Getting Started With XFS™ Filesystems

Document Number 007-2549-001

CONTRIBUTORS

Written by Susan Ellis and John Raithel
Illustrated by Gloria Ackley
Production by Gloria Ackley
Engineering contributions by Doug Doucette, Wei Hu, Tom Phelan, and Chuck Bullis
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson,
Erik Lindholm, and Kay Maitz

Getting Started With XFS™ Filesystems
Document Number 007-2549-001

Contents

List of Examples

List of Figures

List of Tables

About This Guide

Getting Started With XFS Filesystems describes the XFS filesystem and XLV Volume Manager. Developed at Silicon Graphics®, these IRIX™ features provide high-performance alternatives to the Extent File System™ (EFS) and logical volume managers previously available with IRIX. This guide was prepared in conjunction with the initial release of XFS, called IRIX 5.3 with XFS.

The features described in this guide are included in IRIX system software releases beginning with the IRIX 5.3 with XFS release. However, to use several features, you must obtain NetLS™ licenses by purchasing separate software options. The features that require NetLS licenses are:

- The plexing feature of the XLV Volume Manager, which provides mirroring of disks up to four copies. This feature is provided by the Disk Plexing Option software option.
- Guaranteed-rate I/O (GRIO), a feature that enables an application to request a fixed I/O rate and, if granted, be assured of receiving that rate. By default, the system allows four GRIO streams. To obtain up to 40 streams, you must purchase the High Performance Guaranteed-Rate I/O—5-40 Streams software option. An unlimited number of streams is provided by the High Performance Guaranteed-Rate I/O—Unlimited Streams software option.

This guide covers only system administration of XFS filesystems and XLV logical volumes (including volumes used for GRIO). See the section "For More Information" later in this chapter for information about the programmatic interface to XFS, which is provided with the IRIS® Development Option (IDO) software option.

Audience

This guide is written for system administrators and other knowledgeable IRIX users who want to use XFS filesystems and/or XLV logical volumes. Because many of the procedures in this guide can result in loss of files on the system if the procedures are not performed correctly, this guide and its procedures should be used only by people who are

- familiar with UNIX® filesystem administration procedures
- experienced in disk repartitioning using *fx*(1M)
- comfortable performing administration tasks from the shell in the miniroot environment provided by *inst*(1M)
- familiar with filesystem backup concepts and procedures, particularly using *dump*(1M)

How to Use This Guide

This guide provides five chapters of basic information about the design and system administration of the XFS filesystem and XLV volume manager:

- Chapter 1, "Introduction to XFS, XLV, and GRIO," provides an overview of the features of the XFS filesystem, XLV volume manager, and guaranteed-rate I/O system.
- Chapter 2, "XFS Filesystem Administration," describes filesystem administration tasks such as creating XFS filesystems on new disks and converting filesystems from EFS to XFS.
- Chapter 3, "Dumping and Restoring XFS Filesystems," explains how to perform filesystem backups with *xfsdump*(1M) and how to restore filesystems and files using *xfsrestore*(1M).
- Chapter 4, "XLV Logical Volumes," describes the structure and features of XLV logical volumes and explains how to create and manage logical volumes.
- Chapter 5, "Guaranteed-Rate I/O,"explains how to configure and create real-time XFS filesystems on XLV volumes so that applications can use the guaranteed-rate I/O (GRIO) feature of XFS to ensure high-performance I/O.

Two appendixes provide reference information for XFS and XLV:

- Appendix A, "Error Messages," lists error messages that can occur during the creation and administration of XFS filesystems and XLV logical volumes, their possible causes, and advice on how to proceed.
- Appendix B, "Reference Pages," contains the key reference pages for XFS and XLV administration and lists other reference pages that contain related XFS, XLV, and disk management information.

Hardware Requirements

At least 32 MB of memory is recommended for systems with XFS filesystems.

XFS filesystems and XLV logical volumes are not supported on systems with IP4 or IP6 CPUs.

Using XLV logical volumes is not recommended on systems with a single disk.

Some uses of guaranteed-rate I/O, described in Chapter 5, "Guaranteed-Rate I/O," have special disk configuration requirements. These requirements are explained in the section "Hardware Configuration Requirements for GRIO" in Chapter 5.

Conventions

This guide uses these font conventions:

italics Italics are used for command names, reference page names, file names, variables, and the names of *inst*(1M) subsystems.

`fixed-width type`
Fixed-width type is used for examples of command output that is displayed in windows on your monitor.

`bold fixed-width type`
Bold fixed-width type is used for commands and text that you are to type literally.

`<Enter>`	When you see **`<Enter>`**, press the Enter key on the keyboard; do not type in the letters.

Product Support

Silicon Graphics offers a comprehensive product support and maintenance program for its products. For information about using support services for this product, refer to the *Release Notes* that accompany it.

For More Information

For more information about disk management on IRIX, see these sources:

- The *IRIX Advanced Site and Server Administration Guide*, provides detailed information on system administration of Silicon Graphics systems. Although it has not yet been updated to include information on XFS and XLV, it provides background information and procedures on disk management, logical volumes, filesystem administration, and system backups that remain applicable for systems using XFS and XLV.

  The *IRIX Advanced Site and Server Administration Guide* is available for online viewing with the IRIS InSight™ viewer, *insight*(1). It is also available in printed form.
- Online reference pages (man pages) on various disk information and management utilities are included in the standard system software and can be viewed online using the *man*(1) and *xman*(1) commands or the “Man Pages” item on the Help menu of the System Toolchest. Appendix B provides a complete list of these reference pages.
- The guide *Selected IRIX Site Administration Reference Pages* provides printed reference pages for many of the utilities used in the procedures in this guide.

For more information on developing applications that access XFS filesystems, see these sources:

- Online reference pages for system calls and library routines relevant to XFS and GRIO are provided in the IRIS Developer's Option (IDO) software product. Appendix B provides a complete list of these reference pages.
- The *REACT/Pro™ Programmer's Guide* provides information about developing applications that use GRIO.

For instructions for loading the miniroot, see the *Software Installation Administrator's Guide.*

For information on acquiring and installing NetLS licenses that enable the High Performance Guaranteed-Rate I/O software options, see the *Network License System™ Administration Guide.*

For addition information on the software releases that include the new features documented in this guide, see the *Release Notes* for these products:

- *IRIX*
- *eoe*
- *xfs*
- *plexing*
- *grio*
- *nfs*
- *dev*

Chapter 1

Introduction to XFS, XLV, and GRIO

This guide provides the information you need to get started using the new Silicon Graphics filesystem technology:

- XFS is the next-generation Silicon Graphics filesystem. Systems can use XFS filesystems exclusively or have a mixture of XFS and EFS filesystems.
- The XLV volume manager provides an alternative to Silicon Graphics' existing *lv* volume manager and to the IRIS Volume Manager software option. Both XFS and EFS filesystems can be built on XLV logical volumes.
- The guaranteed-rate I/O system (GRIO) allows applications to reserve specific I/O performance to and from the filesystem. It requires the use of an XLV logical volume and XFS filesystem.

This chapter highlights the major features of XFS, XLV, and GRIO.

XFS Features

XFS is designed for use on most Silicon Graphics systems—from desktop systems to supercomputer systems. Its major features include

- full 64-bit file capabilities (files larger than 2 GB)
- rapid and reliable recovery after system crashes because of the use of journaling technology
- efficient support of large, sparse files (files with "holes")
- integrated, full-function volume manager, the XLV Volume Manager
- extremely high I/O performance that scales well on multiprocessing systems

- guaranteed-rate I/O for multimedia and data acquisition uses
- compatible with existing applications and with NFS®
- user-specified block sizes ranging from 512 bytes up to 64 KB

Currently, XFS supports files and filesystems that grow to 2^{40}-1 or 1,099,511,627,775 bytes (one terabyte). Support for filesystems up to 2^{63}-1 bytes is planned for a future release. You can use the filesystem interfaces supplied with the IRIS Development Option (IDO) software option to write 32-bit programs that can track 64-bit position and file size. Many programs work without modification because sequential reads succeed even on files larger than 2 GB. NFS Release 5.3 and later allows you to export 64-bit XFS filesystems to other systems.

XFS uses database journaling technology to provide high reliability and rapid recovery. Recovery after a system crash is completed within a few seconds, without the use of a filesystem checker such as *fsck*(1M). Recovery time is independent of filesystem size.

XFS is designed to be a very high performance filesystem. Under certain conditions, throughput is expected to exceed 100 MB per second. Its performance scales to complement the CHALLENGE™ MP architecture. While traditional files, directories, and filesystems suffer from reduced performance as they grow in size, with XFS there is no performance penalty.

You can create filesystems with block sizes ranging from 512 bytes to 64 KB. For real-time data, the maximum *extent* size is 1 GB. Filesystem extents, which provide contiguous data within a file, are configurable at file creation time using *fcntl*(2) and are multiples of the filesystem block size.

Most filesystem utilities, such as *du*(1), *dvhtool*(1M), *ls*(1), *mount*(1M), *prtvtoc*(1M), and *umount*(1M), work with XFS filesystems as well as EFS filesystems with no user-visible changes. A few utilities, such as *df*(1), *fx*(1M) and *mkfs*(1M) have additional features for XFS. The filesystem utilities *clri*(1M), *fsck*(1M), *findblk*(1M), and *ncheck*(1M) are not used with XFS filesystems.

For backup and restore, the standard IRIX utilities *Backup*(1), *bru*(1), *cpio*(1), *Restore*(1), and *tar*(1) and the optional software product NetWorker® for IRIX can be used for files less than 2 GB in size. To dump XFS filesystems, the new utility *xfsdump*(1M) must be used instead of *dump*(1M). Restoring from these dumps is done using *xfsrestore*(1M). See Table 3-1 and Table 3-2 in Chapter 3, “Dumping and Restoring XFS Filesystems,” for more information about the relationships between *xfsdump*, *xfsrestore*, *dump*, and *restore* on XFS and EFS filesystems.

XLV Features

The new XLV Volume Manager provides these advantages when XLV logical volumes are used as raw devices, when XFS filesystems are created on them, and when EFS filesystems are created on them:

- support for very large logical volumes—up to one terabyte.
- support for disk striping for higher I/O performance
- plexing (mirroring) for higher system and data reliability
- online volume reconfigurations, such as increasing the size of a volume, for less system downtime

With XFS filesystems, XLV provides these additional advantages:

- filesystem journal records on a separate partition, which can be on a separate disk, for maximum performance
- access to real-time data

When XFS filesystems are used on XLV volumes, each logical volume can contain up to three subvolumes: data (required), log, and real-time. The data subvolume normally contains user files and filesystem *metadata* (inodes, indirect blocks, directories, and free space blocks). The log subvolume is used for filesystem journal records. If there is no log subvolume, journal records are placed in the data subvolume. Data with special I/O bandwidth requirements, such as video, can be placed on the real-time subvolume.

XLV increases system reliability and availability by enabling you to add or remove a plex, increase the size of (grow) a volume, and replace failed elements of a plexed volume without taking the volume out of service.

Converting from *lv* logical volumes to XLV logical volumes is easy. Using the programs *lv_to_xlv*(1M) and *xlv_make*(1M), you can convert *lv* logical volumes to XLV without having to dump and restore your data. Converting from IRIS Volume Manager volumes to XLV is beyond the scope of this guide.

Note: The plexing feature of XLV is available only when you purchase the Disk Plexing Option software option. See the *plexing Release Notes* for information on purchasing this software option and obtaining the required NetLS license.

GRIO Features

The guaranteed-rate I/O system (GRIO) allows applications to reserve specific I/O bandwidth to and from the filesystem. Applications request guarantees by providing a file descriptor, data rate, duration, and start time. The filesystem calculates the performance available and, if the request is granted, guarantees that the requested level of performance can be met for a given time. This frees programmers from having to predict the performance and is critical for media delivery systems such as video-on-demand.

Guarantees can be *hard* or *soft*, a way of expressing the trade-off between reliability and performance. Hard guarantees deliver the requested performance, but with some possibility of error in the data (due to the requirements for turning off disk drive self-diagnostics and error-correction firmware). Soft guarantees allow the disk drive to retry operations in the event of an error, but this can possibly result in missing the rate guarantee. Hard guarantees place greater restrictions on the system hardware configuration.

Note: By default, IRIX supports four GRIO streams (concurrent uses of GRIO). To increase the number of streams to 40, you can purchase the High Performance Guaranteed-Rate I/O—5-40 Streams software option. For more streams, you can purchase the High Performance Guaranteed-Rate I/O—Unlimited Streams software option. See the *grio Release Notes* for information on purchasing these software options and obtaining the required NetLS licenses.

Chapter 2

XFS Filesystem Administration

This chapter explains the procedures for creating XFS filesystems, converting EFS filesystems to XFS, and performing filesystem administration tasks that require programs specific to XFS.

The main sections in this chapter are:

- "Planning for XFS Filesystems" on page 5
- "Making an XFS Filesystem on a Disk Partition" on page 14
- "Making an XFS Filesystem on an XLV Logical Volume" on page 16
- "Converting Filesystems on the System Disk From EFS to XFS" on page 18
- "Converting a Filesystem on an Option Disk from EFS to XFS" on page 25
- "Checking Filesystem Consistency" on page 27

Planning for XFS Filesystems

The following subsections discuss choices you must make and preparation for creating an XFS filesystem. Each time you plan to make an XFS filesystem or convert a filesystem from EFS to XFS, you should review each section and make any necessary preparations.

Don't Use XFS When ...

Do not use an XFS filesystem if any of the following is true:

- There is insufficient free disk space (see the section "Checking for Adequate Free Disk Space" in this chapter).

- The system doesn't meet the hardware configuration requirements listed in the section "Hardware Requirements" in "About This Guide."
- The filesystem is the root filesystem and you intend to increase its size later. In this case, delay the conversion until you are ready to increase the size of the filesystem.
- The filesystems are the root and, if present, usr filesystems and you want to continue using the System Recovery procedure (item 4, Recover System, on the System Maintenance Menu). System Recovery doesn't work with XFS filesystems because of the limitations of *bru*(1M), which is used by System Recovery. However, *xfsdump*(1M) can be used to create backups that can be used to recover the system, if necessary.

Prerequisite Software

Using XFS filesystems and XLV logical volumes requires at least IRIX 5.3 with XFS or a later system software release. The procedures in this chapter assume that the proper software has been installed and the system rebooted prior to beginning the procedure.

Some important subsystems in the IRIX 5.3 with XFS and later releases are:

eoe1.sw.unix	The minimum release level is IRIX 5.3 with XFS.
eoe2.sw.efs	This subsystem is required.
eoe2.sw.lv	This subsystem needs to be installed only if *lv* logical volumes are in use on the system.
eoe2.sw.xfs	This subsystem is required.
eoe2.sw.xlv	Install this subsystem if you intend to use the XLV volume manager.
eoe2.sw.xlvplex	Install this subsystem if you have purchased the Disk Plexing Option software option.

In addition to the subsystems listed above, software patches may be required to use all of the features documented in this guide. See the *xfs*, *IRIX*, *eoe*, *nfs*, and *dev Release Notes* for more information.

If you are converting the root and usr filesystems, you must have software distribution CDs or access to a remote distribution directory for IRIX Release 5.3 with XFS or a later system software release. Instructions on loading the miniroot from these CDs is provided in Chapter 3 of the *Software Installation Administrator's Guide*.

Choosing Block Sizes

XFS allows you to choose two types of block sizes for each filesystem. (EFS has a fixed block size of 512 bytes.) One is the filesystem block size, used for user files, and the other is the extent size, used for the real-time subvolume on an XLV logical volume, if present. The extent size is the amount of space that will be allocated to the file every time more space needs to be allocated to it.

For XFS filesystems on disk partitions and for the data subvolume of filesystems on XLV volumes, the block size guidelines for user files are:

- The minimum block size is 512 bytes.
- The maximum block size is 65536 bytes (64K). However, in general block sizes shouldn't be larger than 4096 bytes.
- The default block size is 4096 bytes (4K).
- For root filesystems on systems with separate root and usr filesystems, the recommended block size is 512 bytes. (Root filesystems in this configuration usually don't have much extra disk space and large block sizes compound the problem.)
- For news servers, the recommended block size is 2048 bytes.
- In general, the recommended block size for filesystems under 100 MB is 512 bytes. For larger filesystems 4096 bytes is recommended.

Block sizes are specified in bytes in decimal (default), octal (prefixed by 0), or hexadecimal (prefixed by 0x or 0X). If the number has the suffix "k", it is multiplied by 1024. If the number has the suffix "m", it is multiplied by 1048576 (1024 * 1024).

For real-time subvolumes of XLV logical volumes, the block size is the same as the block size of the data subvolume. The guidelines for the extent size are:

- The extent size must be a multiple of the block size of the data subvolume.
- The minimum extent size is 64 KB.
- The maximum extent size is 1 GB.
- The default extent size is 64 KB.
- The extent size should be matched to the application and the stripe unit of the volume elements used in the real-time subvolume.

Choosing the Log Type and Size

Each XFS filesystem has a log that contains filesystem journaling records. This log requires dedicated disk space. This disk space doesn't show up in *df*(1) listings, nor can you access it with a filename.

The location of the disk space depends on the type of log you choose. The two types of logs are:

external	When an XFS filesystem is created on an XLV logical volume and log records are put into a log subvolume, the log is called an *external* log. The log subvolume is one or more disk partitions dedicated to the log exclusively.
internal	When an XFS filesystem is created on a disk partition, or when it is created on an XLV logical volume that doesn't have a log subvolume, log records are put into a dedicated portion of the disk partition (or data subvolume) that contains user files. This type of log is called an *internal* log.

The guidelines for choosing the log type are:

- If you want the log and the data subvolume to be on different partitions or to use different subvolume configurations for them, use an external log.
- If you are making the XFS filesystem on a disk partition (rather than on an XLV logical volume), you must use an internal log.

- If you are making the XFS filesystem on an XLV logical volume that has no log subvolume, you must use an internal log.
- If you are making the XFS filesystem on an XLV logical volume that has a log subvolume, you must use an external log.

For more information about XLV and log subvolumes, see Chapter 4, "XLV Logical Volumes."

The amount of disk space needed for the log is a function of how the filesystem is used. The amount of disk space required for log records is proportional to the transaction rate and the size of transactions on the filesystem, not the size of the filesystem. Larger block sizes result in larger transactions. Transactions from directory updates (for example, *mkdir*(1), *rmdir*(1), *create*(2), and *unlink*(2)) cause more log data to be generated. You must choose the amount of disk space to dedicate to the log (called the log size).

The minimum log size is 512 blocks. Some guidelines for log sizes are shown in Table 2-1.

Table 2-1 Log Size Guidelines

Log Size	**Blocks**	**Transaction Activity**
Small	512 blocks	Low update activity or small filesystem (less than 100 MB)
Medium	2000 blocks	Average
Large	4000 blocks	Very high

For external logs, the size of the log is the same as the size of the log subvolume. The log subvolume is one or more disk partitions. You may find that you need to re-partition a disk to create a properly sized log subvolume (see the section "Disk Partitioning" in this chapter). For external logs, the size of the log is set when you create the log subvolume with *xlv_make*(1M).

For internal logs, the size of the log is specified when you create the filesystem with *mkfs*(1M).

The log size is specified in bytes or as a multiple of the filesystem block size. Decimal numbers are the default, but they can be specified in octal (prefixed by 0) or hexadecimal (prefixed by 0x or 0X). Numbers with no suffixes are bytes. If the number has the suffix "k", it is multiplied by 1024 bytes. If the number has the suffix "m", it is multiplied by 1048576 (1024 * 1024) bytes or one megabyte. If the number has the suffix "b", it is multiplied by the filesystem block size.

Checking for Adequate Free Disk Space

XFS filesystems may require more disk space than EFS filesystems for the same files. This extra disk space is required to accommodate the XFS log and as a result of block sizes larger than EFS's 512 bytes. However, XFS represents free space more compactly, on average, and the inodes are allocated dynamically by XFS, which can result in less disk space usage.

This procedure can be used to get a rough idea of the amount of free disk space that will remain after a filesystem is converted to XFS:

1. Get the size in kilobytes of the filesystem to be converted and round the result to the next megabyte. For example:

```
df -k
Filesystem                Type  kbytes     use   avail %use  Mounted on
/dev/root                  efs  969857  648451  321406  67%  /
```

   This filesystem is 969857 KB, which rounds up to 970 MB.

2. If you plan to use an internal log (see the section "Choosing the Log Type and Size" in this chapter), give this command to get an estimate of the disk space required for the files in the filesystem after conversion:

   `xfs_estimate -i` *logsize* **`-b`** *blocksize mountpoint*

   logsize is the size of the log. *blocksize* is the block size you chose for user files in the section "Choosing Block Sizes" in this chapter. *mountpoint* is the directory that is the mount point for the filesystem.

   The output of this command tells you how much disk space the files in the filesystem and an internal log of size *logsize* will take after conversion to XFS.

3. If you plan to use an external log, give this command to get an estimate of the disk space required for the files in the filesystem after conversion:

   `xfs_estimate -e 0 -b` *blocksize mountpoint*

   blocksize is the block size you chose for user files in the section "Choosing Block Sizes" in this chapter. *mountpoint* is the directory that is the mount point for the filesystem.

   The first line of output from *xfs_estimate* tells you how much disk space the files in the filesystem will take after conversion to XFS. In addition to this, you will need disk space on a different disk partition for the external log. You should ignore the second line of output.

4. Compare the size of the filesystem from step 1 with the size of the files from step 2 or step 3. For example,

   ```
   970 MB - 739 MB = 231 MB free disk space
   739 MB / 970 MB = 76.2% full
   ```

   Use this information to decide if there will be an adequate amount of free disk space if this filesystem is converted to XFS.

If the amount of free disk space after conversion is not adequate, some options to consider are:

- Implement the usual solutions for inadequate disk space: remove unnecessary files, archive files to tape, move files to another filesystem, add another disk, and so on.
- Repartition the disk to increase size of the disk partition for the filesystem.
- If there isn't sufficient disk space in the root filesystem and you have separate root and usr filesystems, switch to combined root and usr filesystems on a single disk partition.
- If the filesystem is on an *lv* logical volume or an XLV logical volume, increase the size of the volume.
- Create an XLV logical volume with a log subvolume elsewhere, so that all of the disk space can be used for user files.

Disk Partitioning

Many system administrators may find that they want or need to repartition disks when they switch to XFS filesystems and / or XLV logical volumes. The next two subsections explain why you might want to repartition and give some tips on partition sizes and types.

Why Should Disks Be Repartitioned?

Some of the reasons to consider repartitioning are:

- If the system disk has separate partitions for root and usr, the root partition may be running out of space. Repartitioning is a way to increase the space in root (at the expense of the size of usr) or to solve the problem by combining root and usr into a single partition.
- System administration is a little easier on systems with combined root and usr filesystems.
- If you plan to use XLV logical volumes, you may want to put the XFS log into a small subvolume. This requires disk repartitioning to create a small partition for the log subvolume.
- If you plan to use XLV logical volumes, you may want to repartition to create disk partitions of equal size that can be striped or plexed.

How Should Disks Be Repartitioned?

Explaining the details of using *fx* to repartition a disk is beyond the scope of this guide (see the *fx*(1M) reference page for details). However, the list below provides useful information about new features of *fx* and about partitioning details that are specific to XFS and XLV.

- If you are repartitioning the system disk, you must use the standalone version of *fx*. Otherwise, you can use the IRIX version of *fx*. Using the expert mode of *fx* (the **–x** option) shouldn't be necessary.
- If you repartition a system disk, remember that the swap space should never be less than 40 MB. A smaller swap space impacts system performance and makes it impossible to install software using the miniroot.

- New partition types have been added to *fx*. Table 2-2 lists and describes all partition types.

Table 2-2 Disk Partition

Type	Description
efs	Used for an EFS filesystem
lvol	Part of an *lv* logical volume
raw	Used for data
xfs	Used for an XFS filesystem
xfslog	Used for an XFS filesystem log
xlv	Part of an XLV logical volume
volhdr	Volume header
volume	Entire volume

- The repartition / usrroot and repartition / option menu items have been changed. If you select these menu items, you are asked what type of data partition you want. If you enter **`xfs`**, you are asked if you want a log partition. If you answer **`yes`**, *fx* makes partition 15 into a 4 MB xfslog partition, which can be used for an XLV log subvolume.

Dump and Restore Requirements

The filesystem conversion procedures in the sections "Converting Filesystems on the System Disk From EFS to XFS" and "Converting a Filesystem on an Option Disk from EFS to XFS" in this chapter require that you dump the filesystems you plan to convert to tape or to another disk with sufficient free disk space to contain the dump image.

When you convert a system disk, you must use *dump*(1M) and *restore*(1M). When you convert a filesystem on an option disk, you can any backup and restore programs. The reference pages for *dump* and *restore* are included in Appendix B, "Reference Pages."

If you dump to a tape drive, follow these guidelines:

- Have sufficient tapes available for dumping the filesystems to be converted.
- If you are converting filesystems on a system disk, the tape drive must be local.
- If you are converting filesystems on option disks, the tape drive can be local or remote.

The requirements for dumping to a different filesystem are:

- The filesystem being converted must have 2 GB or less in use (the maximum size of the dump image file on an EFS filesystem).
- The filesystem that will contain the dump must have sufficient disk space available to hold the filesystems to be converted.
- If you are converting filesystems on a system disk, the filesystem where you place the dump must be local to the system.
- If you are converting filesystems on option disks, the filesystem you dump to can be local or remote.

Dumping to disk takes about 10 minutes per gigabyte. Dumping to tape takes about 12 minutes per 100 megabytes.

Making an XFS Filesystem on a Disk Partition

This section explains how to create an XFS filesystem on an empty disk partition. This procedure applies to two cases:

- The disk partition is not part of an XLV logical volume.
- The disk partition is part of an XLV logical volume that doesn't have a log subvolume (the log is internal).

See the section "Making an XFS Filesystem on an XLV Logical Volume" in this chapter for instructions on creating an XFS filesystem on an XLV logical volume that has a log subvolume (an external log).

You must be superuser to perform this procedure.

1. Review the subsections within the section “Planning for XFS Filesystems” in this chapter to verify that you are ready to begin this procedure.
2. Identify the device name of the partition, *partition*, where you plan to create the filesystem. For example, if you plan to use partition 7 (the entire disk) of a SCSI option disk on controller 0, unit 2, *partition* is /dev/dsk/dks0d2s7. For more information on determining *partition* (also known as a *special* file), see *dks*(7M) for SCSI disks and *ipi*(7M) for Xylogics IPI disks.
3. If the disk partition is already mounted, unmount it:

   `umount` *partition*

   Any data that is on the disk partition will be destroyed (to convert the data rather than destroy it, use the procedure in the section “Converting a Filesystem on an Option Disk from EFS to XFS” in this chapter instead).
4. Use the *mkfs*(1M) command to create the new XFS filesystem:

   `mkfs -d name=`*partition* **`-b size=`***blocksize* **`-l internal,size=`***logsize*

   blocksize is the filesystem block size (see the section “Choosing Block Sizes” in this chapter) and *logsize* is the size of the area dedicated to log records (see the section “Choosing the Log Type and Size” in this chapter).

   Example 2-1 shows the command line used to create an XFS filesystem and the system output. The filesystem has a 10 MB internal log and a block size of 1K bytes and is on the partition */dev/dsk/dks0d2s7*.

Example 2-1 *mkfs* Command for an XFS Filesystem With an Internal Log

```
mkfs -d name=/dev/dsk/dks0d2s7 -b size=1k -l internal,size=10m
meta-data=/dev/dsk/dks0d2s7      isize=256    agcount=8, agsize=128615 blks
data     =                       bsize=1024   blocks=1028916
log      =internal log           bsize=1024   blocks=10240
realtime =none                   bsize=65536  blocks=0, rtextents=0
```

5. If it doesn’t already exist, create a mount point directory, *mountdir*, for the filesystem:

   `mkdir` *mountdir*

6. To mount the filesystem immediately, give this command:

   `mount` *partition mountdir*

7. To configure the system so that the new filesystem is automatically mounted when the system is booted, add this line to the file */etc/fstab*:

   partition mountdir `xfs rw,raw=`*rawpartition* `0 0`

   where *rawpartition* is the raw version of *partition*. For example, if *partition* is /dev/dsk/dks0d2s7, *rawpartition* is /dev/rdsk/dks0d2s7.

Making an XFS Filesystem on an XLV Logical Volume

This section describes how to make an XFS filesystem on an empty XLV volume that has an external log subvolume and a data subvolume. (Creating XLV volumes is explained in the section "Using xlv_make to Create Volume Objects" in Chapter 4.)

1. Review the subsections within the section "Planning for XFS Filesystems" in this chapter to verify that you are ready to begin this procedure.

2. Use the *mkfs*(1M) command to make the new XFS filesystem:

   `mkfs -b size=`*blocksize volume*

   blocksize is the block size for filesystem (see "Choosing Block Sizes" in this chapter), and *volume* is the device name for the volume.

   Example 2-2 shows the command line used to create an XFS filesystem on a logical volume */dev/dsk/xlv/a* and a block size of 1K bytes and the system output.

Example 2-2 *mkfs* Command for an XFS Filesystem With an External Log

```
mkfs -b size=1k /dev/dsk/xlv/a
meta-data=/dev/dsk/xlv/a        isize=256    agcount=8, agsize=245530 blks
data     =                      bsize=1024   blocks=1964240
log      =volume log            bsize=1024   blocks=25326
realtime =none                  bsize=65536  blocks=0, rtextents=0
```

Example 2-3 shows the command line used to create an XFS filesystem on a logical volume */dev/dsk/xlv/xlv_data1* and the system output. The default block size of 4096 bytes is used and the real-time extent size is set to 128K bytes.

Example 2-3 *mkfs* Command for an XFS Filesystem With a Real-Time Subvolume

```
mkfs_xfs -r extsize=128k /dev/rdsk/xlv/xlv_data1
meta-data=/dev/rdsk/xlv/xlv_data1 isize=256    agcount=8, agsize=4300 blks
data     =                        bsize=4096   blocks=34400
log      =volume log              bsize=4096   blocks=34400
realtime =volume rt               bsize=131072 blocks=2560, rtextents=80
```

3. If it doesn't already exist, create a mount point directory, *mountdir*, for the filesystem:

   mkdir *mountdir*

4. To mount the filesystem immediately, give this command:

   mount *volume mountdir*

5. To configure the system so the new filesystem is automatically mounted when the system is booted, add this line to the file */etc/fstab*:

   volume mountdir `xfs rw,raw=`*rawvolume* `0 0`

   where *rawvolume* is the raw version of *volume*. For example, if *volume* is /dev/dsk/xlv/a, *rawvolume* is /dev/rdsk/xlv/a and the */etc/fstab* entry is:

   ```
   /dev/dsk/xlv/a /a xfs rw,raw=/dev/rdsk/xlv/a 0 0
   ```

Converting Filesystems on the System Disk From EFS to XFS

This section explains the procedure for converting filesystems on the system disk from EFS to XFS. Some systems have two filesystems on the system disk, the root filesystem (mounted at */*) and the usr filesystem (mounted at */usr*). Other systems have a single, combined root and usr filesystem mounted at */*. This procedure covers both cases but assumes that neither *lv* nor XLV logical volumes are in use on the system disk. The basic procedure for converting a system disk is:

- Load the IRIX 5.3 with XFS miniroot.
- Do a complete dump of filesystems on the system disk.
- Repartition the system disk if necessary.
- Create one or two new, empty XFS filesystems.
- Restore the files from the filesystem dumps.
- Reboot the system.

During this procedure, you can repartition the system disk if needed. For example, you can convert from separate root and usr filesystems to a single, combined filesystem, or you can resize partitions to make the root partition larger and the usr partition smaller. See the section “Disk Partitioning” in this chapter for more information.

The early steps of this procedure ask you to identify the values of various variables, which are used later in the procedure. You may find it helpful to make a list of the variables and values for later reference. Be sure to perform only the steps that apply to your situation. Perform all steps as superuser.

Note: It is very important to follow this procedure as documented without giving additional *inst* or shell commands. Unfortunately, deviations from this procedure, even changing to a different directory or going from the *inst* shell to an *inst* menu when not directed to, can have very severe consequences from which recovery is difficult.

1. Review the subsections within the section “Planning for XFS Filesystems” in this chapter to verify that you are ready to begin this procedure. In particular, be sure that the software listed in the section “Prerequisite Software” has been installed and the system has been rebooted.

2. Verify that your backups are up to date. Because this procedure temporarily removes all files from your system disk, it is important that you have a complete set of backups that have been prepared using your normal backup procedures. You will make a complete dump of the system disk in step 11, but you should have your usual backups in addition to the backup made during this procedure.

3. Use *prtvtoc*(1M) to get the device name of the root disk partition, *rootpartition*. For example:

   ```
   # prtvtoc
   Printing label for root disk

   * /dev/rdsk/dks0d1s0 (bootfile "/unix")
   ...
   ```

   The `bootfile` line contains the raw device name of the root disk partition, which is /dev/rdsk/dks0d1s0 in this example. *rootpartition* is the non-raw device name, which is /dev/dsk/dks0d1s0 in this example.

4. If the system disk has separate root and usr filesystems, use the output of *prtvtoc* in the previous step to figure out the device name of the usr partition, *usrpartition*. Look for the line that shows a mount directory of */usr*:

```
Partition  Type  Fs   Start: sec   (cyl)    Size: sec    (cyl)  Mount Directory
...
6          efs   yes     116725  ( 203)       727950   (1266)   /usr
```

   The usr partition number is shown in the first column of this line; it is 6 in this example. To determine the value of *usrpartition*, replace the final digit in *rootpartition* with the usr partition number. For this example, *usrpartition* is /dev/dsk/dks0d1s6.

5. If you are using a tape drive as the backup device, use *hinv*(1M) to get the controller and unit numbers (<*tapecntlr*> and <*tapeunit*>) of the tape drive. For example:

   ```
   # hinv -c tape
   Tape drive: unit 2 on SCSI controller 0: DAT
   ```

   In this example, <*tapecntlr*> is 0 and <*tapeunit*> is 2.

6. If you are using a disk drive as your backup device, use *df*(1) to get the device name, *backupdevice*, and mount point, *backupfs*, of the partition that contains the filesystem where you plan to put the backup. For example:

```
# df
Filesystem                 Type  blocks     use   avail %use  Mounted on
/dev/root                   efs 1992630  538378 1454252  27%  /
/dev/dsk/dks0d3s7           efs 3826812 1559740 2267072  41%  /d3
/dev/dsk/dks0d2s7           efs 2004550      23 2004527   0%  /d2
```

   The filesystem mounted at */d2* has plenty of disk space for a backup of the system disk (/ uses 538,378 blocks and */d2* has 2,004,527 blocks available). The *backupdevice* for */d2* is /dev/dsk/dks0d2s7 and the *backupfs* is /d2.

