned.

C

CMD Clustered metadata, a collection of metadata targets implementing a single file system namespace.

Completion Callback An RPC made by an OST or MDT to another system, usually a client, to indicate that the lock request is now granted.

Configlog An llog file used in a node, or retrieved from a management server over the network with configuration instructions for Lustre systems at startup time.

Configuration Lock A lock held by every node in the cluster to control configuration changes. When callbacks are received, the nodes quiesce their traffic, cancel the lock and await configuration changes after which they reacquire the lock before resuming normal operation.

D

Default stripe pattern Information in the LOV descriptor that describes the default stripe count 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 which can be used during read and write system calls. It bypasses the kernel. I/O cache to memory copy of data between kernel and application memory address spaces.

Directory stripe descriptor An extended attribute that describes the default stripe pattern for files underneath that directory.

E

EA Extended Attribute. A small amount of data which can be retrieved through a name associated with a particular inode. Lustre uses EAa to store striping information (location of file data on OSTs). Examples of extended attributes are ACLs, striping information, and crypto keys.

Eviction The process of eliminating server state for a client that is not returning to the cluster after a timeout or if server failures have occurred.

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 Lock A 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 over the service.

Failout OST An OST which 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.

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 collection of integers which uniquely identify a file or object. The FID structure contains a sequence, identity and version number.

Fileset A group of files that are defined through a directory that represents a file system's start point.

FLDB FID Location Database. This database maps a sequence of FIDs to a server which is managing the objects in the sequence.

Flight Group Group or I/O transfer operations initiated in the OSC, which is simultaneously going between two endpoints. Tuning the flight group size correctly leads to a full pipe.

G

Glimpse callback An RPC made by an OST or MDT to another system, usually a client, to indicate to tthat an extent lock it is holding should be surrendered if it is not in use. If the system is using the lock, then the system should report the object size in the reply to the glimpse callback. Glimpses are introduced to optimize the acquisition of file sizes.

Group Lock

Group upcall

I

Import The state held by a client to fully recover a transaction sequence after a server failure and restart.

Intent Lock A special locking operation introduced by Lustre into 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.

IOV I/O vector. A buffer destined for transport across the network which contains a collection (a/k/a as a vector) of blocks with data.

K

Kerberos An authentication mechanism, optionally available in an upcoming Lustre version as a GSS backend.

L

LBUG A bug that Lustre writes into a log indicating a serious system failure.

LDLM Lustre Distributed Lock Manager.

lfs The Lustre File System configuration tool for end users to set/check file striping, etc. See Section 32.1, "lfs" on page 32-2.

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 through multiple disk failures and other means that cannot be recovered using file system journal recovery.

liblustre Lustre library. A user-mode Lustre client linked into a user program for Lustre fs access. liblustre clients cache no data, do not need to give back locks on time, and can recover safely from an eviction. They should not participate in recovery.

Llite Lustre lite. This term is in use inside the code and module names to indicate that code elements are related to the Lustre file system.

Llog Lustre log. A log of entries used internally by Lustre. 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 almost infinite size. llogs have an originator which writes records and a replicator which cancels record (usually through an RPC), when the records are not needed.

LMV Logical Metadata Volume. A driver to abstract in the Lustre client that it is working with a metadata cluster instead of a single metadata server.

LND Lustre Network Driver. A code module that enables LNET support over a particular transport, such as TCP and various kinds of InfiniBand.

LNET Lustre Networking. A message passing network protocol capable of running and routing through various physical layers. LNET forms the underpinning of LNETrpc.

Load-balancing MDSs A cluster of MDSs that perform load balancing of on system requests.

Lock Client A module that makes lock RPCs to a lock server and handles revocations from the server.

Lock Server A system 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, providing names for their OSTs.

Lustre The name of the project chosen by Peter Braam in 1999 for an object-based storage architecture. Now the name is commonly associated with the Lustre file system.

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 which contains references to a storage object on OSSs.

Lustre lite A preliminary version of Lustre developed for LLNL in 2002. With the release of Lustre 1.0 in late 2003, Lustre Lite became obsolete.

Lvfs A library that provides an interface between Lustre OSD and MDD drivers and file systems; this avoids introducing file system-specific abstractions into the OSD and MDD drivers.

M

Mballoc Multi-Block-Allocate. Lustre functionality that enables the ldiskfs file system to allocate multiple blocks with a single request to the block allocator. Normally, an ldiskfs file system only allocates only one block per request.

MDC MetaData Client - 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 - Server node that is hosting the Metadata Target (MDT).

MDT Metadata Target. A metadata device made available through the Lustre meta-data network protocol.

Metadata Write-back Cache A cache of metadata updates (mkdir, create, setattr, other operations) which an application has performed, but have not yet been flushed to a storage device or server.

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 (introduced in version 1.6) which formats disk file systems on servers with the mkfs.lustre program, and prepares them for automatic incorporation into a Lustre cluster.

N

NAL An older, obsolete term for LND.

NID Network Identifier. Encodes the type, network number and network address of a network interface on a node for use by Lustre.

NIO API A subset of the LNET RPC module that implements a library for sending large network requests, moving buffers with RDMA.

O

OBD Object Device. The base class of layering software constructs that provides Lustre functionality.

OBD API See Storage Object API.

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 OSD device driver.

Object device An instance of an object that exports the OBD API.

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 and T10 protocols is that Lustre 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 for which there is no Lustre file pointing at them. Orphan objects can arise from crashes and are automatically removed by an llog recovery. When a client deletes a file, the MDT gives back a cookie for each stripe. The client then sends the cookie and directs the OST to delete the stripe. Finally, the OST sends the cookie back to the MDT to cancel it.

Orphan handling A component of the metadata service which allows for recovery of open, unlinked files after a server crash. The implementation of this feature retains open, unlinked files as orphan objects until it is determined that no clients are using them.

OSC Object Storage Client. The client unit talking to an OST (via an OSS).

OSD Object Storage Device. A generic, industry term for storage devices with more extended interface than block-oriented devices, such as disks. Lustre uses this name to describe to a software module that implements an object storage API in the kernel. Lustre also uses this name 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 storage objects.

P

Pdirops A locking protocol introduced in the VFS by CFS to allow for concurrent operations on a single directory inode.

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 concept used by LNET. LNET messages are sent to a portal on a NID. Portals can receive packets when a memory descriptor is attached to the portal. Portals are implemented as integers.

Examples of portals are the portals on which certain groups of object, metadata, configuration and locking requests and replies are received.

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.

Reply The concept of re-executing a server request after the server 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 has received a reply are replayed.

Re-sent request A request that has seen no reply can be re-sent after a server reboot.

Revocation Callback An RPC made by an OST or MDT to another system, usually a client, to revoke a granted lock.

Rollback The concept that server state is in a crash lost because it was cached in memory and not yet persistent on disk.

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 gaining broad permissions on servers. 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

Storage Object API The API that manipulates storage objects. This API is richer than that of block devices and includes the create/delete of storage objects, read/write of buffers from and to certain offsets, set attributes and other storage object metadata.

Storage Objects A generic concept referring to data containers, similar/identical to file inodes.

Stride A contiguous, logical extent of a Lustre file written to a single OST.

Stride size The maximum size of a stride, typically 4 MB.

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. Lustre does not use T10.

W

Wide striping Strategy of using many OSTs to store stripes of a single file. This obtains maximum bandwidth to a single file through parallel utilization of many OSTs.

Index

M

N

O

P

Q

R

S

T

U

V

W