7. Create a temporary copy of */etc/fstab* called */etc/fstab.xfs* and edit it with your favorite editor. For example:

```
# cp /etc/fstab /etc/fstab.xfs
# vi /etc/fstab.xfs
```

   Make these changes in */etc/fstab.xfs*:

   - Replace `efs` with `xfs` in the line for the root filesystem, /, if there is a line for root.
   - If root and usr are separate filesystems and will remain so, replace `efs` with `xfs` in the line for the usr filesystem.
   - If root and usr have been separate filesystems, but the disk will be repartitioned during the conversion procedure so that they are combined, remove the line for the usr filesystem.

8. Shut down your workstation using *shutdown*(1M) or the “System Shutdown” item on the System toolchest. Answer prompts as appropriate to get to the five-item System Maintenance menu.

9. Bring up the miniroot from system software CDs or a software distribution directory that contains the release IRIX 5.3 with XFS or a later release of IRIX. See the section “Prerequisite Software” in this chapter for more information.

10. Switch to the shell prompt in *inst*:

```
Inst> sh
```

11. Create a full backup of the root filesystem by giving this command:

    `# **dump 0uCf** *tapesize dumpdevice rootpartition*`

    tapesize is the tape capacity (it's used for backup to disks, too) and *dumpdevice* is the appropriate device name for the tape drive or the name of the file that will contain the dump image. Table 2-3 gives the values of *tapesize* and *dumpdevice* for different tape drives and disk. The *dump*(1M) reference page is included in Appendix B.

Table 2-3 *dump* Arguments for Filesystem Backup

Backup Device	*tapesize*	*dumpdevice*
Disk	2m	use /root/*backupfs*/root.dump for the root filesystem and /root/*backupfs*/usr.dump for the usr filesystem
DAT tape	2m	/dev/rmt/tps<*tapecntlr*>d<*tapeunit*>nsv
DLT tape	10m	/dev/rmt/tps<*tapecntlr*>d<*tapeunit*>nsv
EXABYTE™ 8mm model 8200 tape	2m	/dev/rmt/tps<*tapecntlr*>d<*tapeunit*>nsv
EXABYTE 8mm model 8500 tape	4m	/dev/rmt/tps<*tapecntlr*>d<*tapeunit*>nsv
QIC cartridge tape	150k	/dev/rmt/tps<*tapecntlr*>d<*tapeunit*>ns

12. If usr is a separate filesystem, insert a new tape (if you are using tape) and create a full backup of the usr filesystem by giving this command:

    `# **dump** *tapesize dumpdevice usrpartition*`

    Use the same values of *tapesize* and *dumpdevice* as in step 11.

13. If you do not need to repartition the system disk, skip to step 18.

14. To repartition the system disk, use the standalone version of *fx*(1M). This version of *fx* is invoked from the Command Monitor, so you must bring up the Command Monitor. To do this, quit out of *inst*, reboot the system, shut down the system, then request the Command Monitor. An example of this procedure is:

```
# exit
...
Inst> quit
...
Ready to restart the system.  Restart? { (y)es, (n)o, (sh)ell, (h)elp }: yes
...
login: root
# halt
...
System Maintenance Menu
...
Option? 5
Command Monitor.  Type "exit" to return to the menu.
>>
```

On systems with a graphical System Maintenance menu, choose the last option, `Enter Command Monitor`, instead of choosing option 5.

15. Boot *fx*(1M) and repartition the system disk so that it meets your needs. The example below shows how to use *fx* to switch from separate root and usr partitions to a single root partition.

```
>> boot stand/fx
84032+11488+3024+331696+26176d+4088+6240 entry: 0x89f97610
114208+29264+19536+2817088+60880d+7192+11056 entry: 0x89cd31c0
Currently in safe read-only mode.
Do you require extended mode with all options available? (no) <Enter>
SGI Version 5.3 ARCS   Dec 14, 1994
fx: "device-name" = (dksc) <Enter>
fx: ctlr# = (0) <Enter>
fx: drive# = (1) <Enter>
...opening dksc(0,1,0)
...controller test...OK
Scsi drive type == SGI     SEAGATE ST31200N8640

----- please choose one (? for help, .. to quit this menu)-----
[exi]t             [d]ebug/           [l]abel/           [a]uto
[b]adblock/        [exe]rcise/        [r]epartition/     [f]ormat
fx> repartition/rootdrive
```

```
fx/repartition/rootdrive: type of data partition = (xfs) <Enter>
Warning: you will need to re-install all software and restore user data
from backups after changing the partition layout.  Changing partitions
will cause all data on the drive to be lost.  Be sure you have the drive
backed up if it contains any user data.  Continue? yes

----- please choose one (? for help, .. to quit this menu)-----
[exi]t              [d]ebug/            [l]abel/            [a]uto
[b]adblock/         [exe]rcise/         [r]epartition/      [f]ormat
fx> exit
```

16. Load the miniroot again, using the same procedure you used in step 9.
17. Switch to the shell prompt in *inst*:

    ```
    Inst> sh
    ```

18. Unmount all filesystems except the miniroot:

    ```
    # umount -b /,/proc
    ```

19. Make an XFS filesystem for root:

```
# mkfs -d name=rootpartition -b size=blocksize -l internal,size=logsize
```

blocksize is the filesystem block size (see the section “Choosing Block Sizes” in this chapter) and *logsize* is the size of the area dedicated to log records (see the section “Choosing the Log Type and Size” in this chapter).

Example 2-4 shows an example of this command for a root filesystem and the command output. The filesystem is made on */dev/dsk/dks0d1s0* with a block size of 512 bytes and a log size of 500 KB (1000 blocks * 512 bytes / block).

Example 2-4 Example *mkfs* Command and Output for the Root Filesystem

```
# mkfs -d name=/dev/dsk/dks0d1s0 -b size=512 -l internal,size=1000b
meta-data=/dev/dsk/dks0d1s0      isize=256    agcount=1, agsize=51054 blks
data     =                       bsize=512    blocks=51054
log      =internal log           bsize=512    blocks=1000
realtime =none                   bsize=65536  blocks=0, rtextents=0
```

20. If you have a separate usr filesystem, give this command to make it an XFS filesystem:

```
# mkfs -d name=usrpartition -b size=usrblocksize -l internal,size=usrlogsize
```

    usrblocksize and *usrlogsize* are the block size and log size you've chosen for the usr filesystem.

21. Mount the root filesystem with this command:

    ```
    # mount rootpartition /root
    ```

22. If you have a separate usr filesystem, create the */usr* mount point directory and mount the filesystem with these commands:

    ```
    # mkdir /root/usr
    # mount usrpartition /root/usr
    ```

23. If you made the backup on disk, create a mount point for the filesystem that contains the backup and mount it:

    ```
    # mkdir /backupfs
    # mount backupdevice /backupfs
    ```

24. If you made the backup on tape, restore all files on the root filesystem from the backup you made in step 11 by putting the correct tape in the tape drive and giving these commands:

    ```
    # cd /root
    # mt -t /dev/rmt/tps<tapecntlr>d<tapeunit> rewind
    # restore rf dumpdevice
    ```

    You may need to be patient while the restore is taking place; it normally doesn't generate any output and it can take a while. The *restore*(1M) reference page is included in Appendix B.

25. If you made the backup on disk, restore all files on the root filesystem from the backup you made in step 11 by giving these commands:

    ```
    # cd /root
    # restore rf /backupfs/root.dump
    ```

26. If you made a backup of the usr filesystem in step 12 on tape, restore all files in the backup by putting the correct tape in the tape drive and giving these commands:

    ```
    # cd /root/usr
    # mt -t /dev/rmt/tps<tapecntlr>d<tapeunit> rewind
    # restore rf dumpdevice
    ```

27. If you made a backup of the usr filesystem in step 12 on disk, restore all files in the backup by giving these commands:

    ```
    # cd /root/usr
    # restore rf /backupfs/usr.dump
    ```

28. Move the new version of */etc/fstab* that you created in step 7 into place:

    ```
    # mv /root/etc/fstab.xfs /root/etc/fstab
    ```

29. Exit from the shell and *inst* and restart the system:

```
# exit
#
Calculating sizes .. 100% Done.

Inst> quit
...
Ready to restart the system.  Restart? { (y)es, (n)o, (sh)ell, (h)elp }: yes
Preparing to restart system ...

The system is being restarted.
```

Converting a Filesystem on an Option Disk from EFS to XFS

This section explains how to convert an EFS filesystem on an option disk (a disk other than the system disk) to XFS. It assumes that neither *lv* nor XLV logical volumes are used. You must be superuser to perform this procedure.

1. Review the subsections within the section “Planning for XFS Filesystems” in this chapter to verify that you are ready to begin this procedure.
2. Verify that your backups are up to date. Because this procedure temporarily removes all files from the filesystem you convert, it is important that you have a complete set of backups that have been prepared using your normal backup procedures. You will make a complete dump of the system disk in step 4, but you should have your usual backups in addition to the backup made during this procedure.

3. Identify the device name of the partition, *partition*, where you plan to create the filesystem. For example, if you plan to use partition 7 (the entire disk) of a SCSI option disk on controller 0, unit 2, *partition* is /dev/dsk/dks0d2s7. For more information on determining *partition* (also known as a *special* file), see *dks*(7M) for SCSI disks and *ipi*(7M) for Xylogics IPI disks.

4. Back up all files on the disk partition to tape or disk because they will be destroyed by the conversion process. You can use any backup utility (*Backup*, *bru*, *cpio*, *tar*, and so on) and backup to a local or remote tape drive or a local or remote disk. For example, the command for *dump* for local tape is:

   ```
   # dump 0uCf tapesize dumpdevice partition
   ```

   tapesize is the tape capacity (it's used for backup to disks, too) and *dumpdevice* is the device name for the tape drive. Table 2-3 gives the values of *tapesize* and *dumpdevice* for different local tape drives and disk. You can get the values of <*tapecntlr*> and <*tapeunit*> used in the table from the command *hinv –c tape*. The *dump*(1M) reference page is included in Appendix B.

5. Unmount the partition:

   ```
   umount partition
   ```

6. Use the *mkfs*(1M) command to create the new XFS filesystem:

   ```
   mkfs -d name=partition -b size=blocksize -l internal,size=logsize
   ```

   blocksize is the filesystem block size (see the section "Choosing Block Sizes" in this chapter) and *logsize* is the size of the area dedicated to log records (see the section "Choosing the Log Type and Size" in this chapter). Example 2-1 shows an example of this command line and its output.

7. Mount the new filesystem with this command:

   ```
   mount partition mountdir
   ```

8. In the file */etc/fstab*, in the entry for *partition*, replace `efs` with `xfs`. For example:

   partition mountdir `xfs rw,raw=`*rawpartition* `0 0`

   where *rawpartition* is the raw version of *partition*.

9. Restore the files to the filesystem from the backup you made in step 4. For example, if you gave the *dump* command in step 4, the commands to restore the files from tape are:

   ```
   # cd mountdir
   # mt -t device rewind
   # restore rf dumpdevice
   ```

   The value of *device* is the same as *dumpdevice* without `nsv` or other letters at the end. The *restore*(1M) reference page is included in Appendix B.

   You may need to be patient while the restore is taking place; it doesn't generate any output and it can take a while.

Checking Filesystem Consistency

The filesystem consistency checking program for XFS filesystems is *xfs_check*(1M). (*fsck*(1M) is used only for EFS filesystems.) Unlike *fsck*, *xfs_check* is not invoked automatically on system startup; *xfs_check* should be used only if you suspect a filesystem consistency problem. Before running *xfs_check*, the filesystem to be checked should be unmounted.

The command line used for *xfs_check* depends upon the underlying device:

- If the filesystem is on an XLV logical volume, the *xfs_check* command line is:

  `xfs_check` *xlvvolume*

  xlvvolume is the device file for the logical volume, for example /dev/dsk/xlv/xlv0.

- If the filesystem is on a disk partition, the *xfs_check* command line is:

  `xfs_check -d` *partition*

  partition is the device name for the partition, for example /dev/dsk/dks0d2s7.
- If the filesystem is on an *lv* logical volume, the *xfs_check* command line is:

  `xfs_check -d` *lvvolume*

  lvvolume is the device file for the logical volume, for example /dev/dsk/lv0.

Unlike *fsck*, *xfs_check* does not repair any reported filesystem consistency problems; it only reports them. If *xfs_check* reports a filesystem consistency problem:

- If possible, contact the Silicon Graphics Technical Assistance Center for assistance (see the Release Notes for this product for more information).
- To attempt to recover from the problem, follow this procedure:
  1. Mount the filesystem, but be very careful not to write to it.
  2. Make a filesystem backup with *xfsdump*.
  3. Use *mkfs* to a make new filesystem on the same disk partition or XLV logical volume.
  4. Restore the files from the backup.

Chapter 3

Dumping and Restoring XFS Filesystems

This chapter describes how the *xfsdump* and *xfsrestore* utilities work and how to use them to back up and recover data on XFS filesystems. (The *xfsdump*(1M) and *xfsrestore*(1M) reference pages provide online information on these utilities.) A short section at the end of this chapter, "Other Backup Utilities," discusses XFS-related issues of other utilities that can be used to perform backups.

This chapter contains the following sections:

- About the *xfsdump* and *xfsrestore* Utilities
- Using *xfsdump*
- Using *xfsrestore*
- Dump and Restore with STDIN / STOUT
- Other Backup Utilities and XFS

Table 3-1 and Table 3-2 summarize when to use *xfsdump* and *xfsrestore* and when their EFS counterparts, *dump*(1M) and *restore*(1M), must be used.

Table 3-1 Filesystems and Dump Utilities

For a Filesystem of Type	Dump It Using
EFS	*dump*
XFS	*xfsdump*

Table 3-2 Filesystems and Restore Utilities

For a Dump Made Using	Restore It Using	On a Filesystem of Type
dump	*restore*	EFS or XFS
xfsdump	*xfsrestore*	EFS or XFS

Note than you can restore data in either EFS or XFS filesystems, but must use the restore utility that corresponds with the dump utility used to make the backup. The *xfsdump* and *xfsrestore* utilities are only available with the XFS filesystem.

About the *xfsdump* and *xfsrestore* Utilities

This section provides an overview of the features of the *xfsdump* and *xfsrestore* utilities and describes the data format on storage media that supports this functionality.

Features of *xfsdump* and *xfsrestore*

This section summarizes the features of *xfsdump* and *xfsrestore*. Flexibility of operation, integration with system software, ease of use, and record keeping abilities are discussed.

Flexibility

- With *xfsdump* and *xfsrestore*, you can back up and restore data using local or remote drives. Multiple dumps can be placed on a single media object.
- *xfsdump* and *xfsrestore* support incremental dumps. Also, you can back up filesystems, directories, and / or individual files, and then restore filesystems, directories, and files independent of how they were backed up.
- With *xfsdump* and *xfsrestore*, you can recover from intentional or accidental interruptions.
- With *xfsrestore*, you can restore *xfsdump* data onto EFS filesystems. (*xfsdump* backs up mounted XFS filesystems only.)

Integration

- *xfsdump* and *xfsrestore* support XFS features including 64-bit inode numbers, file lengths, holes, and user-selectable extent sizes.
- *xfsdump* and *xfsrestore* support multiple media types, all IRIX-supported file types (regular, directory, symbolic link, block and character special, FIFO, and socket), and retain hard links.
- *xfsdump* does not affect the state of the filesystem being dumped (for example, access times are retained), and *xfsrestore* restores files as close to the original as possible.
- *xfsrestore* detects and bypasses media errors and recovers rapidly after encountering them.
- *xfsdump* does not cross mount points, local or remote.

User Interface

- *xfsdump* optionally prompts for additional media when the end of the current media is reached. Operator estimates of media capacity are not required. *xfsdump* also supports automated backups.
- *xfsdump* maintains an extensive online inventory of all dumps performed. Inventory contents can be viewed through various filters to quickly locate specific dump information.
- *xfsrestore* supports interactive operation, allowing selection of individual files or directories for recovery. It also permits selection from among backups performed at different times when multiple dumps are available.
- Dump contents may also be viewed noninteractively.

Media Layout

The following section introduces some terminology and then describes the way *xfsdump* formats data on the storage media for use by *xfsrestore*.

Terminology

This section introduces terminology used in the rest of this chapter.

While *xfsdump* and *xfsrestore* are often used with tape media, the utilities actually support multiple kinds of media, so in the following discussions, the term *media object* is used to refer to the media in a generic fashion. The term *dump* refers to the result of a single use of the *xfsdump* command to output data files to the selected media object(s). An instance of the use of *xfsdump* is referred to as a *dump session*.

The dump session sends a single *dump stream* to the media object(s). The dump stream may contain as little as a single file or as much as an entire filesystem. The dump stream is composed of *dump objects*, which are:

- one or more *data segments*
- an optional *dump inventory*
- a *stream terminator*

The data segment(s) contains the actual data, the dump inventory contains a list of the dump objects in the dump, and the stream terminator marks the end of the dump stream. When a dump stream is composed of multiple dump objects, each object is contained in a *media file*. Some output devices, for example standard output, do not support the concept of media files—the dump stream is only the data.

Possible Dump Layouts

The simplest dump, for example the dump of a small amount of data to a single tape, produces a data segment and a stream terminator as the only dump objects. If the optional inventory object is added, you have a dump such as that illustrated in Figure 3-1. (In the data layout diagrams in this section, the optional inventory object is always included.)

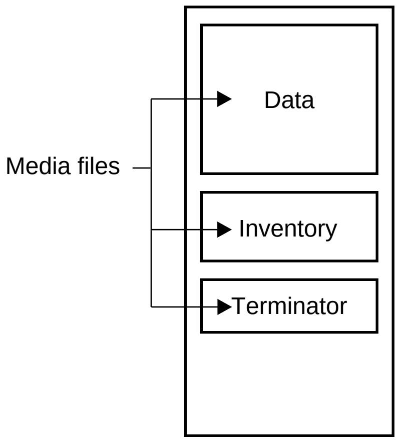

Figure 3-1 Single Dump on Single Media Object

You can also dump data streams that are larger than a single media object. The data stream can be broken between any two media files including data segment boundaries. (The inventory is never broken into segments.) The *xfsdump* utility prompts for a new media object when the end of the current media object is reached. Figure 3-2 illustrates the data layout of a single dump session that requires two media objects.

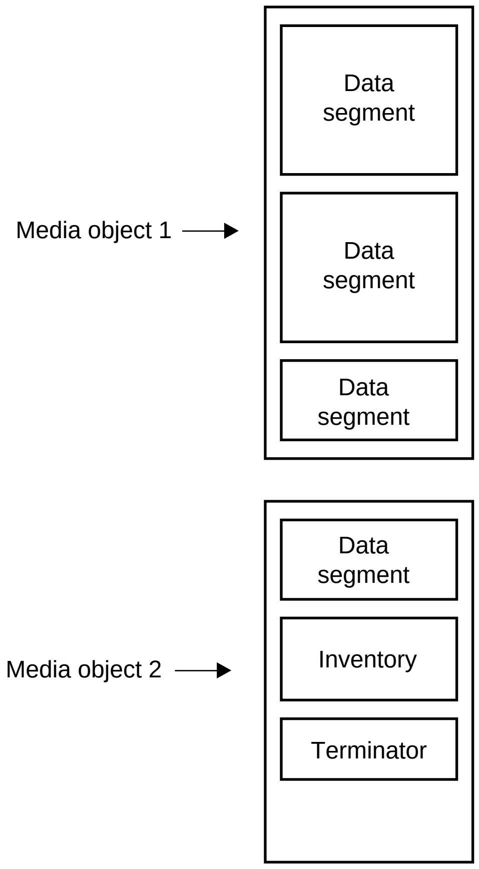

Figure 3-2 Single Dump on Multiple Media Objects

The *xfsdump* utility also accommodates multiple dumps on a single media object. When dumping to tape, for example, the tape is automatically advanced past the existing dump session(s) and the existing stream terminator is erased. The new dump data is then written, followed by the new stream terminator[1]. Figure 3-3 illustrates the layout of media files for two dumps on a single media object.

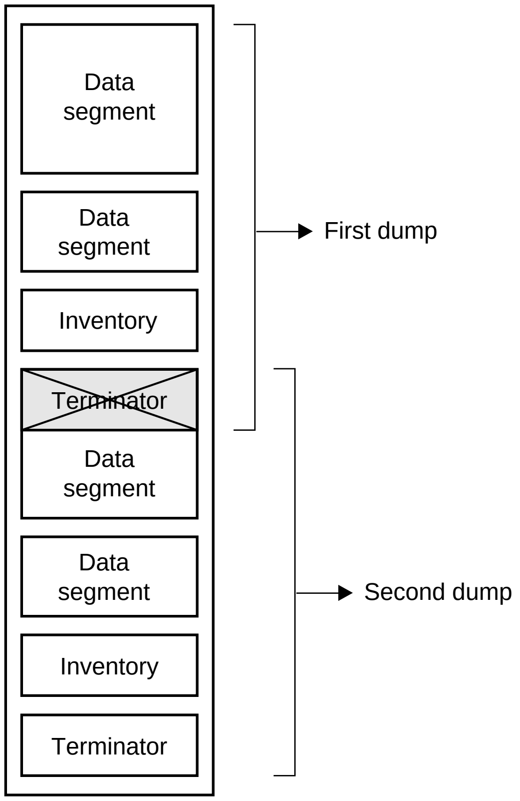

Figure 3-3 Multiple Dumps on Single Media Object

[1] For drives that do not permit termination to operate in this way, other means are used to achieve the same effective result.

Figure 3-4 illustrates a case in which multiple dumps use multiple media objects. If media files already exist on the additional media object(s), the *xfsdump* utility finds the existing stream terminator, erases it, and begins writing the new dump data stream.

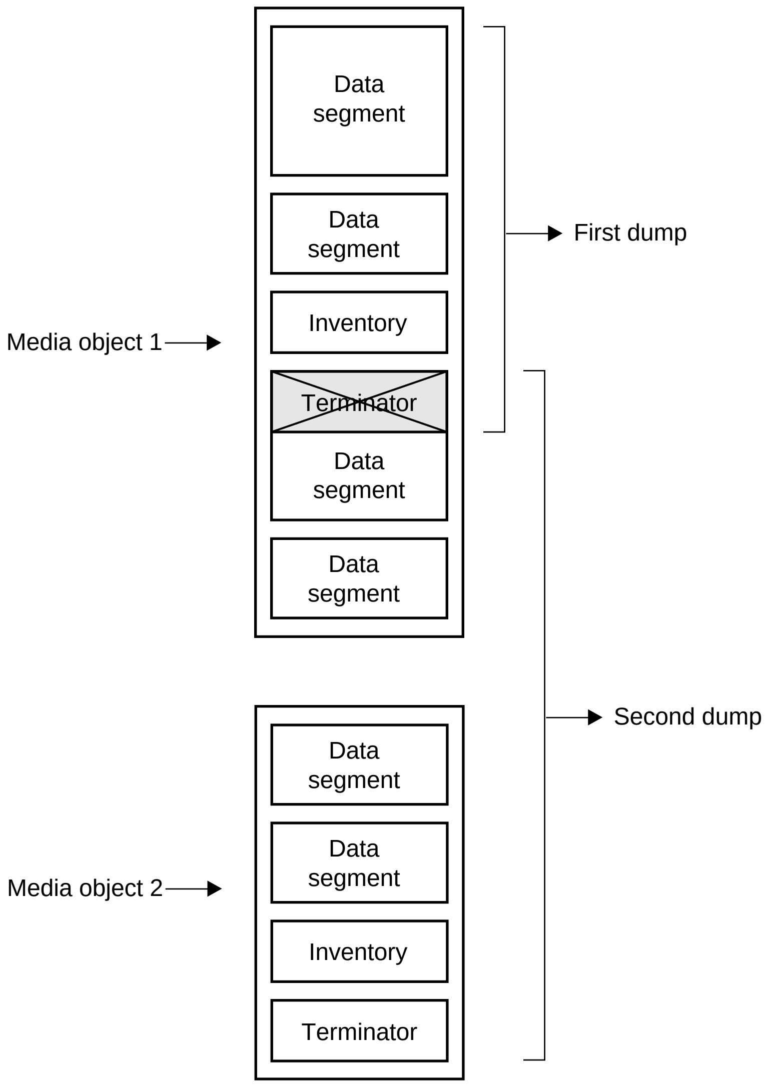

Figure 3-4 Multiple Dumps on Multiple Media Objects

Using *xfsdump*

This section discusses how to use the *xfsdump* command to backup data to local and remote devices. You can get a summary of *xfsdump* syntax with the **–h** option:

```
# xfsdump -h
xfsdump: version X.X
xfsdump: usage: xfsdump [ -f <destination> ]
                        [ -h (help) ]
                        [ -l <level> ]
                        [ -s <subtree> ... ]
                        [ -v <verbosity {silent, verbose, trace}> ]
                        [ -F (don't prompt) ]
                        [ -I (display dump inventory) ]
                        [ -J (inhibit inventory update) ]
                        [ -L <session label> ]
                        [ -M <media label> ]
                        [ -R (resume) ]
                        [ - (stdout) ]
                        <source (mntpnt|device)>
```

You must be the superuser to use *xfsdump*. Refer to the *xfsdump*(1M) reference page for details.

Specifying Media

You can use *xfsdump* to back up data to various media. For example, you can dump data to a tape or hard disk. The drive containing the media object may be connected to the local system or accessible over the network.

Backing Up to a Local Tape Drive

Following is an example of a level 0 dump to a local tape drive. Note that dump level does not need to be specified for a level 0 dump. (Refer to “Incremental and Resumed Dumps” on page 42 for a discussion of dump levels.)

```
# xfsdump -f /dev/tape -L testers_11_21_94 -M test_1 /usr
xfsdump: version 1.0 - type ^C for status and control
xfsdump: level 0 dump of magnolia.wpd.sgi.com:/usr
xfsdump: dump date: Thu Dec 15 10:15:56 1994
xfsdump: session id: d23b2d9e-b21d-1001-887f-080069068eeb
xfsdump: session label: "testers_11_21_94"
xfsdump: preparing tape drive
xfsdump: no previous dumps on tape
xfsdump: ino map phase 1: skipping (no subtrees specified)
xfsdump: ino map phase 2: constructing initial dump list
xfsdump: ino map phase 3: skipping (no pruning necessary)
xfsdump: ino map phase 4: estimating dump size
xfsdump: ino map phase 5: skipping (only one dump stream)
xfsdump: ino map construction complete
xfsdump: beginning media file
xfsdump: media file 0 (media 0, file 0)
xfsdump: dumping ino map
xfsdump: dumping directories
xfsdump: dumping non-directory files
xfsdump: ending media file
xfsdump: media file size 16871936 bytes
xfsdump: dumping session inventory
xfsdump: beginning inventory media file
xfsdump: media file 1 (media 0, file 1)
xfsdump: ending inventory media file
xfsdump: inventory media file size 2102812 bytes
xfsdump: dump complete: 207 seconds elapsed
```

In this case, a session label (**–L** option) and a media label (**–M** option) are supplied, and the entire filesystem is dumped. Since no verbosity option is supplied, the default of *verbose* is used, resulting in the detailed screen output. The dump inventory is updated with the record of this backup because the **-J** option is not specified.

Following is an example of a backup of a subdirectory of a filesystem. In this example, the verbosity is set to *silent*, and the dump inventory is not updated (**–J** option):

```
# xfsdump -f /dev/tape -v silent -J -s people/fred /usr
```

Note that the subdirectory backed up (*/usr/people/fred*) was specified relative to the filesystem, so the specification did not include the name of the filesystem (in this case, */usr*). Since */usr* may be a very large filesystem and the **`-v silent`** option was used, this could take a long time during which there would be no screen output.

Backing Up to a Remote Tape Drive

To back up data to a remote tape drive, use the standard remote system syntax, specifying the system (by hostname if supported by a nameserver or IP address if not) followed by a colon (:), then the pathname of the special file.

Note: For remote backups, use the variable block size tape device if the device supports variable block size operation, otherwise use the fixed block size device (see *intro*(7)).

The following example shows a subtree backup with no inventory to a remote tape device:

```
# xfsdump -f theduke:/dev/rmt/tps0d2v -J -s people/fred /usr
xfsdump: version X.X - type ^C for status and control
xfsdump: dump date: Mon Nov 21 13:56:01 1994
xfsdump: level 0 dump
xfsdump: preparing tape drive
xfsdump: end of data found
xfsdump: ino map phase 1: parsing subtree selections
xfsdump: ino map phase 2: constructing initial dump list
xfsdump: ino map phase 3: pruning unneeded subtrees
xfsdump: ino map phase 4: estimating dump size
xfsdump: ino map phase 5: skipping (only one dump stream)
xfsdump: ino map construction complete
xfsdump: beginning media file
xfsdump: dumping ino map
xfsdump: dumping directories
xfsdump: dumping non-dir files
xfsdump: ending media file
xfsdump: media file size 15190208 bytes
```

In this case, */usr/people/fred* is backed up to the variable block size tape device on the remote system *theduke*.

Note: The superuser account on the local system must be able to *rsh* to the remote system without a password. For more information, see *hosts.equiv*(4).

Backing Up to a File

You can back up data to a file instead of a device. In the following example, a file (*Makefile*) and a directory (*Source*) are backed up to a dump file (*monday_backup*) in */usr/tmp* on the local system:

```
# xfsdump -f /usr/tmp/monday_backup -v silent -J -s \
people/fred/Makefile -s people/fred/Source /usr
```

You may also dump to a file on a remote system, but note that the file must be in the remote system's */dev* directory. For example, the following command backs up the */usr/people/fred* subdirectory on the local system to the regular file */dev/fred_mon_12-2* on the remote system theduke:

```
# xfsdump -f theduke:/dev/fred_mon_12-2 -s people/fred /usr
```

Alternatively, you could dump to any remote file if that file is on an NFS-mounted filesystem. In any case, permission settings on the remote system must allow to write to the file.

Refer to the section "Dump and Restore With STDIN / STDOUT" on page 56 for information on using the standard input and standard output capabilities of *xfsdump* and *xfsrestore* to pipe data between filesystems or across the network.

Reusing Tapes

When you use a new tape as the media object of a dump session, *xfsdump* begins writing dump data at the beginning of the tape without prompting. If the tape already has dump data on it, *xfsdump* begins writing data after the last dump stream, again without prompting.

If, however, the tape contains data that is not from a dump session, *xfsdump* prompts you before continuing:

```
# xfsdump -f /dev/tape /test
xfsdump: version X.X - type ^C for status and control
xfsdump: dump date: Fri Dec 2 11:25:19 1994
xfsdump: level 0 dump
xfsdump: session id: d23cc072-b21d-1001-8f97-080069068eeb
xfsdump: preparing tape drive
xfsdump: this tape contains data that is not part of an XFS dump
xfsdump: do you want to overwrite this tape?
type y to overwrite, n to change tapes or abort (y/n):
```

You must answer **y** if you want to continue with the dump session, or **n** to quit. If you answer **y**, the dump session resumes and the tape is overwritten. If you do not respond to the prompt, the session will eventually timeout. Note that this means that an automatic backup, for example one initiated by a *crontab* entry, will not succeed—unless you specified the **-F** option with the *xfsdump* command, which forces it to overwrite the tape rather than prompt for approval.

Erasing Used Tapes

Erase pre-existing data on tapes with the *mt erase* command. Make sure the tape is not write-protected.

For example, to prepare a used tape in the local default tape drive, enter:

```
# mt -f /dev/tape erase
```

Caution: This erases all data on the tape, including any dump sessions.

The tape can now used by *xfsdump* without prompting for approval.

Incremental and Resumed Dumps

Incremental dumps are a way of backing up less data at a time but still preserving current versions of all your backed-up files, directories, and so on. Incremental backups are organized numerically by levels from 0 through 9. A level 0 dump always backs up the complete filesystem. A dump level of any other number backs up all files that have changed since a dump with a lower dump level number.

For example, if you perform a level 2 backup on a filesystem one day and your next dump is a level 3 backup, only those files that have changed since the level 2 backup are dumped with the level 3 backup. In this case, the level 2 backup is called the *base dump* for the level 3 backup. The base dump is the most recent backup of that filesystem with a lower dump level number.

Resumed dumps work in much the same way. When a dump is resumed after it has been interrupted, the remaining files that had been scheduled to be backed up during the interrupted dump session are backed up, and any files that changed during the interruption are also backed up. Note that you must restore an interrupted dump as if it is an incremental dump (see "Cumulative Restores" on page 52).

Incremental Dump Example

In the following example, a new tape is used and the level 0 dump is the first dump written to it:

```
# xfsdump -f /dev/tape -l 0 -M Jun_94 -L week_1 -v silent /usr
```

A week later, a level 1 dump of the filesystem is performed on the same tape:

```
# xfsdump -f /dev/tape -l 1 -L week_2 /usr
```

The tape is forwarded past the existing dump data and the new data from the level 1 dump is written after it. (Note that it is not necessary to specify the media label for each successive dump on a media object.)

A week later, a level 2 dump is taken:

```
# xfsdump -f /dev/tape -l 2 -L week_3 /usr
```

and so on, for the four weeks of a month in this example, the fourth week being a level 3 dump (up to nine dump levels are supported). Refer to “Cumulative Restores” on page 52 for information on the proper procedure for restoring incremental dumps.

Resumed Dump Example

You can interrupt a dump session and resume it later. To interrupt a dump session, type the interrupt character (typically CTRL-C). You receive a list of options which allow you to interrupt the session, change verbosity level, or resume the session.

In the following example, *xfsdump* is interrupted after dumping approximately 20% of a filesystem:

```
# xfsdump -f /dev/tape -L 210994u -v silent /usr
xfsdump: this tape contains data that is not part of an XFS dump
xfsdump: do you want to overwrite this tape?
type y to overwrite, n to change tapes or abort (y/n): y
overwriting
^C
status: 91/168 files dumped, 20.48 percent complete, 70 seconds elapsed
0: interrupt this session
1: change verbosity
2: continue
 -> 0
session interrupt initiated
xfsdump: dump interrupted prior to ino 11615 offset 0
```

You can later continue the dump by including the **–R** option and a different session label:

```
# xfsdump -f /dev/tape -R -L 2nd210994u -v silent /usr
```

Any files that were not backed up before the interruption, and any file changes that were made during the interruption, are backed up after the dump is resumed.

Note: Use of the **–R** option requires that the dump was made with a dump inventory taken, that is, the **–J** option was not used with *xfsdump*.

Viewing the Dump Inventory

The dump inventory is maintained in the directory */var/xfsdump* created by *xfsdump*. You can view the dump inventory at any time with the *xfsdump* **–I** command. With no other arguments, *xfsdump* **–I** displays the entire dump inventory. (The *xfsdump* **-I** command does not require root privileges.)

The following output presents a section of a dump inventory.

```
# xfsdump -I | more
file system 0:
        fs id:          d23cb450-b21d-1001-8f97-080069068eeb
        session 0:
                mount point:    magnolia.wpd.xyz.com:/test
                device:         magnolia.wpd.xyz.com:/dev/rdsk/dks0d3s2
                time:           Mon Nov 28 11:44:04 1994
                session label:  ""
                session id:     d23cbf44-b21d-1001-8f97-080069068eeb
                level:          0
                resumed:        NO
                subtree:        NO
                streams:        1
                stream 0:
                        pathname:       /dev/tape
                        start:          ino 4121 offset 0
                        end:            ino 0 offset 0
                        interrupted:    YES
                        media files:    2
                        media file 0:
                                mfile index:    0
---more---
```

Notice that the dump inventory records are presented sequentially and are indented to illustrate the hierarchical order of the dump information.

You can view a subset of the dump inventory by specifying the level of depth (1, 2, or 3) that you want to view. For example, specifying depth=2 filters out a lot of the specific dump information as you can see by comparing the previous output with this:

```
# xfsdump -I depth=2
file system 0:
        fs id:          d23cb450-b21d-1001-8f97-080069068eeb
        session 0:
                mount point:    magnolia.wpd.xyz.com:/test
                device:         magnolia.wpd.xyz.com:/dev/rdsk/dks0d3s2
                time:           Mon Nov 28 11:44:04 1994
                session label:  ""
                session id:     d23cbf44-b21d-1001-8f97-080069068eeb
                level:          0
                resumed:        NO
                subtree:        NO
                streams:        1
        session 1:
                mount point:    magnolia.wpd.xyz.com:/test
                device:         magnolia.wpd.xyz.com:/dev/rdsk/dks0d3s2
               .
               .
               .
```

You can also view a filesystem-specific inventory by specifying the filesystem mount point with the *mnt* option. The following output shows an example of a dump inventory display in which the *depth* is set to *1*, and only a single filesystem is displayed:

```
# xfsdump -I depth=1,mnt=magnolia.wpd.xyz.com:/test
file system 0:
        fs id:          d23cb450-b21d-1001-8f97-080069068eeb
```

Note that you can also look at a list of contents on the dump media itself by using the **–t** option with *xfsrestore*. (The *xfsrestore* utility is discussed in detail in the following section.) For example, to list the contents of the dump tape currently in the local tape drive:

```
# xfsrestore -f /dev/tape -t -v silent | more
xfsrestore: dump session found
xfsrestore: session label: "week_1"
xfsrestore: session id: d23cbcb4-b21d-1001-8f97-080069068eeb
xfsrestore: no media label
xfsrestore: media id: d23cbcb5-b21d-1001-8f97-080069068eeb
do you want to select this dump? (y/n): y
selected
one
A/five
people/fred/TOC
people/fred/ch3.doc
people/fred/ch3TOC.doc
people/fred/questions
A/four
people/fred/script_0
people/fred/script_1
people/fred/script_2
people/fred/script_3
people/fred/sub1/TOC
people/fred/sub1/ch3.doc
people/fred/sub1/ch3TOC.doc
people/fred/sub1/questions
people/fred/sub1/script_0
people/fred/sub1/script_1
people/fred/sub1/script_2
people/fred/sub1/script_3
people/fred/sub1/xdump1.doc
people/fred/sub1/xdump1.doc.backup
people/fred/sub1/xfsdump.doc
people/fred/sub1/xfsdump.doc.auto
people/fred/sub1/sub2/TOC
---more---
```

Using *xfsrestore*

This section discusses the *xfsrestore* command, which you must use to view and extract data from the dump data created by *xfsdump*. You can get a summary of *xfsrestore* syntax with the **–h** option:

```
# xfsrestore -h
xfsrestore: version X.X
xfsrestore: usage: xfsrestore [ -a <alt. workspace dir> ]
                              [ -e (don't overwrite existing files)]
                              [ -f <source> ]
                              [ -h (help) ]
                              [ -i (interactive) ]
                              [ -n <file> (restore only if newer than) ]
                              [ -r (cumulative restore) ]
                              [ -s <subtree> ... ]
                              [ -t (contents only) ]
                              [ -v <verbosity {silent, verbose, trace}> ]
                              [ -E (don't overwrite if changed) ]
                              [ -I (display dump inventory) ]
                              [ -L <session label> ]
                              [ -R (resume) ]
                              [ -S <session id> ]
                              [ - (stdin) ]
                              [ <destination> ]
```

You must be the superuser to use *xfsrestore*. Refer to the *xfsrestore*(1M) reference page for additional information.

xfsrestore Operations

Use *xfsrestore* to restore data backed up with *xfsdump*. You can restore files, subdirectories, and filesystems—regardless of the way they were backed up. For example, if you back up an entire filesystem in a single dump, you can select individual files and subdirectories from within that filesystem to restore.

You can use *xfsrestore* interactively or noninteractively. With interactive mode, you can peruse the filesystem or files backed up, selecting those you want to restore. In noninteractive operation, a single command line can restore selected files and subdirectories, or an entire filesystem. You can restore data to its original filesystem location or any other location in an EFS or XFS filesystem.

By using successive invocations of *xfsrestore*, you can restore incremental dumps on a base dump. This restores data in the same sequence it was dumped.

Simple Restores

A simple restore is a non-cumulative restore (for information on restoring incremental dumps, refer to “Cumulative Restores” on page 52). An example of a simple, noninteractive use of *xfsrestore* is:

```
# xfsrestore -f /dev/tape /usr
xfsrestore: version X.X - type ^C for status and control
xfsrestore: preparing tape drive
xfsrestore: dump session found
xfsrestore: no session label
xfsrestore: session id: d23cbbbe-b21d-1001-8f97-080069068eeb
xfsrestore: no media label
xfsrestore: media id: d23cbbbf-b21d-1001-8f97-080069068eeb
do you want to restore this dump? (y/n): y
beginning restore
xfsrestore: restore of level 0 dump of magnolia.wpd.xyz.com:/usr created Tue
Nov 22 15:47:54 1994
xfsrestore: beginning media file
xfsrestore: reading ino map
xfsrestore: initializing the map tree
xfsrestore: reading the directory hierarchy
xfsrestore: restoring non-directory files
xfsrestore: ending media file
xfsrestore: restoring directory attributes
xfsrestore: restore complete: 115 seconds elapsed
```

In this case, *xfsrestore* went to the first dump on the tape and asked if this was the dump to restore. If you had answered **n**, *xfsrestore* would have proceeded to the next dump on the tape (if there was one) and asked if this was the dump you wanted to restore.

You can request a specific dump if you used *xfsdump* with a session label. For example:

```
# xfsrestore -f /dev/tape -L Wed_11_23 /usr
xfsrestore: version X.X - type ^C for status and control
xfsrestore: preparing tape drive
xfsrestore: dump session found
xfsrestore: advancing tape to next media file
xfsrestore: dump session found
xfsrestore: restore of level 0 dump of magnolia.wpd.xyz.com:/usr created Wed
Nov 23 11:17:54 1994
xfsrestore: beginning media file
xfsrestore: reading ino map
xfsrestore: initializing the map tree
xfsrestore: reading the directory hierarchy
xfsrestore: restoring non-directory files
xfsrestore: ending media file
xfsrestore: restoring directory attributes
xfsrestore: restore complete: 200 seconds elapsed
```

In this way you recover a dump with a single command line and do not have to answer **y** or **n** to the prompt(s) asking you if the dump session found is the correct one. To be even more exact, use the **-S** option and specify the unique session ID of the particular dump session:

```
# xfsrestore -f /dev/tape -S \
d23cbf47-b21d-1001-8f97-080069068eeb /usr2/tmp
xfsrestore: version X.X - type ^C for status and control
xfsrestore: preparing tape drive
xfsrestore: dump session found
xfsrestore: advancing tape to next media file
xfsrestore: advancing tape to next media file
xfsrestore: dump session found
xfsrestore: restore of level 0 dump of magnolia.wpd.xyz.com:/test resumed Mon
Nov 28 11:50:41 1994
xfsrestore: beginning media file
xfsrestore: media file 0 (media 0, file 2)
xfsrestore: reading ino map
xfsrestore: initializing the map tree
xfsrestore: reading the directory hierarchy
xfsrestore: restoring non-directory files
xfsrestore: ending media file
xfsrestore: restoring directory attributes
xfsrestore: restore complete: 229 seconds elapsed
```

You can find the session ID by viewing the dump inventory (see “Viewing the Dump Inventory” on page 44). Session labels might be duplicated, but session IDs never are.

Restoring Individual Files

On the *xfsrestore* command line, you can specify an individual file or subdirectory to restore. In this example, the file *people/fred/notes* is restored and placed in the */usr/tmp* directory (that is, the file is restored in */usr/tmp/people/fred/notes*):

```
# xfsrestore -f /dev/tape -L week_1 -s people/fred/notes \
/usr/tmp
```

You can also restore a file “in place” that is, restore it directly to where it came from in the original backup. Note, however, that if you do not use a **-e**, **-E**, or **-n** option, you overwrite any existing file(s) of the same name.

In the following example, the subdirectory *people/fred* is restored in the destination */usr* - this would overwrite any files and subdirectories in */usr/people/fred* with the data on the dump tape:

```
# xfsrestore -f /dev/tape -L week_1 -s people/fred /usr
```

Network Restores

You can use standard network references to specify devices and files on the network. For example, to use the tape drive on a network host named *magnolia* as the source for a restore, you can use the command:

```
# xfsrestore -f magnolia:/dev/tape -L 120694u2 /usr2
xfsrestore: version X.X - type ^C for status and control
xfsrestore: preparing tape drive
xfsrestore: dump session found
xfsrestore: advancing tape to next media file
xfsrestore: dump session found
xfsrestore: restore of level 0 dump of magnolia.wpd.xyz.com:/usr2 created Tue
Dec 6 10:55:17 1994
xfsrestore: beginning media file
xfsrestore: media file 0 (media 0, file 1)
xfsrestore: reading ino map
xfsrestore: initializing the map tree
xfsrestore: reading the directory hierarchy
```

```
xfsrestore: restoring non-directory files
xfsrestore: ending media file
xfsrestore: restoring directory attributes
xfsrestore: restore complete: 203 seconds elapsed
```

In this case, the dump data is extracted from the tape on *magnolia* and the destination is the directory */usr2* on the local system. Refer to the section "Dump and Restore With STDIN / STDOUT" on page 56 for an example of using the standard input option of *xfsrestore*.

Interactive Restores

Use the **–i** option of *xfsrestore* to perform interactive file restoration. With interactive restoration, you can use the commands *ls*, *pwd*, and *cd* to peruse the filesystem, and the *add* and *delete* commands to create a list of files and subdirectories you want to restore. Then you can enter the *extract* command to restore the files, or *quit* to exit the interactive restore session without restoring files. (The use of "wildcards" is not allowed with these commands.)

The following screen output shows an example of a simple interactive restoration.

```
# xfsrestore -f /dev/tape -i -v silent .
xfsrestore: dump session found
xfsrestore: no session label
xfsrestore: session id:    d23cbeda-b21d-1001-8f97-080069068eeb
xfsrestore: no media label
xfsrestore: media id:      d23cbedb-b21d-1001-8f97-080069068eeb
do you want to select this dump? (y/n): y
selected

 --- interactive subtree selection dialog ---

the following commands are available:
        pwd
        ls [ { <name>, ".." } ]
        cd [ { <name>, ".." } ]
        add [ <name> ]
        delete [ <name> ]
        extract
        quit
        help
```

```
-> **ls**
         4122 people/
         4130 two
         4126 A/
         4121 one
-> add two
-> cd people
-> ls
         4124 fred/
-> add fred
-> ls
  *      4124 fred/
-> extract

---------------- end dialog ----------------
```

In the interactive restore session above, the subdirectory *people/fred* and the file *two* were restored relative to the current working directory ("."). Note that an asterisk (*) in your *ls* output indicates your selections.

Cumulative Restores

Cumulative restores sequentially restore incremental dumps to recreate filesystems and are also used to restore interrupted dumps. To perform a cumulative restore of a filesystem, begin with the media object that contains the base level dump and recover it first, then recover the incremental dump with the next higher dump level number, then the next, and so on. Use the **–r** option to inform *xfsrestore* that you are performing a cumulative recovery.

In the following example, the level 0 base dump and succeeding higher level dumps are on */dev/tape*. First the level 0 dump is restored, then each higher level dump in succession:

```
# xfsrestore -f /dev/tape -r -v silent .
xfsrestore: dump session found
xfsrestore: session label: "week_1"
xfsrestore: session id: d23cbcb4-b21d-1001-8f97-080069068eeb
xfsrestore: no media label
xfsrestore: media id: d23cbcb5-b21d-1001-8f97-080069068eeb
do you want to select this dump? (y/n): y
selected
```

Next, enter the same command again, but when prompted if you want to restore the first dump (which you've already restored), type **n**. Then type **y** in response to the continue searching question. When you come to the next dump, restore it:

```
# xfsrestore -f /dev/tape -r -v silent .
xfsrestore: dump session found
xfsrestore: session label: "week_1"
xfsrestore: session id: d23cbcb4-b21d-1001-8f97-080069068eeb
xfsrestore: no media label
xfsrestore: media id: d23cbcb5-b21d-1001-8f97-080069068eeb
do you want to select this dump? (y/n): n
not selected
do you want to continue searching? (y/n): y
continuing search
xfsrestore: dump session found
xfsrestore: session label: "week_2"
xfsrestore: session id: d23cbcb8-b21d-1001-8f97-080069068eeb
xfsrestore: no media label
xfsrestore: media id: d23cbcb5-b21d-1001-8f97-080069068eeb
do you want to select this dump? (y/n): y
selected
.
.
.
```

You then repeat this process, only now skipping the first two dumps and restoring the third, and so on, until you have recovered the entire sequence of incremental dumps. The full and latest copy of the filesystem will then have been restored. In this case, it is restored relative to ".", that is, in the directory you are in when the sequence of *xfsrestore* commands is issued.

Restoring Interrupted Dumps

Restore an interrupted dump just as if it were an incremental dump. Use the **-r** option to inform *xfsrestore* that you are performing an incremental restore, and answer **y** and **n** appropriately to select the proper "increments" to restore (see "Cumulative Restores" on page 52).

Note that if you try to restore an interrupted dump as if it were a non-interrupted, non-incremental dump, the portion of the dump that occurred before the interruption is restored, but not the remainder of the dump. You can determine if a dump is an interrupted dump by looking in the online inventory.

Here is an example of a dump inventory showing an interrupted dump session (the crucial fields are in bold type):

```
# xfsdump -I depth=3,mobjlabel=AugTape,mnt=indy4.xyz.com:/usr
file system 0:
        fs id:          d23cb450-b21d-1001-8f97-080069068eeb
        session 0:
                mount point:    indy4.xyz.com.com:/usr
                device:         indy4.xyz.com.com:/dev/rdsk/dks0d3s2
                time:           Tue Dec  6 15:01:26 1994
                session label:  "180894usr"
                session id:     d23cc0c3-b21d-1001-8f97-080069068eeb
                level:          0
                resumed:        NO
                subtree:        NO
                streams:        1
                stream 0:
                        pathname:       /dev/tape
                        start:          ino 4121 offset 0
                        end:            ino 0 offset 0
                        interrupted:    YES
                        media files:    2
        session 1:
                mount point:    indy4.xyz.com.com:/usr
                device:         indy4.xyz.com.com:/dev/rdsk/dks0d3s2
                time:           Tue Dec  6 15:48:37 1994
                session label:  "Resumed180894usr"
                session id:     d23cc0cc-b21d-1001-8f97-080069068eeb
                level:          0
                resumed:        YES
                subtree:        NO
                streams:        1
                stream 0:
                        pathname:       /dev/tape
                        start:          ino 4121 offset 0
                        end:            ino 0 offset 0
                        interrupted:    NO
                        media files:    2
.
.
.
```

From this it can be determined that session 0 was interrupted and then resumed and completed in session 1.

To restore the interrupted dump session in the example above, use the following sequence of commands:

```
# xfsrestore -f /dev/tape -r -L 180894usr .
# xfsrestore -f /dev/tape -r -L Resumed180894usr .
```

This restores the entire */usr* backup relative to the current directory. (You should remove the *housekeeping* directory from the destination directory when you are finished.)

Interrupted Restores

In a manner similar to *xfsdump* interruptions, you can interrupt an *xfsrestore* session. This allows you to interrupt a restore session and then resume it later. To interrupt a restore session, type the interrupt character (typically CTRL-C). You receive a list of options which allow you to interrupt the session, change the verbosity level, or resume the session.

```
# xfsrestore -f /dev/tape -v silent /usr
xfsrestore: dump session found
xfsrestore: no session label
xfsrestore: session id: d23cbf44-b21d-1001-8f97-080069068eeb
xfsrestore: no media label
xfsrestore: media id: d23cbf45-b21d-1001-8f97-080069068eeb
do you want to select this dump? (y/n): y
selected
status: 92/168 files restored, 41.03 percent complete, 135 seconds elapsed
0: interrupt this session
1: change verbosity
2: continue
 -> 0
session interrupt initiated
```

Resume the *xfsrestore* session with the **–R** option:

```
# xfsrestore -f /dev/tape -R -v silent /usr
```

Data recovery continues from the point of the interruption.

The *housekeeping* and *orphanage* Directories

The *xfsrestore* utility can create two subdirectories in the destination called the *housekeeping* and *orphanage* directories.

The *housekeeping* directory is a temporary directory used during cumulative recovery to pass information from one invocation of *xfsrestore* to the next. It must not be removed during the process of performing the cumulative recovery but should be removed after the cumulative recovery is completed.

The orphanage directory is created if a file or subdirectory is restored that is not referenced in the filesystem structure of the dump. For example, if you dump a very active filesystem, it is possible for new files to be in the non-directory portion of the dump, yet none of the directories dumped reference that file. A WARNING message will be displayed, and the file is placed in the *orphanage* directory, named with its original inode number.

Dump and Restore With STDIN/STDOUT

You can use *xfsdump* and *xfsrestore* to pipe data across filesystems or across the network with a single command line. By piping *xfsdump* standard output to *xfsrestore* standard input you create an exact copy of a filesystem.

For example, to make a copy of */usr/people/fred* in the */usr2* directory, enter:

```
# xfsdump -J -s people/fred - /usr | xfsrestore - /usr2
```

To copy */usr/people/fred* to the network host *magnolia*'s */usr/tmp* directory:

```
# xfsdump -J -s people/fred - /usr | rsh magnolia \
xfsrestore - /usr/tmp
```

This creates the directory */usr/tmp/people/fred* on *magnolia*.

Note: The superuser account on the local system must be able to *rsh* to the remote system without a password. For more information, see *hosts.equiv*(4).

Other Backup Utilities and XFS

You can also use the standard IRIX utilities *cpio*(1), *tar*(1), and *bru*(1), to backup and restore data. This section discusses XFS-related issues in using these utilities.

tar

The **`-K`** option has been added to the *tar*(1) command for use with files larger than 2GB. If the **`-K`** option is not used, *tar* skips any files larger than 2GB and issues a warning. Note that use of this option can create *tar* archives that are not usable on non-XFS systems . The **`-K`** option cannot be used in combination with the **`-O`** option, which creates *tar* archives formatted in an older, pre-POSIX format.

cpio

The **`-K`** option has been added to the *cpio*(1) command for use with files larger than 2GB. If the **`-K`** option is not used, *cpio* skips any files larger than 2GB and issues a warning. Note that use of this option can create *cpio* archives that are not usable on non-XFS systems. The **`-K`** option can only be used in combination with the **`-o`** (output) option. The **`-K`** option cannot be used in combination with the **`-c`** option (which creates *cpio* archives with ASCII headers), nor with the **`-H`** option (used to specify various header formats).

bru

The **`-K`** option has been added to the *bru*(1) command for use with files larger than 2GB. If the **`-K`** option is not used, *bru* skips any files that it cannot compress to less than 2GB and issues a warning. Note that use of this option can create *bru* archives that are not usable on non-XFS systems. The **`-K`** option can only be used in combination with the **`-Z`** (use 12-bit LZW file compression) option.

System Recovery

The PROM Monitor’s System Recovery option does not work correctly for XFS filesystems.

Chapter 4

XLV Logical Volumes

This chapter provides an overview of the XLV Volume Manager and explains how to create and administer XLV logical volumes. The use of logical volumes enables the creation of filesystems or raw devices that span more than one disk partition. Logical volumes behave like regular disk partitions. Filesystems can be created, mounted, and used in the normal way, or they can be used as raw devices.

This chapter contains these main sections:

Note: One feature of the XLV Volume Manager described in this chapter, plexing (mirroring), is available only when you purchase the Disk Plexing Option software option. See the *plexing Release Notes* for information on purchasing this software option and obtaining the required NetLS license.

XLV Overview

Traditionally, UNIX systems represent disk partitions as block and character devices. These “devices” are actually kernel-based interfaces that allow applications to access the partitions on either a character or block basis. The actual disk interactions are performed by disk device drivers.

Some applications, such as high-performance databases, access these partition devices directly for maximum performance. However, most applications simplify their disk access by interfacing with filesystems. Filesystems isolate applications from the concerns of disk management by providing the familiar file and directory model for disk access.

XLV interposes another layer into this model by building *logical volumes* (also known as *volumes*) on top of the partition devices. Volumes appear as block and character devices in the */dev* directory. Filesystems, databases, and other applications access the volumes rather than the partitions. Logical volumes provide services such as disk plexing (also known as mirroring) and striping transparently to the applications that access the volumes. A logical volume might include partitions from several physical disk drives and, thus, be larger than any of the physical disks. EFS or XFS filesystems can be made on XLV logical volumes.

Composition of Logical Volumes

Logical volumes are composed of a hierarchy of logical storage objects: volumes are composed of subvolumes, subvolumes are composed of plexes, and plexes are composed of volume elements. Volume elements are composed of disk partitions. This hierarchy of storage units is shown in Figure 4-1, an example of a relatively complex logical volume.

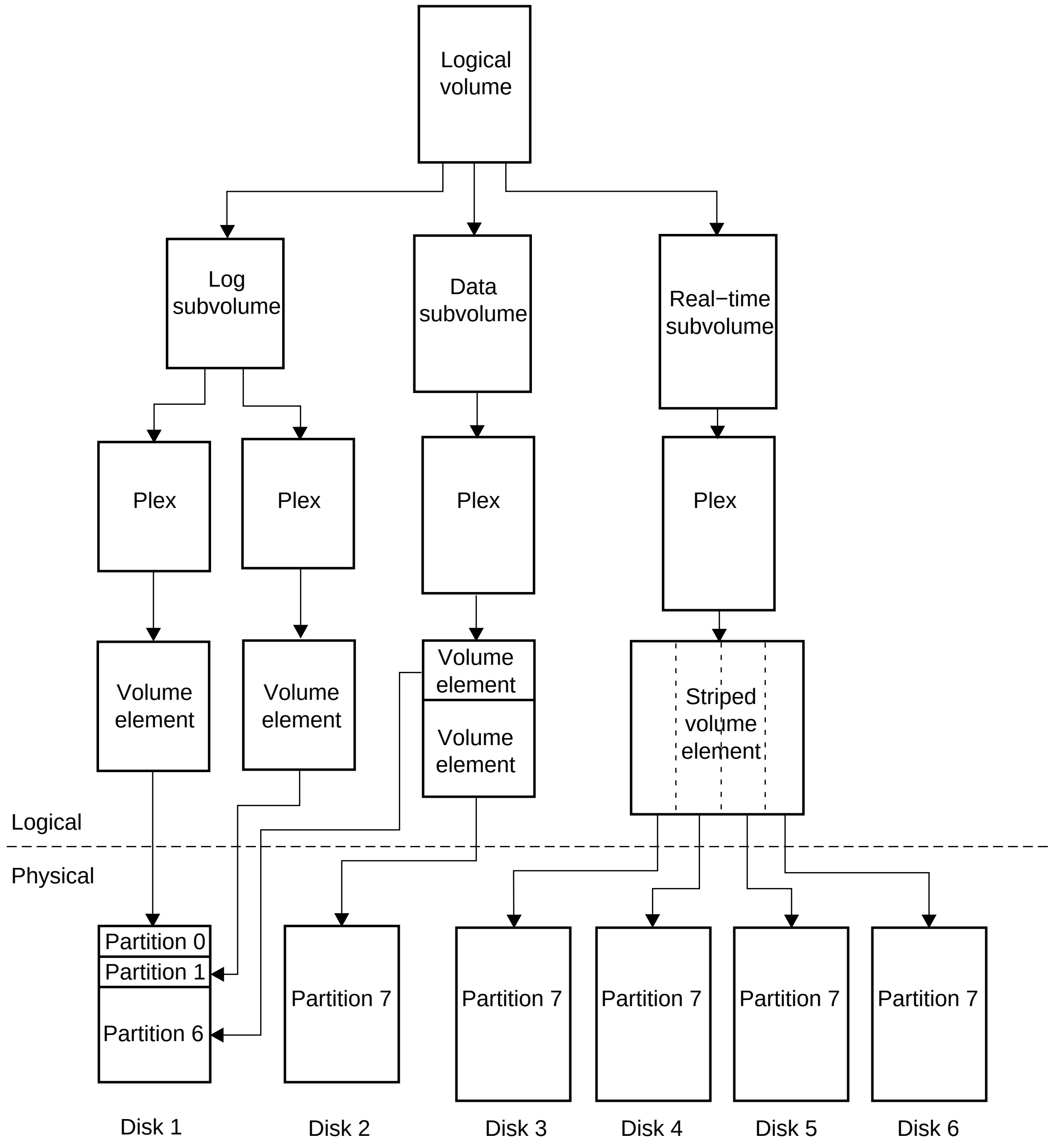

Figure 4-1 Logical Volume Example

Figure 4-1 illustrates the relationships between volumes, subvolumes, plexes, and volume elements. In this example, six physical disk drives contain eight disk partitions. The logical volume has a log subvolume, a data subvolume, and a real-time subvolume. The log subvolume has two plexes (copies of the data) for higher reliability and the data and real-time subvolumes are not plexed (meaning that they each consist of a single plex). The log plexes each consist of a volume element which is a disk partition on disk 1. The plex of the data subvolume consists of two volume elements, a partition that is the remainder of disk 1 and a partition that is all of disk 2. The plex used for the real-time subvolume is striped for increased performance. The striped volume element is constructed from four disk partitions, each of which is an entire disk.

The subsections below describe these logical storage objects in more detail.

Volumes

Volumes are composed of subvolumes. For EFS filesystems, a volume consists of just one subvolume. For XFS filesystems, a volume consists of a data subvolume, an optional log subvolume, and an optional real-time subvolume. The breakdown of a volume into subvolumes is shown in Figure 4-2.

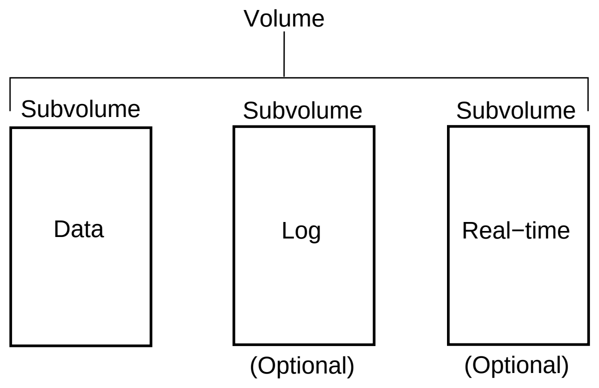

Figure 4-2 Volume Composition

Each volume can be used as a single filesystem or as a raw partition. Volume information used by the system during system startup is stored in disk labels on each disk used by the volume. At system startup, volumes won't come up if any of their subvolumes cannot be brought online. You can create volumes, delete them, and move them to another system.

Subvolumes

As explained in the section"Volumes," each logical volume is composed of one to three subvolumes, as shown in Figure 4-3. A subvolume is made up of one to four plexes.

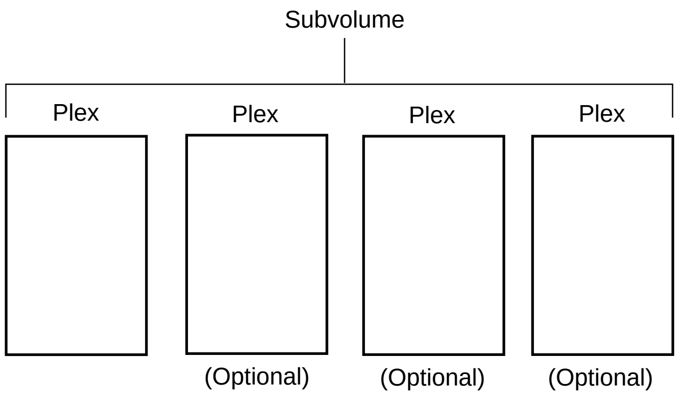

Figure 4-3 Subvolume Composition

Each subvolume is a distinct address space and a distinct type. The types of subvolumes are:

Data subvolume
: The data subvolume is required in all logical volumes. It is the only subvolume present in EFS filesystems.

Log subvolume
: The log subvolume contains XFS journaling information. It is a log of filesystem transactions and is used to expedite system recovery after a crash. Log information is sometimes put in the data subvolume rather than in a log subvolume (see the section "Choosing the Log Type and Size" in Chapter 2 and the *mkfs_xfs*(1M) reference page and its discussion of the **–l** option for more information).

Real-time subvolume
: Real-time subvolumes are generally used for data applications such as video, where guaranteed response time is more important than data integrity. The section “Using the Real-Time Subvolume” in this chapter and Chapter 5, “Guaranteed-Rate I/O,” explain how applications access data on real-time subvolumes.

Subvolumes enforce separation among data types. For example, user data cannot overwrite filesystem log data. Subvolumes also enable filesystem data and user data to be configured to meet goals for performance and reliability. For example, performance can be improved by putting subvolumes on different disk drives.

Each subvolume can be organized independently. For example, the log subvolume can be plexed for fault tolerance and the real-time subvolume can be striped across a large number of disks to give maximum throughput for video playback.

Volume elements that are part of a real-time subvolume should not be on the same disk as volume elements used for data or log subvolumes. This is a recommendation for all files on real-time subvolumes and required for files used for guaranteed-rate I/O with hard guarantees. (See “Hardware Configuration Requirements for GRIO” in Chapter 5 for more information.)

You can create subvolumes, but you cannot detach them from their volumes or delete them. A subvolume is automatically deleted when the volume is deleted.

Plexes

A subvolume can contain from one to four *plexes* (sometimes called *mirrors*). Each plex contains a portion or all of the subvolume’s data. By creating a volume with multiple plexes, system reliability is increased.

If there is just one plex in a subvolume, that plex spans the entire address space of the subvolume. However, when there are multiple plexes, individual plexes can have holes in their address spaces as long as the union of all plexes spans the entire address space.

Normally, data is written to all plexes. However, when necessary (for example when an additional plex is added to a subvolume), a full plex copy, called a *plex revive*, is done automatically by the system. See the *xlv_assemble*(1M) and *xlv_plexd*(1M) reference pages for more information.

A plex is composed of one or more volume elements, as shown in Figure 4-4, up to a maximum of 128 volume elements. Each volume element represents a range of addresses within the subvolume.

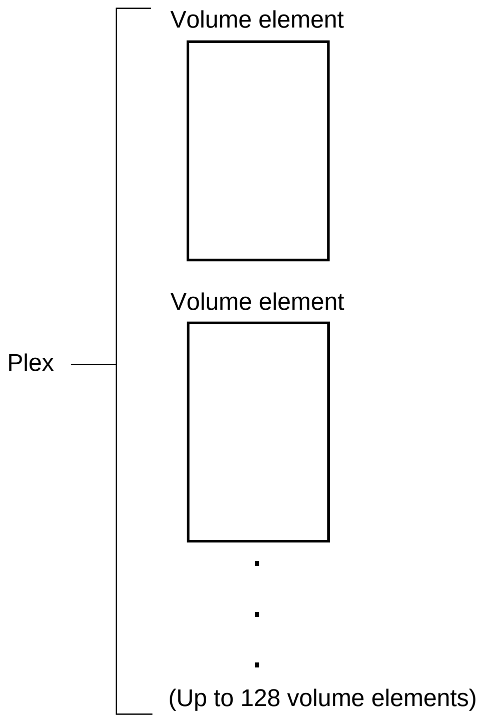

Figure 4-4 Plex Composition

When a plex is composed of two or more volume elements, it is said to have *concatenated* volume elements. With concatenation, data written sequentially to the plex is also written sequentially to the volume elements; the first volume element is filled, then the second, and so on. Concatenation is useful for creating a filesystem that is larger than the size of a single disk.

You can add plexes to subvolumes, detach them from subvolumes that have multiple plexes (and possibly attach them elsewhere), and delete them from subvolumes that have multiple plexes.

Note: To have multiple plexes, you must purchase the Disk Plexing Option software option and obtain and install a NetLS license.

Volume Elements

The simplest type of volume element is a single disk partition. The two other types of volume elements, striped volume elements and multipartition volume elements, are composed of several disk partitions. Figure 4-5 shows a single partition volume element.

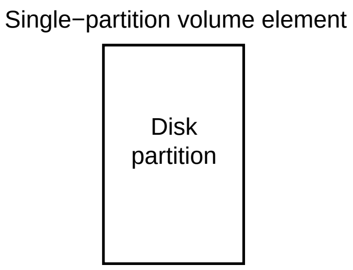

Figure 4-5 Single Partition Volume Element Composition

Figure 4-6 shows a striped volume element. Striped volume elements consist of two or more disk partitions, organized so that an amount of data called the stripe unit is written to each disk partition before writing the next stripe unit-worth of data to the next partition.

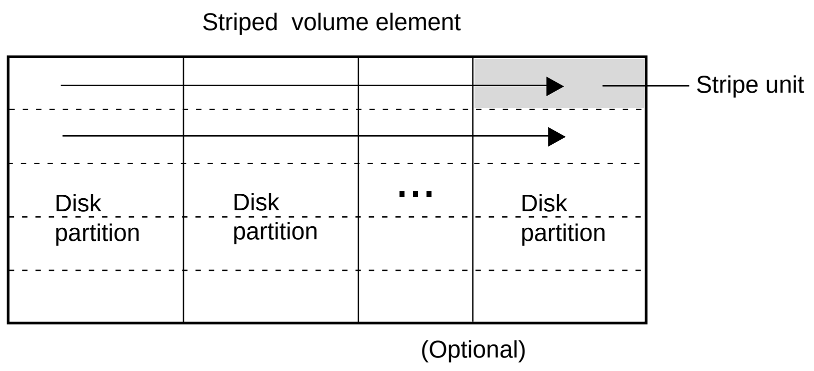

Figure 4-6 Striped Volume Element Composition

Striping can be used to alternate sections of data among multiple disks. This provides a performance advantage by allowing parallel I/O activity.

Figure 4-7 shows a multipartition volume element in which the volume element is composed of more than one disk partition. In this configuration, the disk partitions are addressed sequentially.

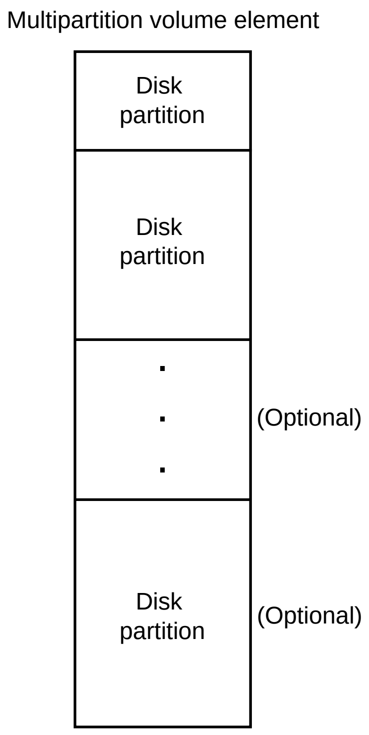

Figure 4-7 Multipartition Volume Element Composition

Any mixture of the three types of volume elements (single partition, striped, and multipartition) can be concatenated in a plex.

Logical Volume Naming

Volumes appear as block and character devices in the */dev* directory. The device names for logical volumes are */dev/dsk/xlv/<volume_name>* and */dev/rdsk/xlv/<volume_name>*, where *<volume_name>* is a volume name specified when the volume is created using *xlv_make*(1M).

When a volume is created on one system and moved (by moving the disks) to another system, the new volume name is the same as the original volume name with the hostname of the original system prepended. For example, if a volume called xlv0 is moved from a system called engrlab1 to a system called engrlab2, the device name of the volume on the new system is */dev/dsk/xlv/engrlab1.xlv0* (the old system name engrlab1 has been prepended to the volume name xlv0).

XLV Daemons

The XLV daemons are:

xlv_labd	*xlv_labd*(1M) writes disk labels. It is started automatically at system startup if it is installed and there are active XLV logical volumes.
xlvd	*xlvd*(1M) handles I/O to plexes and performs plex error recovery. It is created automatically during system startup if plexing software is installed and there are active XLV logical volumes.
xlv_plexd	*xlv_plexd*(1M) is responsible for making all plexes within a subvolume have the same data. It is started automatically at system startup if there are active XLV logical volumes.

XLV does not require an explicit configuration file, nor is it turned on and off with *chkconfig*(1M). XLV is able to assemble logical volumes based solely upon information written in the disk labels. During initialization, the system performs a hardware inventory, reads all the disk labels, and automatically assembles the available disks into volumes.

If some disks are missing, XLV checks to see if there are enough volume elements among the available plexes to map the entire address space. If the whole address space is available, XLV brings the volume online even if some of the plexes are incomplete.

XLV Error Policy

For read failures on log and data subvolumes, XLV rereads from a different plex (when available) and attempts to fix the failed plex by rewriting the results. XLV does not retry on failures for real-time data.

For log and data subvolumes, XLV assumes that the write errors it receives are hard errors (the disk driver and controllers handle soft errors). If the volume element with a hard error is plexed, XLV marks the volume element bad and ignores it. If the volume element is not plexed, the volume element remains associated with the volume and an error is returned.

XLV doesn't handle write errors on real-time subvolumes. Incorrect data is returned without error messages.

Planning a Logical Volume

The following subsections discuss topics to consider when planning a logical volume.

Don't Use XLV When ...

There are some situations where logical volumes cannot be used or are not recommended:

- Swap space cannot be a logical volume.
- Logical volumes aren't recommended on systems with a single disk.
- Striped or concatenated volumes cannot be used for the root filesystem.

Deciding Which Subvolumes to Use

The basic guidelines for choosing which subvolumes to use with EFS filesystems are:

- Only data subvolumes can be used.
- The maximum useful size of a data subvolume (and therefore the volume) on EFS is 8 GB.

The basic guidelines for choosing which subvolumes to use with XFS filesystems are:

- Data subvolumes are required.
- Log subvolumes are optional. If they are not used, log information is put into an *internal log* in the data subvolume (by giving the **–l internal** option to *mkfs*).
- Real-time subvolumes are optional.

Choosing Subvolume Sizes

The basic guidelines for choosing subvolume sizes are:

- The maximum size of a subvolume is one terabyte.
- When real-time subvolumes are used, make a small log subvolume and a small data subvolume. Don’t put much (if any) user data in the filesystem, just real-time data.
- Choosing the size of the log (and therefore the size of the log subvolume) is discussed in the section “Choosing the Log Type and Size” in Chapter 2. Note that if you do not intend to repartition a disk to create an optimal-size log partition, your choice of an available disk partition may determine the size of the log.

Plexing

The basic guidelines for plexing are:

- Use plexing when high reliability and high availability of data are required.
- Plexes can have "holes" in them, portions of the address range not contained by a volume element, as long as at least one of the plexes in the subvolume has a volume element with the address range of the hole.
- The volume elements in each plex of a subvolume must be identical in size with their counterparts in other plexes (volume elements with the same address range). The structure within a volume element (single partition, striped, or multipartition) does not have to match the structure within its counterparts.
- To make volume elements identical in size, you may have to use *dvhtool*(1M). *fx*(1M) partitions in units of cylinders and rounds sizes to megabytes, which may not result in exactly the same number of bytes in two partitions of the "same size" on different types of disks. See the *dvhtool*(1M) reference page for more information.

Striping

The basic guidelines for striping are:

- The root filesystem cannot be striped.
- Striped volume elements must be made of disk partitions that are exactly the same size.

Concatenating Disk Partitions

The basic guidelines for the concatenation of disk partitions are:

- The root filesystem cannot have concatenated disk partitions.
- It is better to concatenate single-partition volume elements into a plex rather than create a single multipartition volume element. This is not for performance reasons, but for reliability. When one disk partition goes bad in a multipartition volume, all disks that contain partitions used in that volume element are taken offline.

Using *xlv_make* to Create Volume Objects

xlv_make(1M) is used to create volumes, subvolumes, plexes, and volume elements from unused disk partitions. It writes only the disk labels; data on the disk partitions is untouched.

After you create a volume, you must make a filesystem on it if necessary and mount the filesystem so that you can use the logical volume. See the section "Preparing a Logical Volume for Use" in this chapter for instructions.

xlv_make can be run interactively or it can take commands from an input file. The remainder of this section gives two examples of using *xlv_make*; the first one is interactive and the second is noninteractive.

Example 1: Simple Logical Volume

This example shows a simple logical volume composed of a data subvolume created from two entire option disks. The disks are on controller 0, units 2 and 3 (use the *hinv*(1M) command, for example *hinv –c disk*, to obtain this information). Partition 7 of each disk is normally the entire disk (use the *prtvtoc*(1M) command, for example *prtvtoc /dev/rdsk/dks0d2vh* and *prtvtoc /dev/rdsk/dks0d3vh*, to obtain this information).

1. Unmount the disks that will be used in the volume if they are mounted. For example:

```
# df
Filesystem                  Type  blocks     use   avail %use  Mounted on
/dev/root                    efs 1939714  430115 1509599  22%  /
/dev/dsk/dks0d2s7            efs 2004550      22 2004528   0%  /d2
/dev/dsk/dks0d3s7            efs 3826812      22 3826790   0%  /d3
# umount /d2
# umount /d3
```

2. Start *xlv_make*:

   ```
   # xlv_make
   xlv_make>
   ```

3. Start creating the volume by specifying its name, for example xlv_volume:

   ```
   xlv_make> vol xlv0
   xlv0
   ```

4. Begin creating the data subvolume:

   ```
   xlv_make> data
   xlv0.data
   ```

   xlv_make echoes the name of each object (volume, subvolume, plex, or volume element) you create.

5. Continue to move down through the hierarchy of the volume by specifying the plex:

   ```
   xlv_make> plex
   xlv0.data.0
   ```

6. Specify the volume elements (disk partitions) to be included in the volume, for example */dev/dsk/dks0d2s7* and */dev/dsk/dks0d3s7*:

   ```
   xlv_make> ve dks0d2s7
   xlv0.data.0.0
   xlv_make> ve dks0d3s7
   xlv0.data.0.1
   ```

   You can specify the last portion of the disk partition pathname (as shown) or the full pathname. *xlv_make* accepts disk partitions that are of types “xlv”, “xfs”, and “efs”. You can use other partition types, for example “lvol”, by giving the **–force** option, for example, *ve –force dks0d2s7*. *xlv_make* automatically changes the partition type to “xlv”.

7. Tell *xlv_make* that you are finished specifying the objects:

   ```
   xlv_make> end
   Object specification completed
   ```

8. Review the objects that you've specified:

   ```
   xlv_make> show

           Completed Objects
   (1)  vol xlv0
   ve xlv0.data.0.0 [empty]
           start=0, end=226799, (cat)grp_size=1
           /dev/dsk/dks0d2s2 (226800 blks)
   ve xlv0.data.0.1 [empty]
           start=226800, end=453599, (cat)grp_size=1
           /dev/dsk/dks0d2s3 (226800 blks)
   ```

9. Write the volume information to the disk labels by exiting *xlv_make*:

   ```
   xlv_make> exit
   Newly created objects will be written to disk.
   Is this what you want?(yes)  yes
   Invoking xlv_assemble
   ```

Example 2: Striped, Plexed Logical Volume

This example shows the noninteractive creation of a logical volume from four equal-sized option disks (controller 0, units 2 through 5). Two plexes will be created with the data striped across the two disks in each plex. The stripe unit will be 128 KB.

1. As in the previous example, unmount the disks to be used if necessary.
2. Create a file, called *xlv0.specs* for example, that contains input for *xlv_make*. For this example and a volume named xlv0, the file contains:

   ```
   vol xlv0
   data
   plex
   ve -stripe -stripe_unit 256 dks0d2s7 dks0d3s7
   plex
   ve -stripe -stripe_unit 256 dks0d4s7 dks0d5s7
   end
   show
   exit
   ```

This script specifies the volume hierarchically: volume, subvolume (data), first plex with a striped volume element, then second plex with a striped volume element. The ve commands have a stripe unit argument of 256. This argument is the number of 512-byte blocks (sectors), so 128K / 512 = 256. The end command signifies that the specification is complete and the (optional) show command causes the specification to be displayed. The disk label is created by the exit command.

3. Run *xlv_make* to create the volume. For example:

   ```
   xlv_make xlv0.specs
   ```

Preparing a Logical Volume for Use

Once you create a logical volume with *xlv_make*, follow these steps to prepare it for use:

1. If there is no filesystem on the logical volume (and you want one) or you want to switch from EFS to XFS, you must create a filesystem with *mkfs*(1M). See Chapter 2, “XFS Filesystem Administration.”
2. Mount the logical volume, for example:

   ```
   mkdir /vol1
   mount /dev/dsk/xlv/xlv0 /vol1
   ```
3. To have the logical volume mounted automatically at system startup, you must add an entry for the volume to */etc/fstab*, for example:

   ```
   /dev/dsk/xlv/xlv0 /vol1 xfs rw,raw=/dev/rdsk/xlv/xlv0 0 0
   ```

   See the *fstab*(4) reference page for more information.

Converting *lv* Logical Volumes to XLV

This section explains the procedure for converting *lv*(7M) logical volumes to XLV logical volumes. The files on the logical volumes are not modified or dumped during the conversion. You must be superuser to perform this procedure.

1. Choose new names for the logical volumes, if desired. XLV, unlike *lv*, only requires names to be valid filenames, so you can choose more meaningful names. For example, you can make the volume names the same as the mount points you use. If you mount logical volumes at */a*, */b*, and */c*, you can name the XLV volumes a, b, and c.
2. Unmount all *lv* logical volumes that you plan to convert to XLV logical volumes. For example:

   ```
   umount /a
   ```
3. Create an input script for *xlv_make* by using *lv_to_xlv*(1M):

   ```
   lv_to_xlv -o scriptfile
   ```

   scriptfile is the name of a temporary file that *lv_to_xlv* creates, for example /usr/tmp/xlv.script. It contains a series of *xlv_make* commands that can be used to create XLV volumes that are equivalent to the *lv* logical volumes listed in */etc/lvtab*.
4. If you want to change the volume names, edit *scriptfile* and replace the names on the lines that begin with `vol` with the new names. For example, change:

   ```
   vol lv0
   ```

   to:

   ```
   vol a
   ```

   The volume name can be any name that is a valid filename.
5. By default, all *lv* logical volumes on the system are converted to XLV. If you do not want all *lv* logical volumes converted to XLV, edit *scriptfile* and remove the *xlv_make* commands for the volumes that you do not want to change. See the section “Using xlv_make to Create Volume Objects” in this chapter and the *xlv_make*(1M) reference page for more information.

6. Create the XLV volumes by running *xlv_make* with *scriptfile* as input:

   `xlv_make` *scriptfile*

7. If you converted all *lv* logical volumes to XLV, remove */etc/lvtab*:

   `rm /etc/lvtab`

8. If you converted just some of the *lv* logical volumes to XLV, edit */etc/lvtab* and remove the entries for the logical volumes you converted.

   `vi /etc/lvtab`

9. Edit */etc/fstab* so that it automatically mounts the XLV logical volumes at startup. These changes to */etc/fstab* are required for each XLV logical volume:

   - In the first field, insert the subdirectory `xlv` after `/dev/dsk`.
   - If you changed the name of the volume, for example from lv0 to a, make the change in the first field.
   - Insert the subdirectory `xlv` into the raw device name.
   - If you changed the name of the volume, for example from lv0 to a, make the change in the raw device.

   For example, if an original line is:

   ```
   /dev/dsk/lv0   /a efs rw,raw=/dev/rdsk/lv0 0 0
   ```

   The changed line, including the name change, is:

   ```
   /dev/dsk/xlv/a /a xfs rw,raw=/dev/rdsk/xlv/a 0 0
   ```

10. Mount the XLV logical volume, for example:

    `mount /a`

Using *xlv_admin* to Administer Logical Volumes

The *xlv_admin*(1M) command is used to modify logical volume objects and their disk labels after they have been created by *xlv_make*. *xlv_admin* is an interactive command with this menu:

```
**************** XLV Administration Menu **********
................ Add Existing Selections...........
1.      Add a ve to an existing plex.
2.      Add a ve at the END of an existing plex.
3.      Add a plex to an existing volume.
................ Detach Selections................
11.     Detach a ve from an existing plex.
12.     Detach a plex from an existing volume.
................ Remove Selections................
21.     Remove a ve from an existing plex.
22.     Remove a plex from an existing volume.
................ Delete Selections................
31.     Delete an object.
32.     Delete all XLV disk labels.
................ Show Selections................
41.     Show object by name and type, only.
42.     Show information for an object.
................ Exit ................
99.     Exit
```

Note: The full menu is shown above; if you do not have a valid license for the Disk Plexing Option software option, several of the plex-related menu selections do not appear.

The following subsections explain how to use *xlv_admin* to perform common operations.

Displaying Logical Volume Objects

To get a list of the highest-level volume objects on a system, use selection 41 of the *xlv_admin* menu, for example:

```
xlv_admin> 41

==================== Listing Objects =============
Volume Element:  'spare_ve'
Volume:          'xlv0'
```

In this example, there are two high-level volume objects, a volume element named spare_ve and a logical volume named xlv0. The volume element is a high-level volume object because it is not part of any plex or subvolume.

To display the complete hierarchy of a high-level volume object, use selection 42 of the *xlv_admin* menu, for example:

```
xlv_admin> 42
Please enter name of object to be operated on.
xlv_admin> xlv0

============= Displaying Requested Object ==========
vol xlv0
ve xlv0.data.0.0 [active]
        start=0, end=226799, (cat)grp_size=1
        /dev/dsk/dks0d2s7 (226800 blks)
ve xlv0.data.0.1 [active]
        start=226800, end=453599, (cat)grp_size=1
        /dev/dsk/dks0d3s7 (226800 blks)
ve xlv0.data.0.2 [active]
        start=453600, end=680399, (cat)grp_size=1
        /dev/dsk/dks0d4s7 (226800 blks)
```

This output shows that xlv0 contains only a data subvolume. The data subvolume has one plex that has three volume elements.

Growing a Logical Volume

Growing a logical volume (increasing its size) can be done in two ways by adding one or more volume elements to the end of one or more of its plexes.

1. If any of the volume elements you plan to add to the volume don't exist yet, create them with *xlv_make*. For example, follow this procedure to create a volume element out of a new disk, */dev/dsk/dks0d4s7*:

   ```
   xlv_make
   xlv_make> ve new_ve dks0d4s7
   new_ve
   xlv_make> end
   Object specification completed
   xlv_make> exit
   Newly created objects will be written to disk.
   Is this what you want?(yes)  yes
   Invoking xlv_assemble
   ```

   The ve command includes a volume element name, new_ve. This is required because the volume element is not part of a larger hierarchy; it is the root object in this case.

2. Use selection 2 of the *xlv_admin* command to add each volume element. For example, to add the volume element from step 1 to plex 0 of the data subvolume of the volume xlv0, use this procedure:

   ```
   xlv_admin
   xlv_admin> 2
   Please enter name of object to be operated on.
   xlv_admin> xlv0.data.0
    Please enter the object you wish to add to the target.
   xlv_admin> new_ve
    Please select choice...
   xlv_admin> 99
   ```

3. If you are growing an XFS filesystem, mount the filesystem if it isn't already mounted:

   mount *volume mountpoint*

   volume is the device name of the logical volume, for example /dev/dsk/xlv/xlv0, and *mountpoint* is the mount point directory for the logical volume.

4. If you are growing an XFS filesystem, use *xfs_growfs*(1M) to grow the filesystem:

   `xfs_growfs -d` *mountpoint*

   mountpoint is the mount point directory for the logical volume.

5. If you are growing an EFS filesystem, unmount the filesystem if it is mounted:

   `umount` *mountpoint*

   mountpoint is the mount point directory for the filesystem.

6. If you are growing an EFS filesystem, use *growfs*(1M) to grow the filesystem:

   `growfs` *volume*

   volume is the device name of the logical volume, for example /dev/dsk/lv0.

Adding a Plex to a Logical Volume

If you have purchased the Disk Plexing Option software option and have installed a NetLS license for it, you can add a plex to an existing subvolume for improved reliability in case of disk failures. The procedure to add a plex to a subvolume is described below. To add more than one plex to a subvolume or to add a plex to each of the subvolumes in a volume, repeat the procedure as necessary.

1. If the plex that you want to add to the subvolume doesn't exist yet, create it with *xlv_make*. For example, to create a plex called plex1 to add to the data subvolume of a volume called root_vol, give these commands:

   ```
   # xlv_make
   xlv_make> show

           Completed Objects
   (1)  vol root_vol
   ve root_vol.data.0.0 [active]
           start=0, end=1992629, (cat)grp_size=1
           /dev/dsk/dks0d1s0 (1992630 blks)

   xlv_make> plex plex1
   ```

```
plex1
xlv_make> ve /dev/dsk/dks0d2s0
plex1.0
xlv_make> end
Object specification completed
xlv_make> exit
Newly created objects will be written to disk.
Is this what you want?(yes)  yes
Invoking xlv_assemble
```

2. Use the *xlv_admin* command menu to add the plex to the volume. For example, to add the standalone plex plex1 to root_vol, use this procedure:

```
# xlv_admin
**************** XLV Administration Menu **********
...
3.      Add a plex to an existing volume.
...
42.     Show information for an object.
...
99.     Exit
...
xlv_admin> 42
Please enter name of object to be operated on.
xlv_admin> root_vol

============= Displaying Requested Object ==========
vol root_vol
ve root_vol.data.0.0 [active]
        start=0, end=1992629, (cat)grp_size=1
        /dev/dsk/dks0d1s0 (1992630 blks)


 Please select choice...
xlv_admin> 42
Please enter name of object to be operated on.
xlv_admin> plex1

============= Displaying Requested Object ==========
plex plex1
ve plex1.0 [empty]
        start=0, end=1992629, (cat)grp_size=1
        /dev/dsk/dks0d2s0 (1992630 blks)
 Please select choice...
xlv_admin> 3
Please enter name of object to be operated on.
xlv_admin> root_vol
 Please enter the object you wish to add to the target.
xlv_admin> plex1
 Please select choice...
```

3. You can confirm that root_vol now has two plexes by using selection 42 of the *xlv_admin* command menu:

   ```
   xlv_admin> 42
   Please enter name of object to be operated on.
   xlv_admin> root_vol

   ============= Displaying Requested Object ==========
   vol root_vol
   ve root_vol.data.0.0 [active]
           start=0, end=1992629, (cat)grp_size=1
           /dev/dsk/dks0d1s0 (1992630 blks)
   ve root_vol.data.1.0 [empty]
           start=0, end=1992629, (cat)grp_size=1
           /dev/dsk/dks0d2s0 (1992630 blks)


    Please select choice...
   ```

   The newly added plex, root_vol.data.1, is in the [empty] state. This is because it is newly created. When a plex is added, *xlv_admin* automatically initiates a plex revive operation to copy the contents of the original plex, root_vol.data.0, to the newly added plex.

4. Exit *xlv_admin*:

   ```
   xlv_admin> 99
   #
   ```

The plex revive completes and the new plex switches to [active] state automatically, but if you want to check its progress and verify that the plex has become active, follow this procedure:

1. List the XLV daemons running, for example:

```
# **ps -ef | grep xlv**
    root    27     1  0 10:49:27 ?        0:00 /sbin/xlv_plexd -m 4
    root    35     1  0 10:49:28 ?        0:00 /sbin/xlv_labd
    root    31     1  0 10:49:27 ?        0:00 xlvd
    root   407    27  1 11:01:01 ?        0:00 xlv_plexd -v 2 -n root_vol.data
-d 50331648 -b 128 -w 0 0 1992629
    root   410   397  2 11:01:11 pts/0    0:00 grep xlv
```

   One instance of *xlv_plexd* is currently reviving root_vol.data. This daemon exits when the plex has been fully revived.

2. Later, check the XLV daemons again, for example:

```
# **ps -ef | grep xlv**
ps -ef | grep xlv
    root    27     1  0 10:49:27 ?        0:00 /sbin/xlv_plexd -m 4
    root    35     1  0 10:49:28 ?        0:00 /sbin/xlv_labd
    root    31     1  0 10:49:27 ?        0:03 xlvd
```

   The instance of *xlv_plexd* that was reviving root_vol.data is no longer running; it has completed the plex revive.

3. Check the state of the plex using *xlv_admin*:

```
# **xlv_admin**
...
xlv_admin> **42**
Please enter name of object to be operated on.
xlv_admin> **root_vol**

============= Displaying Requested Object ==========
vol root_vol
ve root_vol.data.0.0 [active]
        start=0, end=1992629, (cat)grp_size=1
        /dev/dsk/dks0d1s0 (1992630 blks)
ve root_vol.data.1.0 [active]
        start=0, end=1992629, (cat)grp_size=1
        /dev/dsk/dks0d2s0 (1992630 blks)


 Please select choice...
```

Both plexes are now in the [active] state.

4. Exit *xlv_admin*:

```
xlv_admin> 99
#
```

Detaching a Plex from a Volume

Detaching a plex from a volume, perhaps because you want to swap disk drives, can be done while the volume is active. However, the entire address range of the subvolume must still covered by active volume elements in the remaining plex or plexes. *xlv_admin* does not allow you to remove the only active plex in a volume if the other plexes are not yet active. The procedure to detach a plex is:

1. Start *xlv_admin* and display the volume that has the plex that you plan to detach, for example, root_vol:

```
# xlv_admin
...
1.      Add a ve to an existing plex.
...
12.     Detach a plex from an existing volume.
...
42.     Show information for an object.
................ Exit ................
99.     Exit
 Please select choice...
xlv_admin> 42
Please enter name of object to be operated on.
xlv_admin> root_vol

============= Displaying Requested Object ==========
vol root_vol
ve root_vol.data.0.0 [active]
        start=0, end=1992629, (cat)grp_size=1
        /dev/dsk/dks0d1s0 (1992630 blks)
ve root_vol.data.1.0 [active]
        start=0, end=1992629, (cat)grp_size=1
        /dev/dsk/dks0d2s0 (1992630 blks)
```

2. Detach plex 1 and give it the name plex1 by giving these commands:

```
xlv_admin> 12
Please enter name of object to be operated on.
xlv_admin> root_vol
 Please select plex number (0-3).
xlv_admin> 1
Please enter name of new object.
xlv_admin> plex1
 Please select choice...
```

3. To examine the volume and the detached plex, give these commands:

```
xlv_admin> 42
Please enter name of object to be operated on.
xlv_admin> plex1

============= Displaying Requested Object ==========
plex plex1
ve plex1.0 [empty]
        start=0, end=1992629, (cat)grp_size=1
        /dev/dsk/dks0d2s0 (1992630 blks)


 Please select choice...
xlv_admin> 42
Please enter name of object to be operated on.
xlv_admin> root_vol

============= Displaying Requested Object ==========
vol root_vol
ve root_vol.data.0.0 [active]
        start=0, end=1992629, (cat)grp_size=1
        /dev/dsk/dks0d1s0 (1992630 blks)
```

4. Exit *xlv_admin*:

```
xlv_admin> 99
#
```

Deleting an XLV Object

You can delete a volume or any other XLV object by using selection 31 of the *xlv_admin* command menu. The procedure is:

1. If you are deleting a volume, you must unmount it first. For example:

   ```
   umount /vol1
   ```

2. Start *xlv_admin* and list the root of each object hierarchy on the system:

   ```
   # xlv_admin
   ...
   31.     Delete an object.
   ...
   41.     Show object by name and type, only.
   ...
   99.     Exit
    Please select choice...
   xlv_admin> 41

   ==================== Listing Objects =============
   Volume:          'root_vol'
   Plex:            'plex1'
   ```

3. Delete the object, for example the plex plex1:

   ```
   xlv_admin> 31
   Please enter name of object to be operated on.
   xlv_admin> plex1
   ```

4. Confirm that the object is gone:

   ```
   xlv_admin> 41

   ==================== Listing Objects =============
   Volume:          'root_vol'
   ```

5. Exit *xlv_admin*:

   ```
   xlv_admin> 99
   #
   ```

Using the Real-Time Subvolume

Files created on the real-time subvolume of an XLV logical volume are known as real-time files. The next three sections describe the special characteristics of these files.

Files on the Real-Time Subvolume and Utilities

Real-time files have some special characteristics that cause standard IRIX utilities to operate in ways that you might not expect. In particular:

- You cannot create real-time files using any standard utilities. Only specially-written programs can create real-time files. The next section, "Creating Files on the Real-time Subvolume," explains how.
- Real-time files are displayed by *ls*(1), just as any other file. However, there is no way to tell from the *ls* output whether a particular file is on a data subvolume or is a real-time file on a real-time subvolume. Only a specially-written program can determine the type of a file. The F_FSGETXATTR *fcntl*(2) system call is used to determine if a file is a real-time or a standard data file. If the file is a real-time file, the fsx_xflags field of the fsxattr structure has the XFS_XFLAG_REALTIME bit set.
- The *df*(1) utility displays the disk space in the data subvolume by default. When the **–r** option is given, the real-time subvolume's disk space and usage is added. *df* can report that there is free disk space in the filesystem when the real-time subvolume is full, and *df –r* can report that there is free disk space when the data subvolume is full.

Creating Files on the Real-time Subvolume

To create a real-time file, use the F_FSSETXATTR *fcntl*(2) system call with the XFS_XFLAG_REALTIME bit set in the fsx_xflags field of the fsxattr structure. This must be done after the file has first been created / opened for writing, but before any data has been written to the file. Once data has been written to a file, it cannot be changed from a standard data file to a real-time file, nor can files created as real-time files be changed to standard data files.

Real-time files can only be read or written using direct I/O. Therefore, *read*(2) and *write*(2) operations to a real-time file must meet the requirements specified by the F_DIOINFO *fcntl*(2) call. See the *open*(2) reference page for a discussion of the O_DIRECT option to the *open*() system call.

Guaranteed-Rate I/O and the Real-Time Subvolume

The real-time subvolume is used by applications for files that require fixed I/O rates. This feature, called guaranteed-rate I/O, is described in Chapter 5, “Guaranteed-Rate I/O.”

Chapter 5

Guaranteed-Rate I/O

Guaranteed-rate I/O, or GRIO for short, is a mechanism that enables a user application to reserve part of a system's I/O resources for its exclusive use. For example, it can be used to enable "real-time" retrieval and storage of data streams. It manages the system resources among competing applications, so the actions of new processes do not affect the performance of existing ones. GRIO can read and write only files on a real-time subvolume of an XFS filesystem.

This chapter explains important guaranteed-rate I/O concepts, describes how to configure a system for GRIO, and provides instructions for creating an XLV logical volume for use with applications that use GRIO. The main sections in this chapter are:

- "Guaranteed-Rate I/O Overview" on page 92
- "GRIO Guarantee Types" on page 93
- "GRIO System Components" on page 96
- "Hardware Configuration Requirements for GRIO" on page 97
- "Disabling Disk Error Recovery" on page 98
- "Configuring the ggd Daemon" on page 101
- "Example: Setting Up an XLV Logical Volume for GRIO" on page 102
- "GRIO File Formats" on page 106

Note: By default, IRIX supports four GRIO streams (concurrent uses of GRIO). To increase the number of streams to 40, you can purchase the High Performance Guaranteed-Rate I/O—5-40 Streams software option. For more streams, you can purchase the High Performance Guaranteed-Rate I/O—Unlimited Streams software option. See the *grio Release Notes* for information on purchasing these software options and obtaining the required NetLS licenses.

Guaranteed-Rate I/O Overview

The GRIO mechanism is designed for use in an environment where many different processes attempt to access scarce I/O resources simultaneously. GRIO provides a way for applications to determine that resources are already fully utilized and attempts to make further use would have a negative performance impact.

If the system is running a single application that needs access to all the system resources, the GRIO mechanism does not need to be used. Since there is no competition, the application gains nothing by reserving the resources before accessing them.

Applications negotiate with the system to make a GRIO *reservation*, an agreement by the system to provide a portion of the bandwidth of a system resource for a period of time. The only resources supported by GRIO are files residing within a real-time subvolume of an XFS filesystem.

A GRIO reservation is described as the number of bytes per second the application will receive from or transmit to the resource starting at a specific time and continuing for a specific period. The application issues a reservation request to the system, which either accepts or rejects the request. If the reservation is accepted, the application can begin accessing the resource at the reserved time, and it can expect that it will receive the reserved number of bytes per second throughout the time of the reservation. If the system rejects the reservation, it returns the maximum amount of bandwidth that can be reserved for the resource at the specified time. The application can determine if the available bandwidth is sufficient for its needs and issue another reservation request for the lower bandwidth, or it can schedule the reservation for a different time. The GRIO reservation continues until it expires, the file is closed, or an explicit **grio_remove_request**() library call is made (for more information, see the **grio_remove_request**(3X) reference page).

If a process has a rate guarantee on a file, any reference by that process to that file uses the rate guarantee, even if a different file descriptor is used. However, any other process that accesses the same file does so without a guarantee or must obtain its own guarantee. This is true even when the second process has inherited the file descriptor from the process that obtained the guarantee.

Sharing file descriptors between processes in a process group is supported for files used for GRIO, but the processes do not share the guarantee. If a process inherits an open file descriptor from a parent process and wants to have a rate guarantee on the file, the file must be closed and reopened before **grio_request**(3X) is called. If the **sproc**(2) system call is used with the PR_SFDS attribute to keep the open file table synchronized, the automatic removal of rate guarantees on last close of a file is not supported. The rate guarantee is removed when the reservation time expires or the process explicitly calls **grio_remove_request**(3X).

GRIO Guarantee Types

In addition to specifying the amount and duration of the reservation, the application must specify the type of guarantee desired. Each guarantee is a hard guarantee or a soft guarantee. Each guarantee is also a Video on Demand (VOD) guarantee or a non-VOD guarantee. The next few sections describe these types of guarantees and give an example that illustrates the differences between VOD and non-VOD guarantees.

Hard Guarantees

A *hard* guarantee means the system will do everything possible to make sure the application receives the amount of data that has been reserved during each second of the reservation duration.

Hard guarantees are possible only when the disks that are used for the real-time subvolume meet the requirements listed in the section “Hardware Configuration Requirements for GRIO” in this chapter.

Because of these disk configuration requirements, incorrect data can be returned to the application without an error notification, but the I/O requests return within the guaranteed time. If an application requests a hard guarantee and some part of the system configuration makes the granting of a hard guarantee impossible, the reservation is rejected. The application can then issue a reservation request with a soft guarantee.

Soft Guarantees

A *soft* guarantee means the system tries to achieve the desired rate, but there may be circumstances beyond its control that cause it to fail. For example, if a non-real-time disk is on the same SCSI bus as real-time disks and there is a disk data error on the non-real-time disk, the driver retries the request to recover the data. This could cause the rate guarantee on the real-time disk to be missed.

VOD Guarantees

VOD (Video On Demand) is a special type of rate guarantee applied to either hard or soft guarantees. It allows more streams to be supported per disk drive, but requires that the application provide careful control of when and where I/O requests are issued.

VOD guarantees are supported only when using a striped volume. The application must time multiplex the I/O requests to different drives at different times. A process stream can only access a single disk during any one second. Therefore, the stripe unit must be set to the number of kilobytes of data that the application needs to access per second per stream of data. (The stripe unit is set using *xlv_make*(1M) when volume elements are created.) If the process tries to access data on a different disk during a time period, it is suspended until the appropriate time period.

With VOD reservations, if the application does not read the file sequentially, but rather skips around in the file, it will have a performance impact. For example, if disks are four-way striped, it could take as long as four seconds (the size of the volume stripe) for the first I/O request after a seek to complete.

Example: Comparing VOD and Non-VOD

Assume the system has eight disks each supporting twenty-three 64 KB operations per second. For non-VOD GRIO, if an application needs 512 KB of data each second, the eight disks would be arranged in a eight-way stripe. The stripe unit would be 64 KB. Each application read / write operation would be 512 KB and cause concurrent read / write operations on each disk in the stripe. The application could access any part of the file at any time, provided that the read / write operation always started at a stripe boundary. This would provide 23 process streams with 512 KB of data each second.

With a VOD guarantee, the eight drives would be given an optimal I / O size of 512 KB. Each drive can support seven such operations each second. The higher rate (7 x 512 KB versus 23 x 64 KB) is achievable because the larger transfer size does less seeking. Again the drives would be arranged in an eight-way stripe but with a stripe unit of 512 KB. Each drive can support seven 512K streams per second for a total of 8 * 7 = 56 streams. Each of the 56 streams is given a time period. There are eight different time periods with seven different processes in each period. Therefore, 8 * 7 = 56 processes are accessing data in a given time unit. At any given second, the processes in a single time period are only allowed to access a single disk.

Using a VOD guarantee more than doubles the number of streams that can be supported with the same number of disks. The trade off is that the time tolerances are very stringent. Each stream is required to issue the read / write operation within a second. If the process issues the call too late, the request blocks until the next time period for that process on the disk. In this example, this could mean a delay of up to eight seconds. In order to receive the rate guarantee, the application must access the file sequentially. The time periods move sequentially down the stripe allowing each process to access the next 512 KB of the file.

GRIO System Components

Several components make up the GRIO mechanism: a system daemon, support utilities, configuration files, and an application library.

The system daemon is *ggd*(1M). It is started from the script */etc/rc2.d/S94grio* when the system is started. It is always started; unlike some other daemons, it is not turned on and off with *chkconfig*(1M). A lock file is created in the */tmp* directory to prevent two copies of the daemon from running simultaneously. The daemon reads the GRIO configuration files */etc/grio_config* and */etc/grio_disks*.

/etc/grio_config describes the various I/O hardware paths on the system, starting with the system bus and ending with the individual peripherals such as disk and tape drives. It also describes the bandwidth capabilities of each component. The format of this file is described in the section "/etc/grio_config File Format" in this chapter. If you want a soft rate guarantee, you must edit this file. See step 9 in the section "Example: Setting Up an XLV Logical Volume for GRIO" in this chapter for more information.

The utility *cfg*(1M) is used to automatically generate an */etc/grio_config* configuration file for a system's configuration. A checksum is appended to the end of the file by *cfg*. When the *ggd* daemon reads the configuration information, it validates the checksum. You can edit */etc/grio_config* to tune the performance characteristics to fit a given application. See the next section, "Configuring the ggd Daemon," for more information.

/etc/grio_disks describes the performance characteristics for the types of disk drives that may be found on the system. You can edit the file to add support for new drive types. The format of this file is described in the section "/etc/grio_disks File Format" in this chapter.

The library */usr/lib/libgrio.so* contains a collection of routines that enable an application to establish a GRIO session. The library routines are the only way in which an application program can communicate with the *ggd* daemon.

Hardware Configuration Requirements for GRIO

Guaranteed-rate I/O requires the hardware to be configured so that it follows these guidelines:

- Put only real-time subvolume volume elements on a single disk (not log or data subvolume volume elements). This configuration is recommended for soft guarantees and required for hard guarantees.
- Only SCSI disks can be used for real-time subvolumes. IPI, ESDI, and other non-SCSI disks cannot be used.
- For GRIO with hard guarantees, each disk used for hard guarantees must be on a controller whose disks are used exclusively for real-time subvolumes. These controllers cannot have any devices other than SCSI disks on their buses. Any other devices could prevent the disk from accessing the SCSI bus in a timely manner and cause the rate to be missed.
- The drive firmware in each disk used in the real-time subvolume must have the predictive failure analysis and thermal recalibration features disabled. All disk drives have been shipped from Silicon Graphics this way since March 1994.
- For hard guarantees, the disk drive retry and error correction mechanisms must be disabled for all disks used in the real-time subvolume. See the section “Disabling Disk Error Recovery” in this chapter for more information.
- When possible, disks used in the real-time subvolume of an XLV volume should have the RC (read continuous) bit enabled. This allows the disks to perform faster, but at the penalty of occasionally returning incorrect data (without giving an error). Enabling the RC bit is part of the procedure described in the section “Disabling Disk Error Recovery.”
- Disks used in the data and log subvolumes of the XLV logical volume must not have their retry mechanisms disabled. The data and log subvolumes contain information critical to the filesystem and cannot afford an occasional disk error.

Disabling Disk Error Recovery

SCSI disks in XLV logical volumes used by GRIO applications that require hard guarantees must have their parameters modified to prevent the disk from performing automatic error recovery. When the drive does error recovery, its performance degrades and there can be lengthy delays in completing I/O requests. When the drive error recovery mechanisms are disabled, occasionally invalid data is returned to the user without an error indication. Because of this, the integrity of data stored on an XLV real-time subvolume is not guaranteed.

The *fx*(1M) utility is used in expert mode to set the drive parameters for real-time operation. Table 5-1 shows the disk drive parameters that must be changed for GRIO.

Table 5-1 Disk Drive Parameters for GRIO

Parameter	New Setting
Auto bad block reallocation (read)	Disabled
Auto bad block reallocation (write)	Disabled
Delay for error recovery (disabling this parameter enables the read continuous (RC) bit)	Disabled

Caution: Setting disk drive parameters must be performed correctly on approved disk drive types only. Performing the procedure incorrectly, or performing on an unapproved type of disk drive could severely damage the disk drive. Setting disk drive parameters should be performed only by experienced system administrators.

fx reports the disk drive type after the controller test on a line that begins with `Scsi drive type`. The approved disk drives types whose parameters can be set for real-time operation are:

- SGI 0664N1D 6s61
- SGI 0664N1D 4I4I

The procedure for setting disk drive parameters is shown in the example below. It uses the parameters shown in Table 5-1 for a disk drive on controller 131, unit 1.

```
fx -x
fx version 5.3, Nov 18, 1994
fx: "device-name" = (dksc) <Enter>
fx: ctlr# = (0) 131
fx: drive# = (1) 1
fx: lun# = (0)
...opening dksc(131,1,0)


...controller test...OK
Scsi drive type == SGI     0664N1D         6s61
----- please choose one (? for help, .. to quit this menu)-----
[exi]t               [d]ebug/             [l]abel/
[b]adblock/          [exe]rcise/          [r]epartition/
fx > label

----- please choose one (? for help, .. to quit this menu)-----
[sh]ow/         [sy]nc          [se]t/          [c]reate/
fx/label> show

----- please choose one (? for help, .. to quit this menu)-----
[para]meters       [part]itions       [b]ootinfo         [a]ll
[g]eometry         [s]giinfo          [d]irectory
fx/label/show> parameters

----- current drive parameters-----
Error correction enabled          Enable data transfer on error
Don't report recovered errors     Do delay for error recovery
Don't transfer bad blocks         Error retry attempts          10
Do auto bad block reallocation (read)
Do auto bad block reallocation (write)
Drive readahead  enabled          Drive buffered writes disabled
Drive disable prefetch   65535    Drive minimum prefetch         0
Drive maximum prefetch   65535    Drive prefetch ceiling     65535
Number of cache segments     4
Read buffer ratio        0/256    Write buffer ratio         0/256
Command Tag Queueing disabled


----- please choose one (? for help, .. to quit this menu)-----
[para]meters       [part]itions       [b]ootinfo         [a]ll
[g]eometry         [s]giinfo          [d]irectory
fx/label/show> ..

----- please choose one (? for help, .. to quit this menu)-----
```

```
[sh]ow/         [sy]nc          [se]t/          [c]reate/
fx/label> **set**

----- please choose one (? for help, .. to quit this menu)-----
[para]meters              [part]itions              [s]giinfo
[g]eometry                [m]anufacturer_params     [b]ootinfo
fx/label/set> **parameters**
fx/label/set/parameters: Error correction = (enabled) **<Enter>**
fx/label/set/parameters: Data transfer on error = (enabled) **<Enter>**
fx/label/set/parameters: Report recovered errors = (disabled) **<Enter>**
fx/label/set/parameters: Delay for error recovery = (enabled) **disable**
fx/label/set/parameters: Err retry count = (10) **<Enter>**
fx/label/set/parameters: Transfer of bad data blocks = (disabled) **<Enter>**
fx/label/set/parameters: Auto bad block reallocation (write) = (enabled) **disable**
fx/label/set/parameters: Auto bad block reallocation (read) = (enabled) **disable**
fx/label/set/parameters: Read ahead caching = (enabled) **<Enter>**
fx/label/set/parameters: Write buffering = (disabled) **<Enter>**
fx/label/set/parameters: Drive disable prefetch = (65535) **<Enter>**
fx/label/set/parameters: Drive minimum prefetch = (0) **<Enter>**
fx/label/set/parameters: Drive maximum prefetch = (65535) **<Enter>**
fx/label/set/parameters: Drive prefetch ceiling = (65535) **<Enter>**
fx/label/set/parameters: Number of cache segments = (4) **<Enter>**
fx/label/set/parameters: Enable CTQ = (disabled) **<Enter>**
fx/label/set/parameters: Read buffer ratio = (0/256) **<Enter>**
fx/label/set/parameters: Write buffer ratio = (0/256) **<Enter>**
 * * * * * W A R N I N G * * * * *
about to modify drive parameters on disk dksc(131,1,0)! ok? **yes**

----- please choose one (? for help, .. to quit this menu)-----
[para]meters       [part]itions       [b]ootinfo         [a]ll
[g]eometry         [s]giinfo          [d]irectory
fx/label/set> **..**

----- please choose one (? for help, .. to quit this menu)-----
[sh]ow/         [sy]nc          [se]t/          [c]reate/
fx/label> **..**

----- please choose one (? for help, .. to quit this menu)-----
[exi]t             [d]ebug/           [l]abel/           [a]uto
[b]adblock/        [exe]rcise/        [r]epartition/     [f]ormat
fx> **exit**
label info has changed for disk dksc(131,1,0).  write out changes? (yes) **<Enter>**
```

Configuring the *ggd* Daemon

The files */etc/grio_disks*, */etc/grio_config,* and */etc/config/ggd.options* can be modified as described below to configure and tune the *ggd* daemon. After any of these files have been modified, *ggd* must be restarted. Give these commands to restart *ggd*:

```
/etc/init.d/grio stop
/etc/init.d/grio start
```

When *ggd* is restarted, current rate guarantees are lost.

Some ways to configure and tune *ggd* are:

- You can edit */etc/grio_config* to tune the performance characteristics to fit a given application. See the section "/etc/grio_config File Format" for information about the format of this file. *ggd* must then be started with the **–d c** option, so the file checksum is not used. This is done by creating or editing the file */etc/config/ggd.options* and adding **–d c**.
- Run *ggd* as a real-time process. If the system has more than one CPU and you are willing to dedicate an entire CPU to performing GRIO requests, add the **–c** *cpunum* to the file */etc/config/ggd.options*. This causes the CPU to be marked isolated, restricted to running selected processes, and nonpreemptive. After *ggd* has been restarted, you can confirm that the CPU has been marked by giving this command (*cpunum* is 3 in this example):

  ```
  mpadmin -s
  processors: 0 1 2 3 4 5 6 7
  unrestricted: 0 1 2 5 6 7
  isolated: 3
  restricted: 3
  preemptive: 0 1 2 4 5 6 7
  clock: 0
  fast clock: 0
  ```

  Processes using GRIO should mark their processes as real-time and runable only on CPU *cpunum*. The **sysmp**(2) reference page explains how to do this.

To mark an additional CPU for real-time processes after *ggd* has been restarted, give these commands:

```
mpadmin -rcpunum2
mpadmin -Icpunum2
mpadmin -Ccpunum2
```

Example: Setting Up an XLV Logical Volume for GRIO

This section gives an example of configuring a system for GRIO as described in previous sections: creating an XLV logical volume with a real-time subvolume, making a filesystem on the volume and mount it, and configuring and restarting the *ggd* daemon. It assumes that the disk partitions have been chosen following the guidelines in the section “Hardware Configuration Requirements for GRIO” and that the disk drive parameters have already been modified as described in the section “Disabling Disk Error Recovery.”

1. Determine the values of variables that will be used while constructing the XLV logical volume:

vol_name	The name of the volume with a real-time subvolume.
rate	The rate at which applications using this volume will access the data. *rate* is the number of bytes per second per stream (the rate) divided by 1K. This information may be available in published information about the applications or from the developers of the applications. Remember that the GRIO system allows each stream to issue only one read/write request each second. The stream must obtain all the data it needs in one second from a single read call.
num_disks	The number of disks that will be included in the real-time subvolume of the volume.

stripe_unit	When the real-time disks are striped (required for Video on Demand and recommended otherwise), this is the amount of data written to one disk before writing to the next. It is expressed in 512-byte sectors.

For non-VOD guarantees:

stripe_unit `=` *rate* `* 1K /` (*num_disks* `* 512)`

For VOD guarantees:

stripe_unit `=` *rate* `* 1K / 512`

extent_size	The filesystem extent size.

For non-VOD guarantees:

extent_size `=` *rate* `* 1K`

For VOD guarantees:

extent_size `=` *rate* `* 1K *` *num_disks*

opt_IO_size	The optimal I/O size.

For non-VOD guarantees, it should be an even factor of *stripe_unit*, but not less than 64.

For VOD guarantees:

opt_IO_size `=` *rate*

Table 5-2 gives examples for the values of these variables.

Table 5-2 Examples of Values of Variables Used in Constructing an XLV Logical Volume Used for GRIO

Variable	Type of Guarantee	Comment	Example Value
vol_name	any	This name matches the last component of the device name for the volume, /dev/dsk/xlv/*vol_name*	xlv_grio
rate	any	For this example, assume 512 KB per second per stream	512
num_disks	any	For this example, assume 4 disks	4
stripe_unit	hard or soft	512*1K/(4*512)	256

Table 5-2 (continued) Examples of Values of Variables Used in Constructing an XLV Logical Volume Used for GRIO

Variable	Type of Guarantee	Comment	Example Value
	VOD hard or soft	512*1K/512	1024
extent_size	hard or soft	512 * 1K	512k
	VOD hard or soft	512 * 1K * 4	2048k
opt_IO_size	hard or soft	128/1 = 128 or 128/2 = 64 are possible	64
	VOD hard or soft	Same as *rate*	512

2. Create an *xlv_make*(1M) script file that creates the XLV logical volume. (See the section "Using xlv_make to Create Volume Objects" in Chapter 4 for more information.) Example 5-1 shows an example script file for a volume.

Example 5-1 Configuration File for a Volume Used for GRIO

```
# Configuration file for logical volume vol_name. In this
# example, data and log subvolumes are partitions 0 and 1 of
# the disk at unit 1 of controller 1. The real-time
# subvolume is partition 0 of the disks at units 1-4 of
# controller 2.
#
vol vol_name
data
plex
ve dks1d1s0
log
plex
ve dks1d1s1
rt
plex
ve -stripe -stripe_unit stripe_unit dks2d1s0 dks2d2s0 dks2d3s0
dks2d4s0
show
end
exit
```

3. Run *xlv_make* to create the volume:

   ```
   xlv_make script_file
   ```

   script_file is the *xlv_make* script file you created in step 2.

4. Create the filesystem by giving this command:

   ```
   mkfs -r extsize=extent_size /dev/dsk/xlv/vol_name
   ```

5. To mount the filesystem immediately, give these commands:

   ```
   mkdir mountdir
   mount /dev/dsk/xlv/vol_name mountdir
   ```

   mountdir is the full pathname of the directory that is the mount point for the filesystem.

6. To configure the system so that the new filesystem is automatically mounted when the system is booted, add this line to */etc/fstab*:

   ```
   /dev/dsk/xlv/vol_name mountdir xfs
   rw,raw=/dev/rdsk/xlv/vol_name 0 0
   ```

7. If the file */etc/grio_config* exists, and you see `OPTSZ=65536` for each device, skip to step 9.

8. Create the file */etc/grio_config* with this command:

   ```
   cfg -d opt_IO_size
   ```

9. If you want soft rate guarantees, edit */etc/grio_config* and remove this string:

   ```
   RT=1
   ```

   from the lines for disks where software retry is required (see the section "/etc/grio_config File Format" in this chapter for more information).

10. Restart the *ggd* daemon:

    ```
    /etc/init.d/grio stop
    /etc/init.d/grio start
    ```

    Now the user application can be started. Files created on the real-time subvolume volume can be accessed using guaranteed-rate I/O.

GRIO File Formats

The following subsections contain reference information about the contents of the three GRIO configuration files, */etc/grio_config*, */etc/grio_disks*, and */etc/config/ggd.options*.

/etc/grio_config File Format

The */etc/grio_config* file describes the configuration of the system I/O devices. The information in this file is used by the *ggd* daemon to construct a tree that describes the relationships between the components of the I/O system and their bandwidths. In order to grant a rate guarantee on a disk device, the *ggd* daemon checks that each component in the I/O path from the system bus to the disk device has sufficient available bandwidth.

There are two basic types of records in */etc/grio_config*: component records and relationship records. Each record occupies a single line in the file. Component records describe the I/O attributes for a single component in the I/O subsystem. CPU and memory components are described in the file, as well, but do not currently affect the granting or refusal of a rate guarantee.

The format of component records is:

componentname= *parameter*=*value* *parameter*=*value* . . . (*descriptive text*)

componentname is a text string that identifies a single piece of hardware present in the system. Some *componentname*s are:

SYSTEM	The machine itself. There is always one SYSTEM component.
CPU*n*	A CPU board in slot *n*. It is attached to SYSTEM.
MEM*n*	A memory board in slot *n*. It is attached to SYSTEM.
IOB*n*	An I/O board with *n* as its internal location identifier. It is attached to SYSTEM.
IOA*nm*	An I/O adaptor. It is attached to IOB*n* at location *m*.
CTR*n*	SCSI controller number *n*. It is attached to an I/O adaptor.
DSK*n*U*m*	Disk device *m* attached to SCSI controller *n*.

parameter can be one of the following:

OPTSZ	The optimal I/O size of the component
NUM	The number of OPTSZ I/O requests supported by the component each second
SLOT	The backplane slot number where the component is located, if applicable (not used on all systems)
VER	The CPU type of system (for example, IP22, IP19, and so on; not used on all systems)
NUMCPUS	The number of CPUs attached to the component (valid only for CPU components; not used on all systems)
MHZ	The MHz value of the CPU (valid only for CPU components; not used on all systems)
CTLRNUM	The SCSI controller number of the component (valid only for SCSI devices)
UNIT	The SCSI unit number of the component (valid only for SCSI devices)
RT	Set to 1 if the disk is in a real-time subvolume (remove this parameter for soft guarantees)

The *value* is the integer or text string value assigned to the parameter. The string enclosed in parentheses at the end of the line describes the component.

Some examples of component records taken from */etc/grio_config* on an Indy™ system are shown below. Each record is a single line, even if it is shown on multiple lines here.

- `SYSTEM= OPTSZ=65536 NUM=5000 (IP22)`

  The *componentname* SYSTEM refers to the system bus. It supports five thousand 64 KB operations per second.

- `CPU= OPTSZ=65536 NUM=5000 SLOT= 0 VER=IP22 NUMCPUS=1 MHZ=100`

  This describes a 100 MHz CPU board in slot 0. It supports five thousand 64 KB operations per second.

- `CTR0= OPTSZ=65536 NUM=100 CTLRNUM=0 (WD33C93B,D)`

  This describes SCSI controller 0. It supports one hundred 64 KB operations per second.

- `DSK0U0= OPTSZ=65536 NUM=23 CTLRNUM=0 UNIT=1 (SGI SEAGATE ST31200N9278)`

  This describes a SCSI disk attached to SCSI controller 0 at SCSI unit 1. It supports twenty three 64 KB operations per second.

Relationship records describe the relationships between the components in the I/O system. The format of relationship records is:

component`:` *attached_component1* *attached_component2* `...`

These records indicate that if a guarantee is requested on *attached_component1*, the *ggd* daemon must determine if *component* also has the necessary bandwidth available. This is performed recursively until the SYSTEM component is reached.

Some examples of relationship records taken from */etc/grio_config* on an Indy system are:

- `SYSTEM: CPU`

  This describes the CPU board as being attached to the system bus.

- `CTR0: DSK0U1`

  This describes SCSI disk 1 being attached to SCSI controller 0.

/etc/grio_disks File Format

The file */etc/grio_disks* contains information that describes I/O bandwidth parameters of the various types of disk drives that can be used on the system. The *ggd* daemon and *cfg* contain built-in knowledge for the disks supported by Silicon Graphics for optimal I/O sizes of 64K, 128K, 256K, and 512K. To add additional disks or to specify a different optimal I/O size, you must add additional information to the */etc/grio_disks* file.

The records in */etc/grio_disks* are of the form:

`ADD "`*disk id string*`"` *optimal_iosize number_optio_per_second*

The first field is always the keyword ADD. The next field is a 28-character string that is the drive manufacturer's disk ID string. The next field is an integer denoting the optimal I/O size of the device in bytes. The last field is an integer denoting the number of optimal I/O size requests that the disk can satisfy in one second.

Some examples of these records are:

- `ADD     “SGI     SEAGATE ST31200N9278”  64K     23`
- `ADD     “SGI             0064N1D 4I4I”  64K     23`

Both of these disk drives support twenty-three 64 KB requests per second.

/etc/config/ggd.options File Format

/etc/config/ggd.options contains command-line options for the *ggd* daemon. Options you might include in this file are:

–d c	Do not use the checksum at the end of */etc/grio_config*. This is option is required when */etc/grio_config* has been modified to tune performance for an application.
–c *cpunum*	Dedicate CPU *cpunum* to performing GRIO requests exclusively.

If you change this file, you must restart *ggd* to have your changes take effect. See the section “Configuring the ggd Daemon” in this chapter for more information.

Appendix A

Error Messages

This appendix explains some of the error messages that can occur while performing the procedures in this guide.

Error Messages While Converting From EFS to XFS

This section contains error messages that can occur during the conversion of a filesystem from EFS to XFS or when the system is rebooted after a conversion.

mountdir: `Device busy`

If *umount* reports that the root filesystem or any other filesystem is busy when you try to unmount it while in the miniroot, return to the `Inst>` prompt and follow this procedure to force *inst* to release files in the busy filesystem so that it can be unmounted:

```
Inst> quit
Building dynamic ELF inventory file for rqs(1) processing .. 100% Done.
Invoking rqs(1) on necessary dynamic ELF objects .. 100% Done.
Automatically reconfiguring the operating system.
Ready to restart the system.  Restart? { (y)es, (n)o, (sh)ell, (h)elp }: no
Reinvoking software installation.
...
Inst Main Menu
...
Inst> admin

Administrative Commands Menu
...
Admin> umount -b /,/proc
Re-initializing installation history database
Reading installation history .. 100% Done.
Checking dependencies .. 100% Done.
```

`mount:` *device* `on` *mountdir*`: Invalid argument`

This message for a wide variety of problems. For example, this error message occurs if you try to mount a device that doesn't contain a valid filesystem.

```
WARNING: initial mount of root device 0x2000010 failed with
errno 22
PANIC: vfs_mountroot: no root found
```

This panic at system startup is caused by a bad kernel. Some possible causes are:

- *eoe2.sw.xfs* was not installed, but the root filesystem is an XFS filesystem.
- *eoe2.sw.efs* was not installed, but the root filesystem is an EFS filesystem.
- Conversion of a system disk with separate root and usr partitions from EFS to XFS was not performed correctly; */var* wasn't linked to */usr/var*, so kernel object files weren't found in */var/sysgen* when the kernel was autoconfigured.

`XFS dev 0x`*nnnnnnnn* `read error in file system meta-data.`

`XFS dev 0x`*nnnnnnnn* `write error in file system meta-data.`

These panics are caused by disk errors in filesystem metadata. 0x*nnnnnnnn* is the hexadecimal representation of the device that returned the error.

After getting this type of panic, you should:

- Reboot the system and check the filesystems with *xfs_check*(1M) (see the section "Checking Filesystem Consistency" in Chapter 2).
- Replace the disk that gives the errors if panics continue.

General Error Messages

```
NOTICE: Start mounting filesystem: /
NOTICE: Ending xFS recovery for filesystem: / (dev: 128/16)

NOTICE: Start mounting filesystem: /vol1
NOTICE: Ending clean xFS mount for filesystem: /vol1
```

These messages, which occur during system startup, are normal and do not indicate that any error or problem has occurred.

/mountdir`: Filesystem too large for device`

This message is the result of mounting a disk partition that doesn't have an XFS filesystem on it, but it overlaps or has the same starting point as a disk partition that does have an XFS filesystem on it. For example, you see this error message if you make a filesystem on */dev/dsk/dks0d2s7*, and then mount */dev/dsk/dks0d2s0*.

`mount:` *device* `on` *mountdir*`: Invalid argument`

This message for a wide variety of problems. For example, this error message occurs if you try to mount a device that doesn't contain a valid filesystem.

Error Messages from *xlv_make*

The *xlv_make*(1M) reference page provides a complete listing of error messages from *xlv_make* and their causes. The *xlv_make*(1M) reference page is included in Appendix B in the printed version of this guide.

Error Messages from *xfs_check*

The *xfs_check*(1M) reference page provides a listing of error messages from *xfs_check* and describes them. The *xfs_check*(1M) reference page is included in Appendix B in the printed version of this guide.

Appendix B

Reference Pages

This appendix lists reference pages (man pages) that provide information about topics that relate to XFS and XLV. The printed form of this guide includes copies of the reference pages for key utilities and file formats.

All reference pages can be viewed online using the *man*(1) command. On systems with graphics, they can also be viewed using the *xman*(1) command or the “Man Pages” item on the Help toolchest.

XFS, XLV, and GRIO Reference Pages

Table B-1 lists reference pages that contain information related to XFS, XLV, and GRIO. For each category of reference pages, the table lists the subsystem that includes these reference pages.

Table B-1 Related Reference Pages

Category	Reference Pages	Subsystem
General information	grio(5), xfs(4), xlv(7M)	eoe2.man.xfs, eoe2.man.xlv
XFS utilities	mkfs_xfs(1M), xfs_bmap(1M), xfs_check(1M), xfs_estimate(1M), xfs_growfs(1M), xfs_logprint(1M), xfsdump(1M), xfsrestore(1M)	eoe2.man.xfs
XLV utilities	lv_to_xlv(1M), xlv_assemble(1M), xlv_make(1M), xlv_set_primary(1M), xlv_shutdown(1M)	eoe2.man.xlv
XLV daemons	xlv_labd(1M), xlv_plexd(1M), xlvd(1M)	eoe2.man.xlv

Table B-1 (continued) Related Reference Pages

Category	Reference Pages	Subsystem
GRIO utility and daemon	cfg(1M), ggd(1M)	eoe2.man.xfs
GRIO library routines	grio_get_rtgkey(3X), grio_remove_request(3X), grio_request(3X), grio_use_rtgkey(3X)	dev.man.irix_lib
GRIO file formats	grio_config(4), grio_disks(4)	eoe2.man.xfs
Standard utilities and files that have been modified for use with XFS and XLV	Add_disk(1), bru(1), cpio(1), df(1), find(1), fx(1M), mkfs(1M), mount(1M), od(1), tar(1), fstab(4)	eoe1.man.unix
System calls that are new or have been extended for use with XFS	fcntl(2), fstat64(2), ftruncate64(2), getrlimit64(2), lseek64(2), lstat64(2), mmap64(2), mount(2), setrlimit64(2), stat64(2), syssgi(2), truncate64(2)	dev.man.irix_lib
New library routines	aio_cancel64(3), aio_error64(3), aio_read64(3), aio_return64(3), aio_sgi_init(3), aio_sgi_init64(3), aio_suspend64(3), aio_write64(3), lio_listio64(3), fd_to_handle(3X), fgetpos64(3S), free_handle(3X), fseek64(3S), fsetpos64(3S), ftell64(3S), ftw64(3C), handle_to_fshandle(3X), nftw64(3C), open_by_handle(3X), path_to_fshandle(3X), path_to_handle(3X), readlink_by_handle(3X)	dev.man.irix_lib
New data structure	stat64(5)	dev.man.irix_lib

Reference Pages in This Guide

The printed form of this guide includes the reference pages listed below. The IRIS InSight form of this guide doesn't include these reference pages, but they can be viewed online using *man*(1), *xman*(1), or Man Pages on the Help toolchest as described above.

NAME

cfg – generates guaranteed-rate I/O device configuration file

SYNOPSIS

cfg [–**d** optiosize] [–**f** file]

DESCRIPTION

cfg scans the hardware available on the system and creates a file that describes the rates that can be guaranteed on each I/O device. The output file is *file* if the –**f** option is specified, otherwise */etc/grio_config* is used. 64 kilobytes is used as the optimal i/o size unless the –**d** option is used.

The *cfg* utility appends a checksum to the end of the file so that *ggd* can determine if the file has been edited.

NOTES

If the –**d** option is used to change the optimal I/O size, the */etc/grio_disks* file must be edited to indicate the number of requests supported per second for the given optimal I/O size.

FILES

/etc/grio_config
/etc/grio_disks

SEE ALSO

ggd(1M), grio_config(4), grio_disks(4)

NAME

dump, rdump – incremental file system dump

SYNOPSIS

/sbin/dump [key [*argument* ...]] filesystem
/sbin/rdump [key [*argument* ...]] filesystem

DESCRIPTION

dump backs up all files in *filesystem*, or files changed after a certain date to magnetic tape or files. The *key* specifies the date and other options about the dump. *Key* consists of characters from the set **0123456789fusCcdbWwn.** Any arguments supplied for specific options are given as subsequent words on the command line, in the same order as that of the options listed.

If no key is given, the *key* is assumed to be **9u** and the *filesystem* specified is dumped to the default tape device */dev/tape*.

0–**9** This number is the 'dump level'. All files modified since the last date stored in the file */etc/dumpdates* for the same *filesystem* at lesser levels will be dumped. If no date is determined by the level, the beginning of time is assumed; thus the option **0** causes the entire filesystem to be dumped. For instance, if you did a ''level 2'' dump on Monday, followed by a ''level 4'' dump on Tuesday, a subsequent ''level 3'' dump on Wednesday would contain all files modified or added to the *filesystem* since the ''level 2'' (Monday) backup. A ''level 0'' dump copies the entire filesystem to the dump volume.

f Place the dump on the next *argument* file instead of the default tape device */dev/tape.* If the name of the file is ''–'', *dump* writes to standard output. If the name of the file is of the format *machine:device* the *filesystem* is dumped across the network to the remote machine. Since **dump** is normally run by root, the name of the local machine must appear in the *.rhosts* file of the remote machine. If the file name *argument* is of the form *user@machine:device,* **dump** will attempt to execute as the specified user on the remote machine. The specified user must have a *.rhosts* file on the remote machine that allows root from the local machine. **dump** creates a remote server, */etc/rmt,* on the client machine to access the tape device.

u If the **dump** completes successfully, write the date of the beginning of the dump on file */etc/dumpdates.* This file records a separate date for each filesystem and each dump level. The format of */etc/dumpdates* is readable by people, consisting of one free format record per line: filesystem name, increment level and *ctime(3C)* format dump date. */etc/dumpdates* may be edited to change any of the fields, if necessary.

s The size of the dump tape is specified in feet. The number of feet is taken from the next *argument.* When the specified size is reached, **dump** will prompt the operator and wait for the reel / volume to be changed. The default tape size for the standard 9 track half inch reels is 2400 feet. The default for cartridge tapes is an effective tape length of 5400 feet, and this assumes a 9-track QIC-24 tape whose physical tape length is 600 feet. See note on cartridge tapes parameters below.

d The density of the tape, expressed in BPI (bytes per inch), is taken from the next *argument*. This is used in calculating the amount of tape used per reel. The default is 1600 BPI, except for the cartridge tape which has a default density of 1000 BPI. Unless a higher density is specified explicitly, **dump** uses its default density - even if the tape drive is capable of higher-density operation (for instance 6250 BPI). If the density specified does not correspond to the density of the tape device being used, **dump** will not be able to handle end-of-tape properly.

b The blocking factor (number of 1 Kbyte blocks written out together) is taken from the next *argument*. The default is 10. The default blocking factor for tapes of density 6250 BPI and greater is 32. If values larger than 32 are used, **restore** will not correctly determine the block size unless the **b** option is also used. To maximize tape utilization, use a blocking factor which is a multiple of 8. For most types of supported tape drives, the greatest capacity and tape throughput is obtained using a blocking factor of 128 or even larger; note that **restore***(1m)* will only automatically determine the blocking factor if it is 32 or less.

C This specifies the total tape capacity in 1K blocks, overriding the **c**, **s**, and **d** arguments if they are also given. No adjustment is made for possible inter-record gaps, or lost capacity due to stop / start repositioning, so it isn't necessary to guess how the dump algorithm for these factors will affect the parameters. Since they aren't taken into account, and there may also be lost capacity due to retries on media errors (by the drive), one should be conservative when specifying capacity.

The *argument* is parsed with **strtoul***(3)*, so it may be in any base (e.g., a 0x prefix specifies a hex value, a 0 prefix specifies octal, no prefix is decimal). The argument may have a *k*, *K*, *m*, or *M* suffix. The first two multiply the value by 1024, the 3rd and 4th multiple by 1048576, so a tape with a 2.2 Gbyte capacity might be specified as **C 2m** allowing 10% loss to retries, etc.

c Indicates that the tape is a cartridge tape instead of the standard default half-inch reel. This should always be specified when using cartridge tapes. The values for blocking factor, size and density are taken to be 10 (1 KByte blocks), 5400 feet and 1000 BPI respectively unless overridden with the 'b', 's' or 'd' option. Cartridge tapes with multiple tracks have a greater effective length which can be specified with the 's' option.

W **dump** tells the operator what file systems need to be dumped. This information is gleaned from the files */etc/dumpdates* and */etc/fstab*. The **W** option causes **dump** to print out, for each file system in */etc/dumpdates* the most recent dump date and level, and highlights those file systems that should be dumped. The *mnt_freq* field in the */etc/fstab* entry of the file system must be non-zero for **dump** to determine whether the file system should be dumped or not. If the **W** option is set, no other option must be given, and **dump** exits immediately.

w Is like W, but prints only those filesystems which need to be dumped.

n Whenever **dump** requires operator attention, notify by means similar to a *wall*(1) all of the operators in the group ‘‘operator’’.

dump reads the character device associated with the *filesystem* and dumps the contents onto the specified tape device. It searches */etc/fstab* to find the associated character device.

NOTES

rdump is a link to **dump.**

Operator Intervention

dump requires operator intervention on these conditions: end of tape, end of dump, tape write error, tape open error or disk read error (if there are more than a threshold of 32). In addition to alerting all operators implied by the **n** key, **dump** interacts with the operator on **dump's** control terminal at times when **dump** can no longer proceed, or if something is grossly wrong. All questions **dump** poses **must** be answered by typing ‘‘yes’’ or ‘‘no’’, appropriately.

Since making a dump involves a lot of time and effort for full dumps, **dump** checkpoints itself at the start of each tape volume. If writing that volume fails for some reason, **dump** will, with operator permission, restart itself from the checkpoint after the old tape has been rewound and removed, and a new tape has been mounted.

dump reports periodically including usually the percentage of the dump completed, low estimates of the number of blocks to write in 1 Kbyte (or more strictly, TP_BSIZE units from <*protocols/dumprestore.h*>), the number of tapes it will take, the time to completion, and the time to the tape change. The estimated time is given as hours:minutes and is based on the time taken to dump the blocks already on tape. It is normal for this estimate to show variance and the estimate improves over time. The output is verbose, so that others know that the terminal controlling **dump** is busy, and will be for some time.

Suggested Dump Schedule

It is vital to perform full, ‘‘level **0**’’, dumps at regular intervals. When performing a full dump, bring the machine down to single-user mode using **shutdown -is**, Otherwise the dump may not be internally consistent, and may not restore correctly. While preparing for a full dump, it is a good idea to clean the tape drive and heads (most types of drives require head cleaning for approximately every 30 hours of tape motion).

Incremental dumps allow for convenient backup and recovery on a more frequent basis of active files, with a minimum of media and time. However there are some tradeoffs. First, the interval between backups should be kept to a minimum (once a day at least). To guard against data loss as a result of a media failure (a rare, but possible occurrence), it is a good idea to capture active files on (at least) two sets of dump volumes. Another consideration is the desire to keep unnecessary duplication of files to a minimum to save both operator time and media storage. A third consideration is the ease with which a particular backed-up version of a file can be located and restored. The following four-week schedule offers a reasonable tradeoff between these goals.

	Sun	Mon	Tue	Wed	Thu	Fri
Week 1:	Full	5	5	5	5	3
Week 2:		5	5	5	5	3
Week 3:		5	5	5	5	3
Week 4:		5	5	5	5	3

Although the Tuesday — Friday incrementals contain ‘‘extra copies’’ of files from Monday, this scheme assures that any file modified during the week can be recovered from the previous day’s incremental dump.

Dump Parameters

The following table gives a list of available tape formats, size and densities. It is important that the correct parameters be given to **dump,** if they are different from the defaults.

Parameters for cartridge tapes

Cartridge Interface	QIC-24	QIC-120	QIC-150
Number of Tracks	9	15	18
Physical Tape Length (feet)	600	600	600
Effective Tape Length (feet)	5400	9000	10800

Cartridge tapes with multiple tracks have an greater effective length. The tape lengths give above assume a physical tape length of 600 feet. In general the effective tape length can be calculated by multiplying the physical tape length by the number of tracks. Since some tape is usually lost due to tape errors, and because **dump** does not handle end-of-tape gracefully, it pays to be conservative in estimating the effective tape length.

Parameters for half-inch tapes

					Thickness
Reel Sizes (inches)	6.0	7.0	8.5	10.5	
Tape Length (feet)	200	600	1200	2400	1.9 mm
				3600	1.3 mm

The density for these tapes can be any one of the following: 800, 1600, 3200 or 6250 BPI.

Parameters for 8mm tapes

Tape Type	length (meters)	capacity (Mbytes)
P5 (European)	112	2200
P6 (American)	112	2000

There was a bug in **dump** which causes it to miscalculate the number of tapes required when it is given a large value for the density and a small value for tape length. To work around this, a density of 54000 and length of 6000 feet was recommended while using 8mm tapes, rather than the actual density and length, now the calculations are done with floating point numbers, so overflow is no longer an issue; with large capacity drives such as the 8mm and 4mm, it is normally easier to specify capacity as **C 2000k**, rather than trying to calculate a workable density and length.

If you do not wish to use the **C** option, then when using drives with no "inter-record gaps" (i.e., almost every type except 9-track), use the *c* option, and the formula:

capacity in bytes = 7 * densityvalue * lengthvalue and round down a bit to be conservative (allowing for block rewrites, etc.). The density should be kept under 100000 to avoid overflows in the capacity calculations. Thus, for a DAT drive with a 90 meter tape (2 * 10ˆ9 capacity), one might use:

```
2000000000 = 7 * 47619 * 6000
```

or rounding down:

```
dump 0csd 6000 47000
```

EXAMPLES

```
/dev/usr /usr efs rw,raw=/dev/rdsk/dks0d1s6 0 0
```

Here are a few examples on how to dump the /usr filesystem with the above */etc/fstab* entry.

```
dump 0fuc guest@kestrel:/dev/tape /usr
```

will do a level '0' dump of /usr on to a remote cartridge tape device */dev/tape* on host kestrel using the guest account. **dump** also updates the file */etc/dumpdates.*

```
dump 2uc /usr
```

does a level '2' dump of /usr to the local cartridge tape device */dev/tape* and also updates the file */etc/dumpdates.*

```
dump 0sdb 10800 1000 128 /usr
dump 0Cb 125k 128 /usr
```

does a level '0' dump of /usr to the local tape device */dev/tape* using a blocking factor of 128. The tape is specified to have a length of 10800 feet with a density of 1000 BPI (appropriate for a QIC150 drive) in the first case, and a capacity of 125 Mbytes in the second, which allows for retries, lost space to repositioning, etc., also appropriate for a QIC 150 quarter inch tape drive. The ordering of the arguments is dependent on the ordering of the key.

```
dump 1sfc 10800 /dev/mt/tps0d7 /usr
dump 1sfc 10800 /dev/mt/tps0d7 /dev/rdsk/dks0d1s6
```

both do a level '1' dump of /usr to the local cartridge tape device */dev/mt/tps0d7* using a tape length of 10800 feet.

```
dump /usr
```

does a level '9' dump of /usr to the local tape device */dev/tape* and updates the file */etc/dumpdates.*

```
dump 9ucdsf 54000 6000 /dev/mt/tps0d6nrnsv /os
dump 9uCf 2048 /dev/mt/tps0d6nrnsv /os
dump 9uCf 2m /dev/mt/tps0d6nrnsv /os
```

All do a level '9' dump of /os to the local tape device */dev/mt/tps0d6nrnsv* using a tape density of 54000 BPI and tape length of 6000 feet where the tape device being used is an 8mm tape drive (there is a slight difference in capacity between the first form and the others).

```
dump W
```

prints out, for each file system in */etc/dumpdates* the most recent dump date and level, and highlights those file systems that should be dumped.

FILES

/dev/tape	default tape unit to dump to
/etc/dumpdates	new format dump date record
/etc/fstab	dump table: file systems and frequency
/etc/group	to find group *operator*

SEE ALSO

restore(1M), dump(5), fstab(5), group(4), rmt(1M), rhosts(1M), mtio(7), wall(1), shutdown(1M), ctime(3C)

DIAGNOSTICS

While running, **dump** emits many verbose messages.

Exit Codes

```
0 Normal exit.
1 Startup errors encountered.
3 Abnormal termination.
```

BUGS

Fewer than 32 read errors on the filesystem are ignored. Each reel requires a new process, so parent processes for reels already written just hang around until the entire tape is written.

dump with the **W** or **w** options does not report filesystems that have never been recorded in /etc/dumpdates, even if listed in /etc/fstab.

It would be nice if **dump** knew about the dump sequence, kept track of the tapes scribbled on, told the operator which tape to mount when, and provided more assistance for the operator running *restore*.

It is recommended that incremental dumps also be performed with the system running in single-user mode.

dump needs accurate information regarding the length and density of the tapes used. It can dump the filesystem on multiple volumes, but since there is no way of specifying different sizes for multiple tapes, all tapes used should be at least as long as the specified/default length. If **dump** reaches the end of the tape volume unexpectedly (as a result of a longer than actual length specification), it will abort the entire dump.

NAME

ggd – rate-guarantee-granting daemon

SYNOPSIS

ggd [–**d** [**cd**]] [–**c** cpunum]

DESCRIPTION

ggd manages the I/O-rate guarantees that have been granted to processes on the system. The daemon is started from a script in the */etc/rc2.d* directory. It reads the */etc/grio_config* and */etc/grio_disks* files to obtain information about the available hardware devices. Processes can make requests for I/O-rate guarantees by using the *grio_request*(3X) library call. After determining if the I/O rate can be guaranteed, the daemon returns a confirmation or rejection to the calling process.

The */etc/grio_config* and */etc/grio_disks* files are only read when the daemon is started. If these files are edited, the daemon must be stopped and restarted in order to use the new information.

The –**d** option with the **d** modifier causes verbose debugging information to be displayed. The –**d** option with the **c** modifier causes the checksum processing of the **/etc/grio_config** file to be disabled. This allows the system administrator to customize a system by editing the configuration file. The –**c** option causes the daemon to mark the given *cpunum* cpu as a real-time cpu. The cpu will be isolated from the rest of the processors on the system and the *ggd* daemon will be allowed to only run on this cpu. See the *sysmp*(2) reference page for more information on real-time processing.

FILES

/etc/grio_config
/etc/grio_disks

SEE ALSO

cfg(1M), sysmp(2), grio_get_rtgkey(3X), grio_remove_request(3X), grio_request(3X), grio_use_rtgkey(3X), grio_config(4), grio_disks(4)

NOTES

If the *ggd* daemon is killed and restarted, all previous rate guarantees will become invalid. It creates a lock file, */tmp/grio.lock*, to prevent more than one copy of the daemon from running concurrently. If the daemon is killed, this file must be removed before it can be successfully restarted.

NAME

lv_to_xlv – generate a script for converting from lv to XLV

SYNOPSIS

lv_to_xlv [–**f** lvtab_file] [–**o** output_file]

DESCRIPTION

lv_to_xlv parses the file describing the logical volumes used by the local machine and generates the required *xlv_make*(1M) commands to create an equivalent XLV volume. Normally, *lv_to_xlv* uses the logical volume file */etc/lvtab*, but when the –**f** option is specified, the given argument *lvtab_file* is used. If the –**o** option is specified, the *xlv_make*(1M) commands are sent to the file *output_file* instead of stdout.

FILES

/etc/lvtab

SEE ALSO

xlv_make(1M), lvtab(4), xlv(7M)

NOTE

You must be root to run *lv_to_xlv*.

NAME

mkfs – construct a filesystem

SYNOPSIS

mkfs [–**t efs**] efs_mkfs_options
mkfs [–**t xfs**] xfs_mkfs_options

DESCRIPTION

mkfs constructs a filesystem by writing on the special file given as one of the command line arguments. The filesystem constructed is either an EFS filesystem or an XFS filesystem depending on the arguments given. *mkfs* constructs EFS filesystems by executing *mkfs_efs*; XFS filesystems are constructed by executing *mkfs_xfs*. The filesystem type chosen can be forced with the –**t** option (also spelled –**F**). If one of those options is not given, *mkfs* determines which filesystem type to construct by examining its arguments.

SEE ALSO

mkfs_efs(1M), mkfs_xfs(1M)

NAME

mkfs_xfs – construct an XFS filesystem

SYNOPSIS

mkfs_xfs [–**b** subopt=value] [–**d** subopt[=value]] [–**i** subopt=value]
[–**l** subopt[=value]] [–**p** protofile] [–**q**] [–**r** subopt[=value]]
[xlv-device]

DESCRIPTION

mkfs_xfs constructs an XFS filesystem by writing on a special file using the values found in the arguments of the command line. It is invoked automatically by *mkfs*(1M) when *mkfs* is given the –**t xfs** option or options that are specific to XFS.

XFS filesystems are composed of a data section, a log section, and optionally a real-time section. This separation can be accomplished using the XLV volume manager to create a multi-subvolume volume, or by embedding an *internal* log section in the data section. In the former case, the *xlv-device* name is supplied as the final argument. In the latter case a disk partition, *lv*(7M) logical volume, or XLV logical volume without a log subvolume may contain the XFS filesystem, which must be named by the –**d name=***special* option.

Each of the *subopt=value* elements in the argument list above can be given as multiple comma-separated *subopt=value* suboptions if multiple suboptions apply to the same option. Equivalently, each main option may be given multiple times with different suboptions. For example, –**l internal,size=1000b** and –**l internal** –**l size=1000b** are equivalent.

In the descriptions below, sizes are given in bytes, blocks, kilobytes, or megabytes. Sizes are treated as hexadecimal if prefixed by 0x or 0X, octal if prefixed by 0, or decimal otherwise. If suffixed with **b** then the size is converted by multiplying it by the filesystem's block size. If suffixed with **k** then the size is converted by multiplying it by 1024. If suffixed with **m** then the size is converted by multiplying it by 1048576 (1024 * 1024).

–**b** Block size options.

This option specifies the fundamental block size of the filesystem. The valid suboptions are: **log=***value* and **size=***value*; only one can be supplied. The block size is specified either as a base two logarithm value with **log=**, or in bytes with **size=**. The default value is 4096 bytes (4 KB). The minimum value for block size is 512; the maximum is 65536 (64 KB).

–**d** Data section options.

These options specify the location, size, and other parameters of the data section of the filesystem. The valid suboptions are: **agcount=***value*, **file**[=*value*], **name=***value*, and **size=***value*.

The **agcount** suboption is used to specify the number of allocation groups. The data section of the filesystem is divided into allocation groups to improve the performance of XFS. More allocation groups imply that more parallelism can be achieved when allocating blocks and inodes. The minimum allocation group size is 16 MB; the maximum size is just under 4 GB. The data section of the filesystem is divided into *agcount* allocation groups (default value 8, unless the filesystem is smaller than 128 MB or larger than 32 GB).

The **name** suboption is used to specify the name of the special file containing the filesystem. In this case, the log section must be specified as **internal** (with a size, see the –**l** option below) and there can be no real-time section. Either the block or character special device can be supplied. An XLV logical volume with a log subvolume cannot be supplied here.

The **file** suboption is used to specify that the file given by the **name** suboption is a regular file. The suboption value is either 0 or 1, with 1 signifying that the file is regular. This suboption is used only to make a filesystem image (for instance, a miniroot image).

The **size** suboption is used to specify the size of the data section. This suboption is required if –**d file=1** is given. Otherwise, it is only needed if the filesystem should occupy less space than the size of the special file.

–**i** Inode options.

This option specifies the inode size of the filesystem. The XFS inode contains a fixed-size part and a variable-size part. The variable-size part, whose size is affected by this option, can contain: directory data, for small directories; symbolic link data, for small symbolic links; the extent list for the file, for files with a small number of extents; and the root of a tree describing the location of extents for the file, for files with a large number of extents.

The valid suboptions are: **log=***value*, **perblock=***value*, and **size=***value*; only one can be supplied. The inode size is specified either as a base two logarithm value with **log=**, in bytes with **size=**, or as the number fitting in a filesystem block with **perblock=**. The default value is 256 bytes. The minimum value for inode size is 128, and the maximum value is 2048 (2 KB) subject to the restriction that the inode size cannot exceed one half of the filesystem block size.

–**l** Log section options.

These options specify the location, size, and other parameters of the log section of the filesystem. The valid suboptions are: **internal[=***value***]** and **size=***value*.

The **internal** suboption is used to specify that the log section is a piece of the data section instead of being a separate part of an XLV logical volume. The suboption value is either 0 or 1, with 1 signifying that the log is internal.

The **size** suboption is used to specify the size of the log section. This suboption is required if –**l internal[=1]** is given. Otherwise, it is only needed if the log section of the filesystem should occupy less space than the size of the special file.

–**p** *protofile*

If the optional –**p** *protofile* argument is given, *mkfs_xfs* uses *protofile* as a prototype file and takes its directions from that file. The blocks and inodes specifiers in the *protofile* are provided for backwards compatibility, but are otherwise unused. The prototype file contains tokens separated by spaces or newlines. A sample prototype specification follows (line numbers have been added to aid in the explanation):

```
1        /stand/diskboot
2        4872 110
```

```
3       d--777 3 1
4       usr     d--777 3 1
5       sh      ---755 3 1 /bin/sh
6       ken     d--755 6 1
7               $
8       b0      b--644 3 1 0 0
9       c0      c--644 3 1 0 0
10      fifo    p--644 3 1
11      slink   l--644 3 1 /a/symbolic/link
12      :  This is a comment line
13      $
14      $
```

Line 1 is a dummy string. (It was formerly the bootfilename.) It is present for backward compatibility; boot blocks are not used on SGI machines.

Note that some string of characters must be present as the first line of the proto file to cause it to be parsed correctly; the value of this string is immaterial since it is ignored.

Line 2 contains two numeric values (formerly the numbers of blocks and inodes). These are also merely for backward compatibility: two numeric values must appear at this point for the proto file to be correctly parsed, but their values are immaterial since they are ignored.

Lines 3-11 tell *mkfs_xfs* about files and directories to be included in this filesystem. Line 3 specifies the root directory. Lines 4-6 and 8-10 specifies other directories and files. Note the special symbolic link syntax on line 11.

The **$** on line 7 tells *mkfs_xfs* to end the branch of the filesystem it is on, and continue from the next higher directory. It must be the last character on a line. The colon on line 12 introduces a comment; all characters up until the following newline are ignored. Note that this means you may not have a file in a prototype file whose name contains a colon. The **$** on lines 13 and 14 end the process, since no additional specifications follow.

File specifications give the mode, the user ID, the group ID, and the initial contents of the file. Valid syntax for the contents field depends on the first character of the mode.

The mode for a file is specified by a 6-character string. The first character specifies the type of the file. The character range is **–bcdpl** to specify regular, block special, character special, directory files, named pipes (fifos) and symbolic links, respectively. The second character of the mode is either **u** or – to specify setuserID mode or not. The third is **g** or – for the setgroupID mode. The rest of the mode is a 3-digit octal number giving the owner, group, and other read, write, execute permissions (see *chmod*(1)).

Two decimal number tokens come after the mode; they specify the user and group IDs of the owner of the file.

If the file is a regular file, the next token of the specification may be a pathname from which the contents and size are copied. If the file is a block or character special file, two decimal numbers follow that give the major and minor device numbers. If the file is a symbolic link, the next token of the specification is used as the contents of the link. If the file is a directory, *mkfs_xfs* makes the entries **.** and **..** and then reads a list of names and (recursively) file specifications for the entries in the directory. As noted above, the scan is terminated with the token **$**.

–q Quiet option.

Normally *mkfs_xfs* prints the parameters of the filesystem to be constructed; the **–q** flag suppresses this.

–r Real-time section options.

These options specify the location, size, and other parameters of the real-time section of the filesystem. The valid suboptions are: **extsize=***value* and **size=***value*.

The **extsize** suboption is used to specify the size of the blocks in the real-time section of the filesystem. This size must be a multiple of the filesystem block size. The minimum allowed value (and the default) is 64 KB; the maximum allowed value is 1 GB. The real-time extent size should be carefully chosen to match the parameters of the physical media used.

The **size** suboption is used to specify the size of the real-time section. This suboption is only needed if the real-time section of the filesystem should occupy less space than the size of the XLV real-time subvolume.

SEE ALSO

mkfs(1M), mkfs_efs(1M)

BUGS

With a prototype file, it is not possible to specify hard links.

NAME

restore, rrestore – incremental file system restore

SYNOPSIS

/sbin/restore key [name ...]
/sbin/rrestore key [name ...]

DESCRIPTION

restore reads tapes dumped with the **dump (1M)** command and restores them *relative to the current directory*. Its actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter and possibly one or more function modifiers. Any arguments supplied for specific options are given as subsequent words on the command line, in the same order as that of the options listed. Other arguments to the command are file or directory names specifying the files that are to be restored. Unless the **h** key is specified (see below), the appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the key is specified by one of the following letters:

r Restore the entire tape. The tape is read and its full contents loaded into the current directory. This should not be done lightly; the **r** key should only be used to restore a complete ''level 0'' dump tape onto a clear file system or to restore an incremental dump tape after a full level zero restore. Thus

```
/etc/mkfs /dev/dsk/dks0d2s0
/etc/mount /dev/dsk0d2s0 /mnt
cd /mnt
restore r
```

is a typical sequence to restore a complete dump. Another **restore** can be done to get an incremental dump in on top of this. Note that **restore** leaves a file *restoresymtable* in the root directory to pass information between incremental restore passes. This file should be removed when the last incremental tape has been restored. Also, see the note in the **BUGS** section below.

R Resume restoring. **restore** requests a particular tape of a multi volume set on which to restart a full restore (see the **r** key above). This allows **restore** to be interrupted and then restarted.

x The named files are extracted from the tape. If the named file matches a directory whose contents had been written onto the tape, and the **h** key is **not** specified, the directory is recursively extracted. The owner, modification time, and mode are restored (if possible). If no file argument is given, then the root directory is extracted, which results in the entire content of the tape being extracted, unless the **h** key has been specified.

t The names of the specified files are listed if they occur on the tape. If no file argument is given, then the root directory is listed, which results in the entire content of the tape being listed, unless the **h** key has been specified. Note that the **t** key replaces the function of the old *dumpdir* program.

i This mode allows interactive restoration of files from a dump tape. After reading in the directory information from the tape, **restore** provides a shell like interface that allows the user to move around the directory tree selecting files to be extracted. The available commands are given below; for those commands that require an argument, the default is the current directory.

ls [arg] – List the current or specified directory. Entries that are directories are appended with a ‘‘/’’. Entries that have been marked for extraction are prepended with a ‘‘*’’. If the verbose key is set the inode number of each entry is also listed.

cd arg – Change the current working directory to the specified argument.

pwd – Print the full pathname of the current working directory.

add [arg] – The current directory or specified argument is added to the list of files to be extracted. If a directory is specified, then it and all its descendents are added to the extraction list (unless the **h** key is specified on the command line). Files that are on the extraction list are prepended with a ‘‘*’’ when they are listed by **ls**.

delete [arg] – The current directory or specified argument is deleted from the list of files to be extracted. If a directory is specified, then it and all its descendents are deleted from the extraction list (unless the **h** key is specified on the command line). The most expedient way to extract most of the files from a directory is to add the directory to the extraction list and then delete those files that are not needed.

extract – All the files that are on the extraction list are extracted from the dump tape. **restore** will ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

setmodes – All the directories that have been added to the extraction list have their owner, modes, and times set; nothing is extracted from the tape. This is useful for cleaning up after a **restore** has been prematurely aborted.

verbose – The sense of the **v** key is toggled. When set, the verbose key causes the **ls** command to list the inode numbers of all entries. It also causes **restore** to print out information about each file as it is extracted.

help – List a summary of the available commands.

quit – **restore** immediately exits, even if the extraction list is not empty.

The following characters may be used in addition to the letter that selects the function desired.

b The next *argument* to **restore** is used as the block size of the tape (in kilobytes). If the **b** option is not specified, **restore** tries to determine the tape block size dynamically, but will only be able to do so if the block size is 32 or less. For larger sizes, the **b** option must be used with **restore**.

f The next *argument* to **restore** is used as the name of the archive instead of */dev/tape.* If the name of the file is ‘‘–’’, **restore** reads from standard input. Thus, *dump*(1M) and **restore** can be used in a pipeline to dump and restore a file system with the command

```
        dump 0f - /usr | (cd /mnt; restore xf -)
```

If the name of the file is of the format *machine:device* then the filesystem dump is restored from the specified machine over the network. **restore** creates a remote server */etc/rmt,* on the client machine to access the tape device. Since **restore** is normally run by root, the name of the local machine must appear in the *.rhosts* file of the remote machine. If the file name *argument* is of the form *user@machine:device,* **restore** will attempt to execute as the specified use on the remote machine. The specified user must have a *.rhosts* file on the remote machine that allows root from the local machine.

v Normally **restore** does its work silently. The **v** (verbose) key causes it to type the name of each file it treats preceded by its file type.

y **restore** will not ask whether it should abort the restore if gets a tape error. It will always try to skip over the bad tape block(s) and continue as best it can.

m **restore** will extract by inode numbers rather than by file name. This is useful if only a few files are being extracted, and one wants to avoid regenerating the complete pathname to the file.

h **restore** extracts the actual directory, rather than the files that it references. This prevents hierarchical restoration of complete subtrees from the tape.

s The next *argument* to **restore** is a number which selects the dump file when there are multiple dump files on the same tape. File numbering starts at 1.

n Only those files which are newer than the file specified by the next *argument* are considered for restoration. **restore** looks at the modification time of the specified file using the **stat(2)** system call.

e No existing files are overwritten.

E Restores only non-existent files or newer versions (as determined by the file status change time stored in the dump file) of existing files. Note that the **ls(1)** command shows the modification time and not the file status change time. See **stat(2)** for more details.

d Turn on debugging output.

o Normally **restore** does not use **chown(2)** to restore files to the original user and group id unless it is being run by the super-user (or with the effective user id of zero). This is to provide Berkeley style semantics. This can be overridden with the **o** option which will result in **restore** attempting to restore the original ownership to the files.

N Do not write anything to the disk. This option can be used to validate the tapes after a dump. If invoked with the "r" option, **restore** goes through the motion of reading all the dump tapes without actually writing anything to the disk.

DIAGNOSTICS

restore complains about bad key characters.

On getting a read error, *restore* prints out diagnostics. If **y** has been specified, or the user responds ‘‘y’’, **restore** will attempt to continue the restore.

If the dump extends over more than one tape, **restore** will ask the user to change tapes. If the **x** or **i** key has been specified, **restore** will also ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

There are numerous consistency checks that can be listed by **restore.** Most checks are self-explanatory or can ‘‘never happen’’. Common errors are given below.

Converting to new file system format.
: A dump tape created from the old file system has been loaded. It is automatically converted to the new file system format.

<filename>: not found on tape
: The specified file name was listed in the tape directory, but was not found on the tape. This is caused by tape read errors while looking for the file, and from using a dump tape created on an active file system.

expected next file <inumber>, got <inumber>
: A file that was not listed in the directory showed up. This can occur when using a dump tape created on an active file system.

Incremental tape too low
: When doing incremental restore, a tape that was written before the previous incremental tape, or that has too low an incremental level has been loaded.

Incremental tape too high
: When doing incremental restore, a tape that does not begin its coverage where the previous incremental tape left off, or that has too high an incremental level has been loaded.

Tape read error while restoring <filename>
Tape read error while skipping over inode <inumber>
Tape read error while trying to resynchronize
: A tape read error has occurred. If a file name is specified, then its contents are probably partially wrong. If an inode is being skipped or the tape is trying to resynchronize, then no extracted files have been corrupted, though files may not be found on the tape.

resync restore, skipped <num> blocks
: After a tape read error, **restore** may have to resynchronize itself. This message lists the number of blocks that were skipped over.

Error while writing to file /tmp/rstdir*
: An error was encountered while writing to the temporary file containing information about the directories on tape. Use the TMPDIR environment variable to relocate this file in a directory which has more space available.

Error while writing to file /tmp/rstdir*
: An error was encountered while writing to the temporary file containing information about the owner, mode and timestamp information of directories. Use the TMPDIR environment variable to relocate this file in a directory which has more space available.

EXAMPLES

```
restore r
```

will restore the entire tape into the current directory, reading from the default tape device */dev/tape.*

```
restore rf guest@kestrel.sgi.com:/dev/tape
```

will restore the entire tape into the current directory, reading from the remote tape device */dev/tape* on host kestrel.sgi.com using the guest account.

```
restore x /etc/hosts /etc/fstab /etc/myfile
```

will restore the three specified files into the current directory, reading from the default tape device */dev/tape.*

```
restore x /dev/dsk
```

will restore the entire /dev/dsk directory and subdirectories recursively into the current directory, reading from the default tape device */dev/tape*

```
restore rN
```

will read the entire tape and go through all the motions of restoring the entire dump, without writing to the disk. This can be used to validate the dump tape.

```
restore xe /usr/dir/foo
```

will restore (recursively) all files in the given directory /usr/dir/foo. However, no existing files are overwritten.

```
restore xn /usr/dir/bar
```

will restore (recursively) all files which are newer than the given file /usr/dir/bar.

FILES

/dev/tape
: This is the default tape device used unless the environment variable TAPE is set.

/tmp/rstdir*
: This temporary file contains the directories on the tape. If the environment variable TMPDIR is set, then the file will be created in that directory.

/tmp/rstmode*
: This temporary file contains the owner, mode, and time stamps for directories. If the environment variable TMPDIR is set, then the file will be created in that directory.

./restoresymtable
: Information is passed between incremental restores in this file.

SEE ALSO

dump(1M), mount(1M), mkfs(1M), rmt(1M), rhosts(4), mtio(7)

NOTES

rrestore is a link to **restore.**

BUGS

restore can get confused when doing incremental restores from dump tapes that were made on active file systems.

A ''level 0'' dump must be done after a full restore. Because restore runs in user code, it has no control over inode allocation. This results in the files being restored having an inode numbering different from the filesystem that was originally dumped. Thus a full dump must be done to get a new set of directories reflecting the new inode numbering, even though the contents of the files is unchanged, so that later incremental dumps will be correct.

Existing dangling symlinks are modified even if the **e** option is supplied, if the dump tape contains a hard link by the same name.

NAME

xfs_check – check XFS filesystem consistency

SYNOPSIS

xfs_check [–**i** ino] ... [–**s**] [–**v**] xlvspecial

xfs_check –**d** [–**i** ino] ... [–**s**] [–**v**] diskspecial

xfs_check –**f** [–**i** ino] ... [–**s**] [–**v**] file

DESCRIPTION

xfs_check checks whether an XFS filesystem is consistent. It is normally run only when there is reason to believe that the filesystem has a consistency problem. The filesystem to be checked is specified by the *xlvspecial* or *diskspecial* argument, which should be the disk or volume device for the filesystem. Filesystems stored in files can also be checked, using the –**f** flag. The filesystem should normally be unmounted or read-only during the execution of *xfs_check*, otherwise spurious problems are reported.

The options to *xfs_check* are:

–**d** Specifies that the *special* device is a disk partition name or an *lv*(7M) volume name (as opposed to an XLV logical volume).

–**f** Specifies that the special device is actually a file (see the *mkfs_xfs -d file* option). This might happen if an image copy of a filesystem has been made into an ordinary file.

–**s** Specifies that only serious errors should be reported. Serious errors are those that make it impossible to find major data structures in the filesystem. This option can be used to cut down the amount of output when there is a serious problem, when it might make it difficult to see what the real problem is.

–**v** Specifies verbose output; it is impossibly long for a reasonably-sized filesystem. This option is intended for internal use only.

–**i** *ino* Specifies verbose behavior for a specific inode. For instance, it can be used to locate all the blocks associated with a given inode.

Any output from *xfs_check* means that the filesystem has an inconsistency. The only repair mechanism available is to dump the filesystem with *xfsdump(1M),* then use *mkfs_xfs(1M)* to make a new filesystem, then use *xfsrestore(1M)* to restore the data.

DIAGNOSTICS

Under two circumstances, *xfs_check* unfortunately might dump core rather than produce useful output. First, if the filesystem is completely corrupt, a core dump might be produced instead of the message ‘‘*xxx* `is not a valid filesystem`.’’ Second, if the filesystem is very large (has many files) then *xfs_check* might run out of memory.

The following is a description of the most likely problems and the associated messages. Most of the diagnostics produced are only meaningful with an understanding of the structure of the filesystem.

xxx `is not an XLV volume device name`
: The **–d** option is needed for filesystems that reside in disk partitions instead of in XLV volumes.

`agf_freeblks` *n*`, counted` *m* `in ag` *a*
: The freeblocks count in the allocation group header for allocation group *a* doesn't match the number of blocks counted free.

`agf_longest` *n*`, counted` *m* `in ag` *a*
: The longest free extent in the allocation group header for allocation group *a* doesn't match the longest free extent found in the allocation group.

`agi_count` *n*`, counted` *m* `in ag` *a*
: The allocated inode count in the allocation group header for allocation group *a* doesn't match the number of inodes counted in the allocation group.

`agi_freecount` *n*`, counted` *m* `in ag` *a*
: The free inode count in the allocation group header for allocation group *a* doesn't match the number of inodes counted free in the allocation group.

`block` *a/b* `expected inum 0 got` *i*
: The block number is specified as a pair (allocation group number, block in the allocation group). The block is used multiple times (shared), between multiple inodes. This message usually follows a message of the next type.

`block` *a/b* `expected type unknown got` *y*
: The block is used multiple times (shared).

`block` *a/b* `type unknown not expected`
: The block is unaccounted for (not in the freelist and not in use).

`link count mismatch for inode` *nnn* `(name` *xxx*`), nlink` *m*`, counted` *n*
: The inode has a bad link count (number of references in directories).

`rtblock` *b* `expected inum 0 got` *i*
: The block is used multiple times (shared), between multiple inodes. This message usually follows a message of the next type.

`rtblock` *b* `expected type unknown got` *y*
: The real-time block is used multiple times (shared).

`rtblock` *b* `type unknown not expected`
: The real-time block is unaccounted for (not in the freelist and not in use).

`sb_fdblocks` *n*`, counted` *m*
: The number of free data blocks recorded in the superblock doesn't match the number counted free in the filesystem.

`sb_frextents` *n*`, counted` *m*
: The number of free real-time extents recorded in the superblock doesn’t match the number counted free in the filesystem.

`sb_icount` *n*`, counted` *m*
: The number of allocated inodes recorded in the superblock doesn’t match the number allocated in the filesystem.

`sb_ifree` *n*`, counted` *m*
: The number of free inodes recorded in the superblock doesn’t match the number free in the filesystem.

SEE ALSO

mkfs_xfs(1M), xfsdump(1M), xfsrestore(1M), xfs(4)

NAME

xfs_estimate – estimate the space that an XFS filesystem will take

SYNOPSIS

xfs_estimate [–**h?**] [–**b** blocksize] [–**i** logsize] [–**e** logsize] [–**v**] directory ...

DESCRIPTION

For each *directory* argument, *xfs_estimate* estimates the space that directory would take if it were copied to an XFS filesystem. Note that *xfs_estimate* does not cross mount points. Also, the following definitions are used: KB = *1024, MB = *1024*1024, GB = *1024*1024*1024.

–**b** *blocksize*

Use *blocksize* instead of the default blocksize of 4096 bytes. The modifier **k** may be used after the number to indicate multiplication by 1024. For example,

```
xfs_estimate -b 64k /
```

requests an estimate of the space required by the directory / on an XFS filesystem using a blocksize of 64k (65536) bytes.

–**v** Display more information, formatted.

–**h** Display usage message.

–**?** Display usage message.

–**i, -e** *logsize*

Use *logsize* instead of the default log size of 10 MB. **-i** refers to an internal log, while **-e** refers to an external log. The modifier **k** or **m** may be used after the number to indicate multiplication by 1024 or 1048576, respectively.

For example,

```
xfs_estimate -i 1m /
```

requests an estimate of the space required by the directory / on an XFS filesystem using an internal log of 1 megabyte.

EXAMPLES

```
% xfs_estimate -e 10m /var/tmp
/var/tmp will take about 4 megabytes
     with the external log using 2560 blocks or about 11 megabytes

% xfs_estimate -v -e 10m /var/tmp
directory                      bsize  blocks   megabytes   logsize
/var/tmp                        4096     792         4MB  10485760

% xfs_estimate -v /var/tmp
directory                      bsize  blocks   megabytes   logsize
/var/tmp                        4096    3352        14MB  10485760
```

```
% xfs_estimate /var/tmp
/var/tmp will take about 14 megabytes
```

NAME

xfs_growfs – expand an XFS filesystem

SYNOPSIS

xfs_growfs [–**D** size] [–**d**] [–**i**] [–**L** size] [–**l**]
[–**R** size] [–**r**] [–**x**] mount-point

DESCRIPTION

xfs_growfs expands an existing XFS filesystem, see *xfs*(4). The *mount-point* argument should be the pathname of the directory where the filesystem is mounted. The filesystem must be mounted to be grown, see *mount*(1M). The existing contents of the filesystem are undisturbed, and the added space becomes available for additional file storage.

The –**d** option specifies that the data section of the filesystem will be grown. If the –**D** *size* option is given, the data section will be grown to that size, otherwise the data section will be grown to the largest size possible. The size is expressed in filesystem blocks.

The –**r** option specifies that the real-time section of the filesystem will be grown. If the –**R** *size* option is given, the real-time section will be grown to that size, otherwise the real-time section will be grown to the largest size possible. The size is expressed in filesystem blocks. The filesystem does not need to have contained a real-time section before the *growfs* operation.

The –**l** option specifies that the log section of the filesystem will be grown, shrunk, or moved. If the –**L** *size* option is given, the log section will be changed to be that size, if possible. The size is expressed in filesystem blocks. The size of an internal log must be smaller than the size of an allocation group (this value is printed at *mkfs*(1M) time). If the –**i** option is given, the new log will be an internal log (inside the data section). If the –**x** option is given, the new log will be an external log (in an XLV log subvolume). If neither –**i** nor –**x** is given with –**l**, then the log will continue to be internal or external as it was before.

xfs_growfs is most often used in conjunction with *logical volumes*, see *xlv*(7M) or *lv*(7M). However, it can also be used on a regular disk partition, for example if a partition has been enlarged while retaining the same starting block.

PRACTICAL USE

Filesystems normally occupy all of the space on the device where they reside. In order to grow a filesystem, it is necessary to provide added space for it to occupy. Therefore there must be at least one spare new disk partition available. Adding the space is done through the mechanism of *logical volumes*. If the filesystem already resides on a logical volume, the volume is simply extended using *mklv*(1M) or *xlv_admin*(1M). If the filesystem is currently on a regular partition, it is necessary to create a new logical volume whose first member is the existing partition, with subsequent members being the new partition(s) to be added. Again, *mklv* or *xlv_admin* is used for this. In either case *xfs_growfs* is run on the mounted filesystem, and the expanded filesystem is then available for use.

SEE ALSO

mkfs_xfs(1M), mklv(1M), mount(1M), xlv_make(1M), lv(7M), xlv(7M)

NAME

xfsdump – XFS filesystem incremental dump utility

SYNOPSIS

xfsdump [–**f** destination] [–**l** level] [–**s** pathname ...]
[–**v** verbosity] [–**F**] [–**I** [subopt=value ...]] [–**J**]
[–**L** session_label] [–**M** media_label] [–**R**] [–]
filesystem

DESCRIPTION

xfsdump backs up files in a filesystem. The files are dumped to storage media, a regular file, or standard output. Options allow the operator to have all files dumped, just files that have changed since a previous dump, or just files contained in a list of pathnames.

The *xfsrestore*(1M) utility re-populates a filesystem with the contents of the dump.

Each invocation of *xfsdump* dumps just one filesystem. That invocation is termed a dump session. The dump session sends a single dump stream to the destination. The dump stream can span several media objects, and a single media object can contain several dump streams. The typical media object is a tape cartridge. The media object records the dump stream as one or more media files. A media file is a self-contained partial dump. The portion of the dump stream contained on a media object can be split into several media files to minimize the impact of media dropouts on the entire dump stream.

xfsdump maintains an online dump inventory in */var/xfsdump/inventory*. The –**I** option displays the inventory contents hierarchically. The levels of the hierarchy are: filesystem, dump session, stream, and media file.

–**f** *destination*
Specifies the dump destination. It can be the pathname of a device (such as a tape drive), a regular file, or a remote tape drive (see *rmt*(1M)). This option must be omitted if the standard output option (a lone – preceding the filesystem specification) is specified.

–**l** *level*
Specifies a dump level of 0 to 9. The dump level determines the base dump to which this dump is relative. The base dump is the most recent dump at a lesser level. A level 0 dump is absolute – all files are dumped. A dump level where 1 <= *level* <= 9 is referred to as an incremental dump. Only files that have been changed since the base dump are dumped. Subtree dumps (see the –**s** option below) cannot be used as the base for incremental dumps.

–**s** *pathname ...*
Restricts the dump to files contained in the specified pathnames (subtrees). Up to 100 pathnames can be specified. A *pathname* must be relative to the mount point of the filesystem. For example, if a filesystem is mounted at */d2*, the *pathname* argument for the directory */d2/users* is ‘‘users’’. A *pathname* can be a file or a directory; if it is a directory, the entire hierarchy of files and subdirectories rooted at that directory is dumped. Subtree dumps cannot be used as the base for incremental dumps (see the –**l** option above).

–**v** *verbosity_level*
: Specifies the level of detail of the messages displayed during the course of the dump. The argument can be **silent**, **verbose**, or **trace**. The default is **verbose**.

–**F** Don't prompt the operator. When *xfsdump* encounters a media object containing non-xfsdump data, *xfsdump* normally asks the operator for permission to overwrite. With this option the overwrite is performed, no questions asked. When *xfsdump* encounters end-of-media, *xfsdump* normally asks the operator if another media object will be provided. With this option the dump is instead interrupted.

–**I** Displays the *xfsdump* inventory (no dump is performed). *xfsdump* records each dump session in an online inventory in */var/xfsdump/inventory*. *xfsdump* uses this inventory to determine the base for incremental dumps. It is also useful for manually identifying a dump session to be restored. Suboptions to filter the inventory display are described later.

–**J** Inhibits the normal update of the inventory. This is useful when the media being dumped to will be discarded or overwritten.

–**L** *session_label*
: Specifies a label for the dump session. It can be any arbitrary string up to 255 characters long.

–**M** *media_label*
: Specifies a label for all media objects (for example, tape cartridges) written during the session. It can be any arbitrary string up to 255 characters long.

–**R** Resumes a previously interrupted dump session. If the most recent dump at this dump's level (–**l** option) was interrupted, this dump contains only files not in the interrupted dump and consistent with the incremental level. However, files contained in the interrupted dump that have been subsequently modified are re-dumped.

– A lone – causes the dump stream to be sent to the standard output, where it can be piped to another utility such as *xfsrestore*(1M) or redirected to a file. This option cannot be used with the –**f** option. The – must follow all other options, and precede the filesystem specification.

The filesystem, *filesystem*, can be specified either as a mount point or as a special device file (for example, */dev/dsk/dks0d1s0*). The filesystem must be mounted to be dumped.

NOTES

Dump Interruption

A dump can be interrupted at any time and later resumed. To interrupt, type control-C (or the current terminal interrupt character). The operator is prompted to select one of several operations, including dump interruption. After the operator selects dump interruption, the dump continues until a convenient break point is encountered (typically the end of the current file). Very large files are broken into smaller subfiles, so the wait for the end of the current file is brief.

Dump Resumption

A previously interrupted dump can be resumed by specifying the –**R** option. If the most recent dump at the specified level was interrupted, the new dump does not include files already dumped, unless they have changed since the interrupted dump.

Media Management

A single media object can contain many dump streams. Conversely, a single dump stream can span multiple media objects. If a dump stream is sent to a media object already containing one or more dumps, *xfsdump* appends the new dump stream after the last dump stream. Media files are never overwritten. If end-of-media is encountered during the course of a dump, the operator is prompted to insert a new media object into the drive. The dump continuation is appended after the last media file on the new media object.

Inventory

Each dump session updates an inventory database in */var/xfsdump/inventory*. *xfsdump* uses the inventory to determine the base of incremental and resumed dumps.

This database can be displayed by invoking *xfsdump* with the –**I** option. The display uses tabbed indentation to present the inventory hierarchically. The first level is filesystem. The second level is session. The third level is media stream (currently only one stream is supported). The fourth level lists the media files sequentially composing the stream.

Several suboptions are available to filter the display. Specifying –**I depth=***n* (where *n* is 1, 2, or 3) limits the hierarchical depth of the display. Specifying –**I mobjid=***value* (where *value* is a media id) or –**I mobjlabel=***value* (where *value* is a media label) limits the display to media files contained in the specified media object. Similarly, the display can be restricted to a specific filesystem identified by mount point using –**I mnt=***host-qualified_mount_point_pathname*, by filesystem id using –**I fsid=***filesystem_id*, or by device using –**I dev=***host-qualified_device_pathname*. At most three suboptions may be specified at once: one to constrain the depth, one to constrain the media object, and one to constrain the filesystem. For example, –**I depth=1,mobjlabel="tape 1",mnt=host1:/test_mnt** would display only the filesystem information (depth=1) for those filesystems which were mounted on *host1:/test_mnt* at the time of the dump, and only those filesystems dumped to the media object labeled "tape 1".

There is currently no way to remove dumps from the inventory.

An additional media file is placed at the end of each dump stream. This media file contains the inventory information for the current dump session. This is currently unused.

When operating in the miniroot environment, *xfsdump* does not create and does not reference the inventory database. Thus incremental and resumed dumps are not allowed.

Labels

The operator can specify a label to identify the dump session and a label to identify a media object. The session label is placed in every media file produced in the course of the dump, and is recorded in the inventory.

The media label is used to identify media objects, and is independent of the session label. Each media file on the media object contains a copy of the media label. An error will be returned if the operator specifies a media label which does not match the media label on a media object containing valid media files. Media labels are recorded in the inventory.

UUIDs

UUIDs (Universally Unique Identifiers) are used in three places: to identify the filesystem being dumped, to identify the dump session, and to identify each media object. The inventory display (–**I**) includes all of these.

Dump Level Usage

The dump level mechanism provides a structured form of incremental dumps. A dump of level *level* includes only files that have changed since the most recent dump at a level less than *level*. For example, the operator can establish a dump schedule that involves a full dump every Friday and a daily incremental dump containing only files that have changed since the previous dump. In this case Friday's dump would be at level 0, Saturday's at level 1, Sunday's at level 2, and so on, up to the Thursday dump at level 6.

The above schedule results in a very tedious restore procedure to fully reconstruct the Thursday version of the filesystem; *xfsrestore* would need to be fed all 7 dumps in sequence. A compromise schedule is to use level 1 on Saturday, Monday, and Wednesday, and level 2 on Sunday, Tuesday, and Thursday. The Monday and Wednesday dumps would take longer, but the worst case restore requires the accumulation of just three dumps, one each at level 0, level 1, and level 2.

Miniroot Restrictions

xfsdump is subject to the following restrictions when operated in the miniroot environment: non-restartable, no incrementals, no online inventory, synchronous I/O.

FILES

/var/xfsdump/inventory	dump inventory database

SEE ALSO

rmt(1M), xfsrestore(1M)

DIAGNOSTICS

The exit code is 0 on normal completion, non-zero if an error occurs or the dump is terminated by the operator.

BUGS

xfsdump always rewinds tape media, then seeks to the end of the last dump prior to appending the current dump.

Some of the command line options are not checked until the media has been rewound. Thus, errors in those options are not reported immediately.

xfsdump does not dump unmounted filesystems.

The dump frequency field of */etc/fstab* is not supported.

xfsdump does not have the capability to send mail when operator intervention is required.

Only one –**f** option is allowed, because *xfsdump* does not have the capability to partition the dump into multiple streams, each directed to a different media drive.

No means is provided to remove media objects from the inventory.

xfsdump requires root privilege (except for inventory display).

xfsdump can only dump XFS filesystems.

The media format used by *xfsdump* can only be understood by *xfsrestore*.

Dumps may not be written to fixed block size tape devices via the remote tape device interface.

xfsdump does not know how to manage CD-ROM or other removable disk drives.

NAME

xfsrestore – XFS filesystem incremental restore utility

SYNOPSIS

xfsrestore [–**a** housekeeping] [**-e**] [–**f** source] [–**i**]
[–**n** file] [–**r**] [–**s** subtree ...] [–**t**] [–**v** verbosity]
[–**E**] [–**I** [subopt=value ...]] [–**L** session_label]
[–**S** session_id] [–]
destination

DESCRIPTION

xfsrestore restores filesystems from dumps produced by *xfsdump*(1M). Two modes of operation are available: simple and cumulative.

The default is simple mode. *xfsrestore* populates the specified destination directory, *destination*, with the files contained in the dump media.

The –**r** option specifies the cumulative mode. Successive invocations of *xfsrestore* are used to apply a chronologically ordered sequence of delta dumps to a base (level 0) dump. The contents of the filesystem at the time each dump was produced is reproduced. This can involve adding, deleting, renaming, linking, and unlinking files and directories.

A delta dump is defined as either an incremental dump (*xfsdump* –**l** option with level > 0) or a resumed dump (*xfsdump* –**R** option). The deltas must be applied in the order they were produced. Each delta applied must have been produced with the previously applied delta as its base.

–**a** *housekeeping*
Each invocation of *xfsrestore* creates a directory called *xfsrestorehousekeeping*. This directory is normally created directly under the *destination* directory. The –**a** option allows the operator to specify an alternate directory, *housekeeping*, in which *xfsrestore* creates the *xfsrestorehousekeeping* directory. When performing a cumulative (–**r** option) restore, each successive invocation of *xfsrestore* must specify the same alternate directory.

–**e** Prevents *xfsrestore* from overwriting existing files in the *destination* directory.

–**f** *source*
Specifies the source of the dump to be restored. This can be the pathname of a device (such as a tape drive), a regular file, or a remote tape drive (see *rmt*(1M)). This option must be omitted if the standard input option (a lone – preceding the *destination* specification) is specified.

–**i** Selects interactive operation. Once the on-media directory hierarchy has been read, an interactive dialogue is begun. The operator uses a small set of commands to peruse the directory hierarchy, selecting files and subtrees for extraction. The available commands are given below. Initially nothing is selected, except for those subtrees specified with –**s** command line options.

ls [*arg*] List the entries in the current directory or the specified directory, or the specified non-directory file entry. Both the entry's original inode number and name are displayed. Entries that are directories are appended with a ‘‘/’’. Entries that have been selected for extraction are prepended with a ‘‘*’’.

cd [*arg*]	Change the current working directory to the specified argument, or to the filesystem root directory if no argument is specified.
pwd	Print the pathname of the current directory, relative to the filesystem root.
add [*arg*]	The current directory or specified file or directory within the current directory is selected for extraction. If a directory is specified, then it and all its descendents are selected. Entries that are selected for extraction are prepended with a ‘‘*’’ when they are listed by **ls**.
delete [*arg*]	The current directory or specified file or directory within the current directory is deselected for extraction. If a directory is specified, then it and all its descendents are deselected. The most expedient way to extract most of the files from a directory is to select the directory and then deselect those files that are not needed.
extract	Ends the interactive dialogue, and causes all selected subtrees to be restored.
quit	*xfsrestore* ends the interactive dialogue and immediately exits, even if there are files or subtrees selected for extraction.
help	List a summary of the available commands.

Simultaneous cumulative (–**r** option) and interactive restores are not allowed.

–**n** *file*
: Allows *xfsrestore* to restore only files newer than *file.* The modification time of *file* (i.e., as displayed with the **ls -l** command) is compared to the i-node modification time of each file on the source media (i.e., as displayed with the **ls -lc** command) . A file is restored from media only if its i-node modification time is greater than or equal to the modification time of *file.*

–**r** Selects the cumulative mode of operation.

–**s** *subtree*
: Specifies a subtree to restore. Any number of –**s** options are allowed. The restore is constrained to the union of all subtrees specified. Each subtree is specified as a pathname relative to the restore *destination*. If a directory is specified, the directory and all files beneath that directory are restored. Simultaneous cumulative (–**r** option) and subtree restores are not allowed.

–**t** Displays the contents of the dump, but does not create or modify any files or directories. It may be desirable to set the verbosity level to **silent** when using this option.

–**v** *verbosity_level*
: Specifies the level of detail of the messages displayed during the course of the restore. The argument can be **silent**, **verbose**, or **trace**. The default is **verbose**.

–**E** Prevents *xfsrestore* from overwriting newer versions of files. The i-node modification time of the on-media file is compared to the i-node modification time of corresponding file in the *destination* directory. The file is restored only if the on-media version is newer than the version in the *destination* directory. The i-node modification time of a file can be displayed with the *ls -lc* command.

–**I** Causes the *xfsdump* inventory to be displayed (no restore is performed). Each time *xfsdump* is used, an online inventory in */var/xfsdump/inventory* is updated. This is used to determine the base for incremental dumps. It is also useful for manually identifying a dump session to be restored (see the –**L** and –**S** options). Suboptions to filter the inventory display are described later.

–**L** *session_label*
Specifies the label of the dump session to be restored. The source media is searched for this label. It is any arbitrary string up to 255 characters long. The label of the desired dump session can be copied from the inventory display produced by the –**I** option.

–**S** *session_id*
Specifies the session UUID of the dump session to be restored. The source media is searched for this UUID. The UUID of the desired dump session can be copied from the inventory display produced by the –**I** option.

– A lone – causes the standard input to be read as the source of the dump to be restored. Standard input can be a pipe from another utility (such as *xfsdump*(1M)) or a redirected file. This option cannot be used with the –**f** option. The – must follow all other options, and precede the *destination* specification.

The dumped filesystem is restored into the *destination* directory. There is no default; the *destination* must be specified.

NOTES

Cumulative Restoration

A base (level 0) dump and an ordered set of delta dumps can be sequentially restored, each on top of the previous, to reproduce the contents of the original filesystem at the time the last delta was produced. The operator invokes *xfsrestore* once for each dump. The –**r** option must be specified. The *destination* directory must be the same for all invocations. Each invocation leaves a directory named *xfsrestorehousekeeping* in the *destination* directory (however, see the –**a** option above). This directory contains the state information that must be communicated between invocations. The operator must remove this directory after the last delta has been applied.

xfsrestore also generates a directory named *orphanage* in the *destination* directory. *xfsrestore* removes this directory after completing a simple restore. However, if *orphanage* is not empty, it will not be removed. This can happen if files present on the dump media are not referenced by any of the restored directories. The *orphanage* has an entry for each such file. The entry name is the file's original inode number.

xfsrestore does not remove the *orphanage* after cumulative restores. Like the *xfsrestorehousekeeping* directory, the operator must remove it after applying all delta dumps.

Media Management

A dump consists of one or more media files contained on one or more media objects. A media file contains all or a portion of the filesystem dump. Large filesystems are broken up into multiple media files to minimize the impact of media dropouts, and to accommodate media object boundaries (end-of-media).

A media object is any storage medium: a tape cartridge, a remote tape device (see *rmt*(1M)), a regular file, or the standard input (currently other removable media drives are not supported). Tape cartridges can contain multiple media files, which are typically separated by (in tape parlance) file marks. If a dump spans multiple media objects, the restore must begin with the media object containing the first media file dumped. The operator is prompted when the next media object is needed.

Media objects can contain more than one dump. The operator can select the desired dump by specifying the dump label (–**L** option), or by specifying the dump UUID (–**S** option). If neither is specified, *xfsrestore* scans the entire media object, prompting the operator as each dump session is encountered.

The inventory display (–**I** option) is useful for identifying the media objects required. It is also useful for identifying a dump session. The session UUID can be copied from the inventory display to the –**S** option argument to unambiguously identify a dump session to be restored.

Dumps placed in regular files or the standard output do not span multiple media objects, nor do they contain multiple dumps.

Inventory

Each dump session updates an inventory database in */var/xfsdump/inventory*. This database can be displayed by invoking *xfsrestore* with the –**I** option. The display uses tabbed indentation to present the inventory hierarchically. The first level is filesystem. The second level is session. The third level is media stream (currently only one stream is supported). The fourth level lists the media files sequentially composing the stream.

Several suboptions are available to filter the display. Specifying –**I depth=***n* (where *n* is 1, 2, or 3) limits the hierarchical depth of the display. Specifying –**I mobjid=***value* (where *value* is a media id) or –**I mobjlabel=***value* (where *value* is a media label) limits the display to media files contained in the specified media object. Similarly, the display can be restricted to a specific filesystem identified by mount point using –**I mnt=***host-qualified_mount_point_pathname*, by filesystem id using –**I fsid=***filesystem_id*, or by device using –**I dev=***host-qualified_device_pathname*. At most three suboptions may be specified at once: one to constrain the depth, one to constrain the media object, and one to constrain the filesystem. For example, –**I depth=1,mobjlabel="tape 1",mnt=host1:/test_mnt** would display only the filesystem information (depth=1) for those filesystems which were mounted on *host1:/test_mnt* at the time of the dump, and only those filesystems dumped to the media object labeled "tape 1".

There is currently no way to remove dumps from the inventory.

An additional media file is placed at the end of each dump stream. This media file contains the inventory information for the current dump session. This is currently unused.

Media Errors

xfsdump is tolerant of media errors, but cannot do error correction. If a media error occurs in the body of a media file, the filesystem file represented at that point is lost. The bad portion of the media is skipped, and the restoration resumes at the next filesystem file after the bad portion of the media.

If a media error occurs in the beginning of the media file, the entire media file is lost. For this reason, large dumps are broken into a number of reasonably sized media files. The restore resumes with the next media file.

FILES

/var/xfsdump/inventory	dump inventory database

SEE ALSO

rmt(1M), xfsdump(1M)

DIAGNOSTICS

The exit code is 0 on normal completion, and non-zero if an error occurred or the restore was terminated by the operator.

BUGS

There is no option to restore a specific *media* file contained in a media object.

Subtree options and interactive commands may be used to eliminate some files from the restore. *xfsrestore* does not know how to skip media files and media objects which do not contain selected files.

xfsrestore can only handle dumped filesystems with 8 million or less directory entries. This can be increased to 32 million directory entries by increasing the maximum value of rlimit_vmem_max in /var/sysgen/mtune/kernel to 0x7fffffff (see *mtune*(4), *systune*(1M)).

Pathnames of restored non-directory files (relative to the *destination* directory) must be 1023 characters or less. Longer pathnames are discarded and a warning message displayed.

There is no verify option to *xfsrestore*. This would allow the operator to compare a filesystem dump to an existing filesystem, without actually doing a restore.

Restores can not be resumed from the point of interruption. The entire restore must be restarted from the beginning.

xfsrestore restores the owner, group, and mode of each file and directory exactly. Thus, for example, files owned by root at the time of the dump are owned by root after the restoration.

xfsrestore must be run with root privilege.

Dumps cannot be read on fixed block size tape devices via the remote tape device interface.

The interactive commands (–**i** option) do not understand regular expressions.

Arguments to interactive commands (–**i** option) are pathnames, but are limited to just one level (for example, **ls foo** is allowed, but **ls foo/bar** is not).

NAME

xlv_admin – modifies XLV logical volume objects and their disk labels

SYNOPSIS

xlv_admin [–**r** root]

DESCRIPTION

xlv_admin is a menu-driven command that is used to modify existing XLV objects (volumes, plexes, volume elements, and XLV disk labels). *xlv_admin* can operate on XLV volumes even while they are mounted and in use.

xlv_admin supports a single command line option:

–**r** *root* Use *root* as the root directory. This is used in the miniroot when / is mounted as */root*.

The *xlv_admin* menu is:

```
**************** XLV Administration Menu **********
................ Add Existing Selections...........
1. Add a ve to an existing plex.
2. Add a ve at the END of an existing plex.
3. Add a plex to an existing volume.
................ Detach Selections................
11. Detach a ve from an existing plex.
12. Detach a plex from an existing volume.
................ Remove Selections................
21. Remove a ve from an existing plex.
22. Remove a plex from an existing volume.
................ Delete Selections................
31. Delete an object.
32. Delete all XLV disk labels.
................ Show Selections................
41. Show object by name and type, only.
42. Show information for an object.
................ Exit................
99. Exit
```

Note that the selections that refer to plexes (for example selection 3) are displayed only when your system has been licensed for the plexing portion of XLV.

xlv_admin provides five types of operations: **add**, **detach**, **remove**, **delete**, and **show**:

add The **add** operations allow you to add an XLV object to another XLV object. This allows you to, for example, add a plex to a volume. The plex or volume element to be added must first be created via *xlv_make*(1M). Note that *xlv_admin* refers to the larger object (the volume, in this case) as the ‘‘object to be operated on.’’

detach The **detach** operations allow you to separate a part of an XLV object and make it an independent XLV object. If you ''detach'' a plex from a plexed volume, for example, that plex would be separated from the volume and made into a standalone plex. The original volume would have one less plex.

remove The **remove** operations allow you to destroy a part of an XLV object. Removing plex number 1 from a volume with two plexes results in an XLV volume that has a single plex. The disk partitions that were part of the removed plex are no longer part of any XLV object.

delete The **delete** operations allow you to delete entire XLV objects.

show The **show** operations allow you to examine the list of XLV objects on the system and their structure.

The following are details on the various menu selections:

`1. Add a ve to an existing plex.`

Allows you to add a volume element to a gap in a plex. This is useful, for example, when you are replacing a faulty disk. *xlv_admin* prompts you for the name of the plex, the relative position within the plex to insert the volume element, and the name of the volume element that you want to add. (The first volume element in a plex is at position 0.)

The plex to be operated on can be a standalone plex or a part of a volume. If the plex is part of a volume, then the volume, subvolume, and plex must be specified. If the volume has only one plex then *xlv_admin* automatically uses that plex. The user can use an XLV name (for example, movies.data.0) to name the plex. If only a component of the name is given, then *xlv_admin* prompts for the remaining components. In all cases the ''object to be operated on'' is the plex. In the example below it is volume *test*. The following shows an example where we are inserting a volume element *ve5* to a gap in the volume *test*. (You can tell that there is a gap because the first volume element starts at block number 76200.) Note that we first display the configuration of *test* and *ve5* before we add *ve5* to *test*.

```
xlv_admin> 42
Please enter name of object to be operated on.
xlv_admin> test

============= Displaying Requested Object ==========
vol test
ve test.data.0.0 [active]
        start=76200, end=152399, (cat)grp_size=1
        /dev/dsk/dks0d2s1 (76200 blks)
ve test.data.0.1 [active]
        start=152400, end=228599, (cat)grp_size=1
        /dev/dsk/dks0d2s2 (76200 blks)


 Please select choice...
```

```
xlv_admin> 42
Please enter name of object to be operated on.
xlv_admin> ve5

============= Displaying Requested Object ==========
ve ve5 [empty]
        start=0, end=76199, (cat)grp_size=1
        /dev/dsk/dks0d2s5 (76200 blks)


 Please select choice...
xlv_admin> 1
Please enter name of object to be operated on.
xlv_admin> test
 Please enter ve number.
xlv_admin> 0
 Please enter the object you wish to add to the target.
xlv_admin> ve5
 Please select choice...
xlv_admin> 42
Please enter name of object to be operated on.
xlv_admin> test

============= Displaying Requested Object ==========
vol test
ve test.data.0.0 [stale]
        start=0, end=76199, (cat)grp_size=1
        /dev/dsk/dks0d2s5 (76200 blks)
ve test.data.0.1 [active]
        start=76200, end=152399, (cat)grp_size=1
        /dev/dsk/dks0d2s1 (76200 blks)
ve test.data.0.2 [active]
        start=152400, end=228599, (cat)grp_size=1
        /dev/dsk/dks0d2s2 (76200 blks)


 Please select choice...
xlv_admin>
```

`2. Add a ve at the END of an existing plex.`

Allows you to grow a volume by adding a volume element to the end of a plex. You can use this in conjunction with *xfs_growfs*(1M) to grow an XFS filesystem without having to unmount it.

Assuming that we have a volume element, *spareve*, that contains a single disk partition */dev/dsk/dks1d4s2*, the following sequense of commands adds it to the end of plex 0 of the data subvolume of volume *db1*:

```
 Please select choice...
xlv_admin> 2
 Please enter name of object to be operated on.
xlv_admin> db1.data.0
 Please enter the object you wish to add to the target.
xlv_admin> spareve

 Please select choice...
xlv_admin> 42
 Please enter name of object to be operated on.
xlv_admin> db1
vol db1
ve db1.data.0.0 [active]
 start=0, end=1100799, (cat)grp_size=1
 /dev/dsk/dks1d4s0 (1100800 blks)
ve db1.data.0.1 [active]
 start=1100800, end=2201599, (cat)grp_size=1
 /dev/dsk/dks1d4s1 (1100800 blks)
ve db1.data.0.2 [active]
 start=2201600, end=3302399, (cat)grp_size=1
 /dev/dsk/dks1d4s2 (1100800 blks)
```

3. Add a plex to an existing volume.

Allows you to add a plex to a volume. This allows you to create duplicate copies of the data on the volume for greater reliability. This operation is sometimes called *mirroring*. When you pick this selection, *xlv_admin* prompts you for the volume to add the plex to and the name of the plex. After the plex has been added, *xlv_admin* automatically initiates a plex revive operation; this copies the data from the original XLV plexes to the newly added plex so that the plex holds the same data as the original plexes in the volume. The following shows how to add a plex named *plex2* to the data subvolume of volume *db1*:

```
 Please select choice...
xlv_admin> 3
 Please enter name of object to be operated on.
xlv_admin> db1.data
 Please enter the object you wish to add to the target.
xlv_admin> plex2
```

You can use selection 42 to display volume *db1* and see that the disk partitions that were part of *plex2* are now a component of *db2*. Note that *plex2* no longer exists as a standalone plex since it has been merged into volume *db1*.

11. `Detach a ve from an existing plex.`

    Allows you to separate a volume element from a plex. This volume element can later be reinserted into some other XLV object. The plex from which the volume element is detached may be a standalone plex or part of a volume. The detached volume element remains an XLV object. The user first specifies the object from which the volume element will be detached and then the name to be given to the detached volume element.

    Note that detach and remove operations differ in how they handle the volume element once it has been separated from the plex. A detach operation leaves the volume element intact while the remove operation destroys the volume element by freeing its associated disks for use by other volumes. The detach operation may be thought of as an unlink. You should use either the detach or remove operation depending on whether you want the volume element to be left intact after it has been separated from its plex.

12. `Detach a plex from an existing volume.`

    Allows you to separate a plex from a volume. The user first specifies the volume and subvolume from which the plex is to be detached and then the name to assign to the newly created standalone plex. This plex can later be added back to a volume by choosing selection 3.

    The following example shows how to detach the first plex from a volume:

```
xlv_admin> 12
 Please enter name of object to be operated on.
xlv_admin> db1.data
 Please select plex number (0-3).
xlv_admin> 0
 Please enter name of new object.
xlv_admin> detplex0
 Please select choice...
xlv_admin>
```

21. `Remove a ve from an existing plex.`

    Allows you to separate a volume element from a plex and destroys the removed volume element. The following shows how you can remove the second volume element from a plex:

```
xlv_admin> 21
 Please enter name of object to be operated on.
xlv_admin> db1.data
 Please select plex number (0-3).
xlv_admin> 0
 Please enter ve number.
xlv_admin> 1
 Please select choice...
```

```
        xlv_admin>
```

22. `Remove a plex from an existing volume.`
    Allows you to separate a plex from a volume and destroys the removed plex.

31. `Delete an object.`
    Allows you to delete a volume, a standalone plex, or a standalone volume element. This operation removes the XLV configuration from the disk partitions that make up the XLV object. Because the XLV configuration information is stored in the volume header (see *vh*(7M)), this operation does not affect any user data that may have been written to the user disk partitions.

32. `Delete all XLV disk labels.`
    Allows you to delete the XLV configuration from all the disks on the system. You might want to do this, for example, to initialize all the disks on a new system to ensure that there are no leftover XLV configuration information on the disks. Note that this is a very dangerous operation. Deleting the disk labels destroys all of the XLV objects on the system.

41. `Show object by name and type, only.`
    Allows you to view all the XLV objects on the system. This command lists only the names and types of the XLV objects. The following shows what the output of this selection looks like:

```
 Please select choice...
xlv_admin> 41

==================== Listing Objects =============
Volume:     'root_vol'
Volume:     'db1'
Volume Element:   've12'
Plex:             'plex2'
```

42. `Show information for an object.`
    Allows you to see detailed information on an XLV object. It displays all the XLV parameters as well as the disk partitions that make up the object.

    In the example below, you can see that the volume named *db1* has one subvolume of type data that contains two plexes. The first plex has two volume elements, while the second plex only has one volume element. The first volume element in each plex covers the same range of disk blocks. For each volume element, *xlv_admin* displays the partitions that make up the volume element, the size of the partition, and the range of this volume's disk blocks that map to the volume element.

```
 Please select choice...
xlv_admin> 42
 Please enter name of object to be operated on.
xlv_admin> db1
vol db1
```

```
ve db1.data.0.0 [active]
 start=0, end=1100799, (cat)grp_size=1
 /dev/dsk/dks1d4s0 (1100800 blks)
ve db1.data.0.1 [active]
 start=1100800, end=2201599, (cat)grp_size=1
 /dev/dsk/dks1d4s1 (1100800 blks)
ve db1.data.1.0 [active]
 start=0, end=1100799, (cat)grp_size=1
 /dev/dsk/dks1d4s2 (1100800 blks)
```

Note that the *xlv_admin* operations are complete in that they modify the XLV disk labels and kernel as appropriate. If an operation is not successful, an error message is printed to the screen explaining the failure.

SEE ALSO

xlv_assemble(1M), xlv_make(1M), xlv_plexd(1M), xlv_shutdown(1M), xlv(7M)

NOTES

Note that the *xlv_admin* operations modify both the XLV disk labels and the kernel data structures as appropriate. This means that you do not need to run *xlv_assemble*(1M) for your changes to take effect. The only exception to this is selection 32, which affects only the disk labels.

xlv_admin automatically initiates plex revive operations (see *xlv_plexd*(1M)) as required when you add a new plex or when you add a volume element to a plexed volume.

You must be root to run *xlv_admin*.

NAME

xlv_assemble – initialize logical volume objects from disk labels

SYNOPSIS

xlv_assemble [–**h** name] [–**lnq**] [–**r** root] [–**tvKP**]

DESCRIPTION

xlv_assemble scans all the disks attached to the local system for logical volume labels. It assembles all the available logical volumes and generates a configuration data structure. *xlv_assemble* also creates the device nodes for all XLV volumes in */dev/dsk/xlv* and */dev/rdsk/xlv*. The kernel is then activated with the newly created configuration data structure. If necessary, *xlv_assemble* will also ask the *xlv_plexd*(1M) to perform any necessary plex revives.

xlv_assemble is automatically run on system startup from a script in the */etc/init.d/xlv* directory. By default, it is also automatically run after you run *xlv_make*(1M).

xlv_assemble supports the following options:

–**h** *name*	Use *name* as the local nodename. Every logical volume label contains a system nodename. See the –**l** option below.
–**l**	Assemble only those logical volumes that were created on this local system. Local logical volumes have the local nodename in their logical volume labels. The default is to assemble all logical volumes.
–**n**	Scan all disks for logical volume labels, but don’t save the logical volume configuration and don’t activate the kernel with this configuration.
–**q**	Proceed quietly and don’t display status messages after putting together the logical volume configuration.
–**r** *root*	Use *root* as the root directory. This is used in the miniroot when / is mounted as /root.
–**t**	Display terse status messages naming the logical volumes configured.
–**v**	Display verbose status messages about the logical volumes configured.
–**K**	Don’t activate the kernel with this logical volume configuration.
–**P**	Don’t initiate plex revives on the logical volumes configured.

FILES

/dev/dsk/xlv/...
/dev/rdsk/xlv/...
/dev/dsk/xlv_root
/dev/rdsk/xlv_root

SEE ALSO

xlv_admin(1M), xlv_labd(1M), xlv_make(1M), xlv_plexd(1M), xlv_shutdown(1M), xlvd(1M), xlv(7M)

NOTE

You must be root to run *xlv_assemble*.

NAME

xlv_labd, xlv_plexd, xlvd – logical volume daemons

SYNOPSIS

xlv_labd
xlv_plexd [–**m** #_subprocs] [–**b** blocksize] [–**w** sleep-interval]
    [–**v** verbosity] [–**h**]

DESCRIPTION

xlv_labd, *xlv_plexd*, and *xlvd* are logical volume daemons. *xlv_labd* and *xlv_plexd* reside in user process space and *xlvd* resides in kernel process space.

The XLV label daemon, *xlv_labd*, is a user process that writes logical volume disk labels. It is normally started during system restart. Upon startup, *xlv_labd* immediately calls into the kernel to wait for an action request from the kernel daemon, *xlvd*. When an action request comes, *xlv_labd* processes it and updates the appropriate volume disk labels. After completing the update, *xlv_labd* calls back into the kernel to wait for another request.

The XLV plex copy daemon, *xlv_plexd*, is a user process responsible for making all plexes within a subvolume consistent. The master *xlv_plexd* process is started at system startup time, with the –**m** option, and subsequently used when new plexes are added. It receives requests to revive plexes via the named pipe */etc/.xlv_plexd_request_fifo* and starts child processes to perform the actual plex copy.

–**m** *#_subprocs*	*#_subprocs* is the maximum number of subprocesses the master *xlv_plexd* process forks off at any given time.
–**b** *blocksize*	*blocksize* is the granularity of a single plex copy operation in blocks. The default is 128 blocks, which means XLV initiates a plex copy of 128 blocks, sleeps as indicated by the –**w** option (see below), then moves on to the next set of 128 blocks.
–**w** *sleep-interval*	*sleep-interval* is an arbitrary delay enforced at regular intervals while performing a plex copy in order to share available disk bandwidth. The default delay is 0.
–**v** *verbosity*	*verbosity* is the level of verbosity. The minimum is 0 and the maximum is 3. *xlv_plexd* writes its messages to syslog. The default *verbosity* is 2.
–**h**	Print the help message.

The XLV daemon, *xlvd*, is a kernel process that handles I/O to plexes and performs plex error recovery. When disk labels require updating, *xlvd* initiates an action request to *xlv_labd* to perform the disk label update. If there aren't multiple plexes, *xlvd* does not do anything.

NOTE

All three daemons are automatically started and do not need to be explicitly invoked.

FILES

/etc/.xlv_plexd_request_fifo

SEE ALSO

xlv(7M)

NAME

xlv_make – create logical volume objects

SYNOPSIS

xlv_make [–**f**] [–**v**] [–**A**] [input_file]

DESCRIPTION

xlv_make creates new logical volume objects by writing logical volume labels to the devices that are to constitute the volume objects. A volume object can be an entire volume, a plex, or a volume element. *xlv_make* allows you to create objects that are not full volumes so that you can maintain a set of spares.

xlv_make supports the following options:

–**f** Force *xlv_make* to create a *volume element* even if the partition type for the partition specified does not correspond with its intended usage. This is useful, for example, in converting *lv*(7M) volumes to *xlv*(7M) volumes. It is also used to allow creation of objects involving currently mounted partitions.

–**v** Verbose option. Causes *xlv_make* to generate more detailed output. Also, it causes *xlv_assemble*(1M) to generate output upon exit from *xlv_make*.

–**A** Do not invoke *xlv_assemble*(1M) upon exit from *xlv_make*. The default is to invoke *xlv_assemble* with the –**q** option unless the –**v** option is specified, in which case *xlv_assemble* is invoked with no options. To invoke other *xlv_assemble* options, specify the –**A** option and invoke *xlv_assemble* manually.

xlv_make only allows you to create volume objects out of disk partitions that are not currently part of other volume objects. Partitions must be of a type suitable for use by *xlv_make.* Suitable types are **xfs**, **efs**, **xlv**, and **xfslog**. Partition types other than these will be rejected unless the –**f** command line option or the **ve** –**force** interactive command is specified. See *fx*(1M) for more information regarding partition types. *xlv_admin*(1M) must be used to modify or destroy volume objects.

xlv_make can be run either interactively or it can take its commands from an input file, *input_file.* *xlv_make* is written using Tcl. Therefore, all the Tcl features such as variables, control structures, and so on can be used in *xlv_make* commands.

xlv_make creates volume objects by writing the disk labels. To make the newly created logical volumes active, *xlv_assemble*(1M) must be run. *xlv_assemble* is, by default, automatically invoked upon successful exit from *xlv_make*; *xlv_assemble* scans all the disks attached to the system and automatically assembles all the available logical volumes.

Objects are specified top-down and depth-first. You start by specifying the top-level object, and continue to specify the pieces that make it up. When you have completed specifying an object at one level, you can back up and specify another object at the same level.

The commands are:

vol *volume_name*
: Specifies a volume. The *volume_name* is required. It can be up to 14 characters in length.

log Specifies a log subvolume.

data Specifies a data subvolume.

rt Specifies a real-time subvolume. Real-time subvolumes are used for guaranteed-rate I/O and also for high performance applications that isolate user data on a separate subvolume.

plex [*plex_name*]
: Specifies a plex. If this plex is specified outside of a volume, then *plex_name* must be given. A plex that exists outside of a volume is known as a standalone plex.

ve [*volume_element_name*] [–**stripe**] [–**concat**] [–**force**] [–**stripe_unit** *stripe_unit_size*] [–**start** *blkno*] *device_pathnames*
: Specifies a volume element. If this volume element is specified outside of a plex, then *volume_element_name* must be given.

–**stripe** Specifies that the data within this volume element will be striped across all the disks named by *device_pathnames.*

–**concat** Specifies that all the devices named by *device_pathnames* are to be joined linearly into a single logical range of blocks. This is the default if no flags are specified.

–**force** Forces the specification of the volume element when the partition type does not agree with the volume element's intended usage. For example, a partition with type ''xfslog'' could be assigned to a data subvolume. Also, –**force** allows the specification of an object that includes a partition that is currently mounted.

–**stripe_unit** *stripe_unit_size*
: specifies the number of blocks to write to one disk before writing to the next disk in a stripe set. *stripe_unit_size* is expressed in 512-byte blocks. –**stripe_unit** is only meaningful when used in conjunction with –**stripe**. The default stripe unit size, if this flag is not set, is one track. Note: *lv* called this parameter the granularity.

–**start** *blkno* Specifies that this volume element should start at the given block number within the plex.

end Terminates the specification of the current object.

clear Removes the current, uncompleted object.

show Prints out all the volume objects on the system. This includes existing volume objects (created during an earlier *xlv_make* session) and new objects specified during this session that have not been created (written out to the disk labels) yet.

exit Create the objects specified during this session by writing the disk labels out to all the disks affected, and exit *xlv_make*. In interactive mode, the user will be prompted to confirm this action if any new objects have been created.

quit Leave *xlv_make* without creating the specified objects (without writing the disk labels). All the work done during this invocation of *xlv_make* will be lost. In interactive mode, the user is prompted to confirm this action if any objects have been specified.

help Displays a summary of *xlv_make* commands.

? Same as **help**.

sh Fork a shell.

EXAMPLES

Example 1

To make a volume from a description in an input file called *volume_config.txt*, give this command:

```
# xlv_make volume_config.txt
```

Example 2

This example shows making some volume objects interactively.

```
# xlv_make
```

Make a spare plex so we can plug it into another volume on demand.

```
xlv_make> plex spare_plex1
spare_plex1
xlv_make> ve /dev/dsk/dks0d2s1 /dev/dsk/dks0d2s2
spare_plex1.0
xlv_make> end
Object specification completed
```

Now make a small volume. (Note that *xlv_make* automatically adds a */dev/dsk* to the disk paritition name if it is missing from the *ve* command.)

```
xlv_make> vol small
small
xlv_make> log
small.log
xlv_make> plex
small.log.0
xlv_make> ve dks0d2s3
small.log.0.0
xlv_make> data
small.data
xlv_make> plex
small.data.0
```

```
xlv_make> ve dks0d2s14 dks0d2s12
small.data.0.0
xlv_make> end
Object specification completed
xlv_make> show
vol small
ve small.log.0.0        d710aa7d-b21d-1001-868d-080069077725
  start=0, end=1523, (cat)grp_size=1
  /dev/dsk/dks0d2s3 (1524 blks)   d710aa7e-b21d-1001-868d-080069077725
ve small.data.0.0       d710aa81-b21d-1001-868d-080069077725
  start=0, end=4571, (cat)grp_size=2
  /dev/dsk/dks0d2s14 (1524 blks)  d710aa82-b21d-1001-868d-080069077725
  /dev/dsk/dks0d2s12 (3048 blks)  d710aa83-b21d-1001-868d-080069077725
plex spare_plex1
ve spare_plex1.0        d710aa77-b21d-1001-868d-080069077725
  start=0, end=3047, (cat)grp_size=2
  /dev/dsk/dks0d2s1 (1524 blks)   d710aa78-b21d-1001-868d-080069077725
  /dev/dsk/dks0d2s2 (1524 blks)   d710aa79-b21d-1001-868d-080069077725

xlv_make> help
vol volume_name  - Create a volume.
data | log | rt  - Create subvolume of this type.
plex [plex_name] - Create a plex.
ve [-start] [-stripe] [-stripe_unit N] [-force] [volume_element_name] partition(s)
end  - Finished composing current object.
clear- Delete partially created object.
show - Show all objects.
exit - Write labels and terminate session.
quit - Terminate session without writing labels.
help or ? - Display this help message.
sh - Fork a shell.

xlv_make> exit
#
```

Note that the strings like *d710aa82-b21d-1001-868d-080069077725* shown above are the universally unique identifiers (UUIDs) that identify each XLV object.

Example 3

This example shows a description file that makes the same volume objects as in Example 2.

```
# A spare plex
plex spare_plex1
ve dks0d2s1 dks0d2s2
# A small volume
```

```
vol small
log
plex
ve dks0d2s3
data
plex
ve dks0d2s14 dks0d2s12
end
# Write labels before terminating session.
exit
```

Example 4

This example shows making a complex volume interactively. It makes a volume for an XFS filesystem that has a single-partition log and a plexed (mirrored) data subvolume that is striped.

```
# xlv_make
xlv_make> vol movies
movies
xlv_make> log
movies.log
xlv_make> plex
movies.log.0
xlv_make> ve /dev/dsk/dks0d2s1
movies.log.0.0
```

Let the data subvolume have two plexes, each of which consists of two sets of striped disks. The data written to the data subvolume will be copied to both movies.data.0 and movies.data.1.

```
xlv_make> data
movies.data
xlv_make> plex
movies.data.0
xlv_make> ve -stripe dks0d1s6 dks0d2s6 dks0d3s6
movies.data.0.0
xlv_make> ve -stripe dks0d4s6 dks0d5s6
movies.data.0.1
xlv_make> plex
movies.data.1
xlv_make> ve -stripe dks1d1s6 dks1d2s6 dks1d3s6
movies.data.1.0
xlv_make> ve -stripe dks1d4s6 dks1d5s6
movies.data.1.1
```

Add a small real-time subvolume. Stripe the data across two disks, with the stripe unit set to 1024 512-byte sectors.

```
xlv_make> rt
movies.rt
xlv_make> plex
movies.rt.0
xlv_make> ve -stripe -stripe_unit 1024 dks4d1s6 dks4d2s6
movies.rt.0.0
xlv_make> end
Object specification completed
xlv_make> exit
#
```

DIAGNOSTICS

`Previous object not completed`
: You have tried to specify a new object before the previous object has been completely specified. For example, the sequence **plex plex** is not valid because the volume elements for the first plex have not been specified yet.

`A volume has not been specified yet`
: This error results from giving **rt**, **data**, or **log** without first specifying a volume to which these subvolumes belong.

`An object with that name has already been specified`
: This error results from giving the **vol** *volume_name*, **plex** *plex_name*, or **ve** *volume_element_name* command when an object with the same name already exists or has been specified in this session.

`A log subvolume has already been specified for this volume`

`A data subvolume has already been specified for this volume`

`A real-time subvolume has already been specified for this volume`
: These errors results from giving the **log**, **data**, or **rt** command for a volume that already has a subvolume of the given type.

`A subvolume has not been specified yet`
: You have given a **volume** command and then given the **plex** command without first specifying a subvolume to which the plex belongs.

`Too many plexes have been specified for this subvolume`
: You have already specified the maximum allowable number of plexes for this subvolume.

`A plex has not been specified yet`
: You have given a **ve** command without first giving the **plex** command.

`Too many volume elements have been specified for this plex`
: You have reached the maximum number of volume elements that can be in a single plex.

`An error occurred in creating the specified objects`
: An error occurred while writing the volume configuration out to the disk labels.

`Unrecognized flag:` *flag*
: *flag* is not recognized.

`Unexpected symbol:` *symbol*
: *symbol* is an unknown command.

`A volume name must be specified`
: You have given a **vol** command without giving the name of the volume as an argument.

`Too many disk partitions`
: You have specified too many devices for the volume element.

`Cannot determine size of` *partition*`; please verify that the device exists`
: *xlv_make* is unable to figure out the size of the specified disk partition. Make sure that the device exists.

`Unequal partition sizes, truncating the larger partition`
: The partitions specified for a striped volume element are not of the same size. This leaves some disk space unusable in the larger partition because data is striped across all the partitions in a volume element.

`A disk partition must be specified`
: You have given the **ve** command without specifying the disk partitions that belong to the volume element as arguments to the command.

`Unknown device: %s`
: You have specified a disk partition that either has no device node in */dev/dsk* or is missing altogether.

`Illegal value`
: The value is out of range for the given flag.

`The volume element's address range must be increasing`
: When you specify the starting offset of a volume element within a plex by using the **ve** –**start** command, you must specify them in increasing order.

`Disk partition` *partition* `is already being used`
: The disk partition named in the **ve** command is already in use by some other volume object.

`Disk partition` *partition* `is mounted; use ''–force'' to override`
: The disk partition named in the **ve** command is currently mounted. Use of the –**force** argument is required to perform the operation.

`Address range doesn’t match corresponding volume element in other plexes`
: A volume element within a plex must have the same address range in all plexes for the subvolume that includes those plexes.

`There are partially specified objects, use ‘‘quit’’ to exit without`
: creating them You have entered the **quit** command while there are specified, but not created objects. You should enter **quit** again to really quit at this point and discard specified objects.

`Missing flag value for: %s`
: A command was given that requires an additional argument that was not given.

`Malloc failed`
: There is insufficient memory available for *xlv_make* to operate successfully.

`An error occurred in updating the volume header`
: An attempt to modify a disk’s volume header was unsuccessful.

`A striped volume element must have at least two partitions`
: The **ve –stripe** command was given and only one partition was specified.

`Log ve should have partition type xfslog`

`Data ve should have partition type xlv`

`Rt ve should have partition type xlv`

`Standalone object should have partition type xlv or xfslog`

`Mixing partition type xfslog with data types not allowed`
: All the paritions that make up a volume element must have the same partition type, either xlv or xfslog.

`Partition type must be consistent with other ve’s in plex`
: Partition type does not correspond with intended usage.

`Partition could already belong to lv.`
: Check /etc/lvtab A warning that this partition may already belong to an *lv* volume.

`Illegal partition type`
: An attempt was made to specify a partition that cannot, under any circumstance, be used in an *xlv*(7M) volume. An example of such a partition would be the volume header.

`Subvolume type does not match any known`
: The subvolume being operated on is of no known type.

`Size mismatch`
: The partition size information in the volume header does not match that contained in the xlv label.

`Device number mismatch`

A warning that the device number in the xlv label does not match that of the volume header.

`The same partition cannot be listed twice`

The **ve** command was given with the same partition listed twice.

SEE ALSO

xlv_admin(1M), xlv_assemble(1M), xlv_labd(1M), xlv_plexd(1M), xlv_shutdown(1M), xlvd(1M), xlv(7M)

Tcl and the Tk Toolkit by John K. Ousterhout, Addison-Wesley, 1994.

NOTES

The disk labels created by *xlv_make* are stored only in the volume header of the disks. They do not destroy user data. Therefore, you can make an *lv*(7M) volume into an XLV volume and still preserve all the data on the logical volume.

xlv_make changes the partition type of partitions used in newly created objects to either *xlv* or *xfslog* depending upon their usage.

You must pick a different name for each volume, standalone plex, and standalone volume element. You cannot have, for example, both a volume and a plex named *yy*.

You must be root to run *xlv_make*.

NAME

xlv_set_primary – set the primary plex of a logical volume

SYNOPSIS

xlv_set_primary device_name

DESCRIPTION

xlv_set_primary finds the XLV volume and plex to which *device_name* belongs and makes that plex the active copy. All the other plexes that belong to this volume are marked stale. This causes all of the plexes in this volume to be synchronized to the contents of the active plex when the volume is later assembled by *xlv_assemble*(1M).

xlv_set_primary is designed for use during the miniroot when only a single plex of the volume is running. Making that plex the primary plex of the volume ensures that whatever changes are made to this plex (for example, installing software) are made to the other plexes when they come online.

This command has no effect if *device_name* is not part of an XLV volume.

SEE ALSO

xlv_admin(1M), xlv_assemble(1M), xlv_plexd(1M), xlv(7M)

NOTE

You must be root to run *xlv_set_primary*.

NAME

xlv_shutdown – shutdown XLV volumes

SYNOPSIS

xlv_shutdown [–**v**] [–**n** volume-name]

DESCRIPTION

xlv_shutdown is used to gracefully shut down (‘‘disassemble’’) logical volumes after their corresponding filesystems have been unmounted. It is called by */etc/umountfs*, which is called by */etc/inittab* at system shutdown time. *xlv_shutdown* typically does not need to be explicitly invoked.

xlv_shutdown gets the XLV volumes from the kernel and cleanly shuts them down. This ensures that all the plexes in a volume are in sync so that they do not need to be revived when restarted. After a volume has been shut down, *xlv_assemble*(1M) needs be run before using the volume again. Note that *xlv_shutdown* does not shut down a root volume or volumes with mounted filesystems.

xlv_shutdown supports the following options:

–**n** *volume-name*	Shut down only the given volume. The default behavior is to close down all possible volumes.
–**v**	Display verbose status messages.

SEE ALSO

shutdown(1M), xlv_assemble(1M), inittab(4), xlv(7M)

NOTE

You must be root to run *xlv_shutdown*.

NAME

grio_config – description of device I/O rates

DESCRIPTION

The */etc/grio_config* file contains information describing the I/O rates for each device or controller in the system. This information is read by *ggd* and is used to allocate I/O-rate guarantees to requesting processes.

The *grio_config* file is composed of entries of two different types. The first describes a system element and its bandwidth:

device_name`= OPTSZ=# NUM=# CTLRNUM=# UNIT=# RT=1` (*comment*)

OPTSZ, **NUM**, **CTLRNUM**, and **UNIT** are keywords. **OPTSZ** refers to the optimal I/O size of the device in bytes, and **NUM** is the number of **OPTSZ** sized I/O requests that can be guaranteed each second. These fields are required for each device. **CTLRNUM** and **UNIT** refer to the device SCSI controller and unit respectively. These are used only when necessary to identify a particular device. **RT=1** is used to indicate that the disk device is part of an XLV real-time subvolume and that the error retry mechanism should be disabled. The comment field is optional, but usually contains a description of the device.

The second type of entry describes the relationship between elements in the system:

device_name: *dev1 dev2 dev3*

This means that *dev1*, *dev2*, and *dev3* are attached to *device_name.* In order to get a rate guarantee on one of these devices, a rate guarantee must also be obtained on *device_name* as well.

With these entries, *ggd* is able to construct a performance tree. This tree is used to determine if an I/O-rate-guarantee request can be satisfied.

FILES

/etc/grio_config

SEE ALSO

cfg(1M), ggd(1M), grio.disks(4), grio(5), xlv(5)

NOTES

Currently, all devices have **OPTSZ** set to 64K bytes. If a device has **OPTSZ** and **NUM** values of 0, the I/O characteristics of the device could not be determined, and the device is not considered when making rate guarantees.

NAME

grio_disks – description of guaranteed I/O rates for disk drives

DESCRIPTION

The */etc/grio_disks* file contains information describing the I/O rates for individual types of disk drives.

The entries are of the form:

```
ADD "SGIxxxxxxxxxxxxxxxxxxxxxxxxx"    #     #
```

The first item is the key word **ADD**. The next item is a 28 character string describing the type of disk drive. This is the same as the disk drive ID string. Drives recommended by Silicon Graphics usually have the "SGI" string as the first characters in this string. The next number describes the optimal I/O size in bytes for the disk device. The final number is the number of optimal sized I/O requests that can be performed by the disk drive each second.

The performance characteristics for most supported disk drives are already known by the *ggd* daemon. This file is used to allow system administrators to add the characteristics of new types of drives so that *ggd* can make reliable guarantees.

FILES

/etc/grio_disks

SEE ALSO

cfg(1M), ggd(1M), grio.config(4), grio(5)

NOTE

The number of optimal sized I/O requests that can be guaranteed each second may be significantly less than the maximum performance of the drive. This is because each request is considered to be distinct and may require a maximum length seek before the request is issued.

NAME

xfs – layout of the XFS filesystem

DESCRIPTION

An XFS filesystem can reside on a regular disk partition or on a logical volume (see *lv*(7M) and *xlv*(7M)). An XFS filesystem has up to three parts: a data section, a log section, and a real-time section. For disk partition and *lv* logical volume filesystems, the real-time section is absent, and the log area is contained within the data section. For XLV logical volume filesystems, the real-time section is optional, and the log section can be separate from the data section or contained within it. The filesystem sections are divided into a certain number of *blocks*, whose size is specified at *mkfs*(1M) time with the –**b** option.

The data section contains all the filesystem metadata (inodes, directories, indirect blocks) as well as the user file data for ordinary (non-real-time) files and the log area if the log is *internal* to the data section. The data section is divided into a number of *allocation groups*. The number and size of the allocation groups are chosen by *mkfs* so that there is normally a small number of equal-sized groups. The number of allocation groups controls the amount of parallelism available in file and block allocation. It should be increased from the default if there is sufficient memory and a lot of allocation activity. More allocation groups are added (of the original size) when *xfs_growfs*(1M) is run.

The log section (or area, if it is internal to the data section) is used to store changes to filesystem metadata while the filesystem is running until those changes are made to the data section. It is written sequentially during normal operation and read only during mount. When mounting a filesystem after a crash, the log is read to complete operations that were in progress at the time of the crash.

The real-time section is used to store the data of real-time files. These files had an attribute bit set through *fcntl*(2) after file creation, before any data was written to the file. The real-time section is divided into a number of *extents* of fixed size (specified at *mkfs* time). Each file in the real-time section has an extent size that is a multiple of the real-time section extent size.

Each allocation group contains several data structures. The first sector contains the superblock. For allocation groups after the first, the superblock is just a copy and is not updated after *mkfs*. The next three sectors contain information for block and inode allocation within the allocation group. Also contained within each allocation group are data structures to locate free blocks and inodes; these are located through the header structures.

All these data structures are subject to change, and the headers that specify their layout on disk are not provided.

SEE ALSO

mkfs(1M), xfs_growfs(1M), fcntl(2), syssgi(2), lv(7M), xlv(7M)

NAME

grio – guaranteed-rate I / O

DESCRIPTION

Guaranteed-rate I / O (GRIO) refers to a guarantee made by the system to a user process indicating that the given process will receive data from a peripheral device at a predefined rate regardless of any other activity on the system. The purpose of this mechanism is to manage the sharing of scarce I / O resources amongst a number of competing processes, and to permit a given process to reserve a portion of the system’s resources for its exclusive use for a period of time.

Currently, the only I / O resources that can be reserved using the GRIO mechanism are files stored on the real-time subvolume of an XFS filesystem.

A GRIO guarantee is defined as the number of bytes that can be read or written to a given file by a given process, each second. If a process has a GRIO guarantee on a file and it issues I / O requests in sizes equal to the guaranteed amount, then the read or write calls are guaranteed to complete in less than one second. If the process issues I / O requests at a size or rate greater than the guarantee, the excess requests are blocked until such time as they fall within the scope of the guarantee.

There are a number of components in the GRIO mechanism. The first is the guarantee-granting daemon, *ggd.* This is a user level process that is started when the system is booted. It controls the granting of guarantees, the initiation and expiration of existing guarantees, and the monitoring of the available bandwidths of each I / O device on the system. User processes communicate with the daemon using the *grio_request*(3X), *grio_remove_request*(3X), *grio_get_rtgkey*(3X), and *grio_use_rtgkey*(3X) library calls.

When *ggd* is started, it reads the files */etc/grio_config* and */etc/grio_disks* to determine the bandwidths of the various devices on the system. These files are generated by the *cfg* utility but may be edited by the system administrator to tune performance. If *ggd* is terminated, all existing rate guarantees are removed.

The next component of the GRIO mechanism is the XLV volume manager. Rate guarantees may only be obtained from files on the real-time subvolume of an XFS filesystem. The disk driver command retry mechanism is disabled on the disks that make up the real-time subvolume. This means that if a drive error occurs, the data is lost. The intent of real-time files is to read / write data from the disk as rapidly as possible. If the device driver is forced to retry one process’s disk request, it causes the requests from other processes to become delayed.

If one partition of a disk is used in a real-time subvolume, the entire disk is considered to be used for real-time operation. If one disk on a SCSI controller is used for real-time operation then all the other devices on that controller must be used for real-time operation as well.

In order to use the guaranteed-rate I / O mechanism effectively, the XLV volume and XFS filesystem must be set up properly. The next section gives an example.

By default, the *ggd* daemon will allow two process streams to obtain rate guarantees. If support for more streams is desired, it is necessary to obtain licenses for the additional streams. The license information is stored in the */usr/var/netls/nodelock* file and interpreted by the *ggd* daemon on startup.

EXAMPLE

The example in this section describes a method of laying out the disks, filesystem, and real-time file that enables the greatest number of processes to obtain guarantees on a single file concurrently. It is not necessary to construct a file in this manner in order to use GRIO, however fewer processes can obtain rate guarantees on the file as a result. Assume that there are four disk partitions available for the real-time subvolume of an XLV volume. Each one of the partitions is on a different physical disk.

Before setting up the XFS filesystem, the I/O request size used by the user process must be determined. In order to get the greatest I/O rate, the file data should be striped across all the disks in the subvolume. To avoid filesystem fragmentation and to force all I/O operations to be on stripe boundaries, the file extent size should be an even multiple of the volume stripe width. Rate guarantees are always made assuming I/O request sizes that are even multiples of an optimal I/O size. The optimal I/O size is specified on a per device basis in the */etc/grio_config* file but it is usually 64K bytes. Therefore, the I/O request size should be a multiple of 64K bytes and equal to the volume stripe width. The file extent size should be set to a multiple of the volume stripe width.

In this example, let the file extent size be equal to the stripe width. The application always issues I/O operations of size equal to the extent size. Assuming there are four disks available, let the stripe step size be equal to 64k bytes. The file extent size and volume stripe width are set to 256K bytes. All application I/O operations then will be performed in 256k byte blocks.

Once the XLV volume and XFS filesystem have been created, the application can create the real-time file. Real-time files must be read or written using direct, synchronous I/O requests. The *open*(2) manual page describes the use and buffer alignment restrictions when using direct I/O. When creating a real-time file, the **F_FSSETXATTR** command must be issued to set the **XFS_XFLAG_REALTIME** flag. This can only be issued on a newly created file. It is not possible to mark a file as real-time once non-real-time data blocks have been allocated to it. Rate guarantees cannot be obtained when creating a file. In order for a rate guarantee to be obtained, it is necessary to know the layout of the blocks of the file on the disks. This cannot be determined until after the file has been written.

After the real-time file has been created, the application can issue a *grio_request*(3X) to obtain the rate guarantee. With the rate guarantee established, the application read or write requests to the file, using the given file descriptor, will complete within the guaranteed time. This will continue until the file is closed, the guarantee is removed by the application via *grio_remove_request*(3X), or the guarantee expires.

DIAGNOSTICS

If a rate cannot be guaranteed, *ggd* returns an error to the requesting process. It also returns the amount of bandwidth currently available on the device. The process can then determine if this amount is sufficient and if so issue another rate guarantee request.

FILES

/etc/grio_config
/etc/grio_disks
/usr/var/netls/nodelock

SEE ALSO

ggd(1M), grio_get_rtgkey(3X), grio_remove_request(3X), grio_request(3X), grio_use_rtgkey(3X), grio_config(4), grio_disks(4)

NAME

xlv – logical volume disk driver

SYNOPSIS

/dev/dsk/xlv/*
/dev/rdsk/xlv/*

DESCRIPTION

XLV devices provide access to disk storage as *logical volumes*. A logical volume is an object that behaves like a disk partition, but its storage may span several physical disk devices.

Using XLV, you can concatenate disks together to create larger logical volumes, stripe data across disks to create logical volumes with greater throughput, and plex (or mirror) disks for reliability. In addition, XLV enables you to change the configuration of volumes while the volume is actively being used as a filesystem.

The geometry of logical volumes (e.g., the disks that belong to it, how they are put together, etc.) are stored in the disk labels of the disks that belong to the logical volumes. When the system starts up, the utility *xlv_assemble*(1M) scans all the disks on the system and automatically assembles them into logical volumes. *xlv_assemble*(1M) also creates any necessary device nodes.

XLV device names always begin with **/dev/{r}dsk/xlv/***device_name* where the *device_name* is assigned by the creator of the volume. See *xlv_make*(1M) for how volumes are created.

Device numbers range from 0 to one less than the maximum number of logical volume devices configured in the system. This is 10 by default; this number may be changed by rebuilding a kernel with *lboot*(1M).

There is a kernel driver, referred to as *xlv*, and some daemons for the logical volume devices. The driver is a 'pseudo device' not directly associated with any physical hardware; its function is to map requests on logical volume devices into requests on the underlying disk devices. The daemons take care of error recovery and dynamic reconfiguration of volumes.

Volume Objects

XLV allows you to work with whole volumes and pieces of volumes. Pieces of volumes are useful for creating and reconfiguring volumes in units that are larger than individual disk partitions.

Each *volume* consists of up to three *subvolumes*. An *xfs*(4) filesystem usually has a large *data* subvolume in which all the user files and metadata such as inodes are stored and a small *log* subvolume in which the filesystem log is stored. For high-performance and real-time applications, a volume can also have a *real-time* subvolume that contains only user files aligned at configurable block boundaries. Guaranteed rate I/O can be done to real-time subvolumes. See *grio*(5).

Each subvolume can be independently organized as 1 to 4 *plexes*. Plexes are sometimes known as mirrors. XLV makes sure that the data in all the plexes of a subvolume are the same. Plexes are useful for reliability since a subvolume remains available if any of its plexes are available. Since each subvolume is independently organized, you can choose to plex any, all, or none of the subvolumes within a volume.

Each plex consists of up to 128 *volume elements*. Each volume element is a collection of *disk partitions* that may be either striped or concatenated. By adding volume elements, you can extend the size of a subvolume – even one that is striped. Volume elements within a plex do not need to be of the same size. However, all the volume elements at the same offset in all the plexes of the subvolume must be the same size. For example, the first and second volume elements in a plex can have different sizes. But the first volume element in all the plexes of the subvolume must be the same size. This restriction is necessary because the volume element is the unit of recovery. Note that if XLV gets an unrecoverable disk error on one disk partition in a volume element, the entire volume element is taken offline.

Each volume element can consist of from 1 to 100 disk partitions. The disks can be treated as either a *concatenated set* (in which case XLV writes to the partitions sequentially) or as a *striped set* (in which case XLV writes a stripe unit's worth of data to one disk and then rotates to the next disk in the stripe set.) In general, it is better to use volume elements that contain single disks when you want to concatenate disks together and only use volume elements with multiple disks when you want to use disk-striping. This is because the volume element is the unit of recovery.

XLV allows you to create and work with volumes, subvolumes, plexes, and volume elements. The interesting operations associated with volumes are: creating them, assembling disk partitions into volumes, mounting them, changing volume configurations, shutting them down, and destroying them.

Naming Volume Objects

Each XLV object is composed of a hierarchy of lower level objects. For example, a volume is composed of subvolumes that are in turn composed of plexes, etc. To let you refer to a component of an XLV object, XLV has adopted a hierarchical naming convention. For example:

movies.data.0.5.50	Refers to the volume named *movie*, the *data* subvolume, plex *0* of that subvolume, volume element *5* within that plex, and disk partition *50* within that volume element. Note that the numbers are zero-based.
movies.log.2	Refers to plex number 2 in the log subvolume of the volume named *movies*.
movies.rt.1.5	Refers to volume element 5 within plex number 1 of the real-time subvolume of the volume named *movies*.

If you create an object outside of a volume, then that object has a user-assigned name. For example, **spare_plex.2.1** refers to disk partition number 1 of volume element number 2 of a standalone plex named *spare_plex*. *spare_plex* does not currently belong to any subvolumes.

These names are echoed by *xlv_make*(1M) as objects are created. They are also useful in specifying the objects to change via *xlv_admin*(1M).

Creating Volumes

Volumes are created via *xlv_make*(1M). This utility writes the volume geometry to all the disks that belong to the volume object. The geometry is written to the volume headers. See *vh*(7M).

Assembling Volumes

After a volume has been created, it must be made known to the kernel driver before I/O can be initiated to the volume. The command *xlv_assemble*(1M) scans all the disks attached to the system and assembles all the logical volumes that it finds. It then passes the configuration to the kernel. This is usually done during system startup. Once a volume has been assembled, I/O can be performed.

Working with Filesystems

The normal filesystem utilities such as *mkfs*(1M) and *mount*(1M) work with logical volumes.

A logical volume consisting of a single disk partition (that may be plexed) can be used as *root*(7M). You cannot boot directly off a logical volume; you must specify the underlying disk partition. partition.

Modifying Volumes

The geometry of a volume object can be modified either offline or online. To modify a volume object offline, first unmount the filesystem, then destroy the volume object by using *xlv_admin*(1M). Then, you can run *xlv_make*(1M) to create new XLV objects. Note that *xlv_make* only allows you to use disk partitions that are not currently part of volume objects.

You can also modify volume objects while they are online by using *xlv_admin*(1M). You can grow a volume, add a plex, and remove a plex while the volume is actively being used. Note that I/O is blocked while the configuration is being changed. The blocked I/O is completed after the configuration has been written out to the disk labels.

You can also use *xlv_admin* to remove a volume element from a plex while the volume is online if there is at least one other plex that covers the range of disk blocks affected. Note that you can choose to plex only a portion of the address space of a subvolume.

Working with Plexes

When there are multiple plexes, XLV recovers from read errors. In addition, XLV attempts to rewrite the data back to the failed plex. XLV masks write errors if it can write to at least one of the plexes.

When a plexed volume starts up, XLV automatically makes sure that all the data among the plexes within each subvolume is consistent. This may involve copying the data from one plex to the others. While this is going on, the volume is available at a degraded performance. You can eliminate the need for plex recovery by shutting down the plex with *xlv_shutdown*(1M). *xlv_shutdown* synchronizes the plexes and marks them as been the same so that when they restart, XLV knows that the plexes are consistent and can therefore avoid the plex copies.

FILES

/dev/dsk/xlv/*
/dev/rdsk/xlv/*
/var/sysgen/master.d/xlv

SEE ALSO

cfg(1M), lv_to_xlv(1M), xlv_admin(1M), xlv_assemble(1M), xlv_make(1M), xlv_shutdown(1M), grio(5), xlv_labd(1M), xlv_plexd(1M), xlvd(1M)

NOTES

XLV runs on both XFS and EFS filesystems. In addition, you can read and write to XLV devices using the raw device interfaces.

XLV disk labels are stored on the disks themselves. Therefore, you can physically reposition the disk drives and XLV still assembles them correctly.

You can upgrade from an existing *lv*(7M) volume to an XLV volume by using *lv_to_xlv*(1M).

When you are running in the miniroot, the XLV device nodes are created in */root/dev/dsk/xlv* and */root/dev/rdsk/xlv* .

Index

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2549-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389