

VERITAS File System 4.1

Programmer's Reference Guide

Solaris

N16156F

March 2005

Disclaimer

The information contained in this publication is subject to change without notice. VERITAS Software Corporation makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. VERITAS Software Corporation shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this manual.

VERITAS Legal Notice

Copyright © 2005 VERITAS Software Corporation. All rights reserved. VERITAS, the VERITAS Logo, and Storage Foundation are trademarks or registered trademarks of VERITAS Software Corporation or its affiliates in the U.S. and other countries. Other names may be trademarks of their respective owners.

VERITAS Software Corporation
350 Ellis Street
Mountain View, CA 94043
USA
Phone 650-527-8000 Fax 650-527-2908
www.veritas.com

Third-Party Legal Notices

Data Encryption Standard (DES) Copyright

Copyright © 1990 Dennis Ferguson. All rights reserved.

Commercial use is permitted only if products that are derived from or include this software are made available for purchase and/or use in Canada. Otherwise, redistribution and use in source and binary forms are permitted.

Copyright 1985, 1986, 1987, 1988, 1990 by the Massachusetts Institute of Technology. All rights reserved.

Export of this software from the United States of America may require a specific license from the United States Government. It is the responsibility of any person or organization contemplating export to obtain such a license before exporting.

WITHIN THAT CONSTRAINT, permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representations about the suitability of this software for any purpose. It is provided as is without express or implied warranty.

Contents

How This Guide Is Organized	viii
Software Development Support	x
Chapter 1. The VERITAS File System Software Developer's Kit	1
VERITAS File System Software Developer's Kit Features	2
Application Interface	2
File Change Log	3
Multi-Volume Support	3
VxFS I/O	3
Software Developer's Kit Packages	3
Required Libraries and Header Files	4
Compiling Environment	5
Chapter 2. File Change Log	7
File Change Log File	8
Record Types	10
Special Records	10
Obsolete Records	11
Typical Record Sequences	12
Superblock	12
Record Details	14
Records With a Trailing File Name	16
Records Without a Trailing File Name	19



Tunables	19
Programmatic Interface	20
Reverse Path Name Lookup	23
Chapter 3. Multi-Volume Support	27
Uses for Multi-Volume Support	29
Volume APIs	30
Examples of Volume Set Operations	30
Querying the Volume Set for a File System	30
Modifying a Volume Within a File System	31
Volume Encapsulation	31
Allocation Policy APIs	32
An Illustration of File Allocation	33
Creating and Assigning Policies	34
Querying the Defined Policies	35
Enforcing a Policy on a File	35
Data Structures	36
Examples	37
Defining and Assigning Allocation Policies	37
Using Volume APIs	39
Chapter 4. Named Data Streams	41
Uses for Named Data Streams	43
Named Data Streams Programmatic Interface	43
Listing Named Data Streams	44
Namespace for Named Data Streams	45
Behavior Changes in Other System Calls	45
Example	46
Programmatic API	47
Programmer's Reference	47

Chapter 5. VxFS I/O	49
Freeze/Thaw	50
Caching Advisories	52
Direct I/O	52
Concurrent I/O	53
Unbuffered I/O	54
Other Advisories	54
Extents	55
Extent Attributes	56
Reservation: Preallocating Space to a File	57
Fixed Extent Size	57
Application Programming Interface for Extent Attributes	58
Allocation Flags	59
Allocation Flags With Fixed Extent Size	61
How to use Extent Attribute APIs	61





Preface

This document contains a general description of the content and usage of the VERITAS File System Software Developer's Kit. Each chapter introduces and discusses a VxFS file system feature, its possible uses, and a description of how to use the application programming interface for common operations. The SDK contains specific programming examples utilizing these interfaces. VxFS is distributed by VERITAS as part of the Storage Foundation Product Suites.

In addition to this document, the VxFS SDK packages contain manual pages, sample programs that demonstrate usage of the features, and coding examples of how to use the API calls. There is also information on VxFS library and include file locations, and compilation examples.

This guide assumes that you have a:

- ◆ Basic understanding of system administration
- ◆ Working knowledge of the UNIX operating system
- ◆ General understanding of file systems
- ◆ Familiarity with programming in the C language and compiling in a UNIX environment



How This Guide Is Organized

[Chapter 1 “The VERITAS File System Software Developer’s Kit,”](#) introduces the major features and characteristics of the VxFS SDK.

[Chapter 2 “File Change Log,”](#) describes the VxFS File Change Log (FCL) API, which tracks changes to files and directories in a file system. The File Change Log can be used by applications such as backup products, web crawlers, search and indexing engines, and replication software that typically scan an entire file system searching for modifications since a previous scan.

[Chapter 3 “Multi-Volume Support,”](#) describes the multi-volume support (MVS) API, which allows several volumes to be represented by a single logical object. All I/O to and from an underlying logical volume is directed by way of *volume sets*. This feature can be used only in conjunction with VERITAS Volume Manager.

[Chapter 4 “Named Data Streams,”](#) describes the named data streams API, which associates multiple data streams with a file.

[Chapter 5 “VxFS I/O,”](#) describes the input/output control (IOCTL) directives, including freezing and thawing a file system, caching advisories, and extents.

Conventions

Convention	Usage	Example
monospace	Used for path names, commands, output, directory and file names, functions, and parameters.	Read tunables from the <code>/etc/vx/tunefstab</code> file. See the <code>ls(1)</code> manual page for more information.
monospace (bold)	Indicates user input.	# ls pubs C:\> dir pubs
<i>italic</i>	Identifies book titles, new terms, emphasized text, and variables replaced with a name or value.	See the <i>User's Guide</i> for details. The variable <i>system_name</i> indicates the system on which to enter the command.
bold	Depicts GUI objects, such as fields, list boxes, menu selections, etc. Also depicts GUI commands.	Enter your password in the Password field. Press Return .
blue text	Indicates hypertext links.	See " Getting Help " on page x.
#	Unix superuser prompt (all shells).	# cp /pubs/4.0/user_book /release_mgnt/4.0/archive
C:\>	Windows user prompt.	C:\> copy \pubs\4.0\user_book c:\release_mgnt\4.0\archive



Getting Help

For technical assistance, visit <http://support.veritas.com> and select phone or email support. This site also provides access to resources such as TechNotes, product alerts, software downloads, hardware compatibility lists, and the VERITAS customer email notification service. Use the Knowledge Base Search feature to access additional product information, including current and past releases of product documentation.

Diagnostic tools are also available to assist in troubleshooting problems associated with the product. These tools are available on disc or can be downloaded from the VERITAS FTP site. See the `README.VRTSspt` file in the `/support` directory for details.

For license information, software updates and sales contacts, visit <https://my.veritas.com/productcenter/ContactVeritas.jsp>. For information on purchasing product documentation, visit <http://webstore.veritas.com>.

Software Development Support

VERITAS offers software development support through membership in the VERITAS Enabled™ Program. For information on VERITAS Enabled and becoming a VERITAS partner, visit the website at <http://www.veritas.com/enabled>.

If you are not a member of the VERITAS Enabled Program, VERITAS sponsors the VERITAS Architect Network, an online forum for discussing VERITAS products. The Application Development Support forum is provided specifically for the development community to exchange ideas and expertise on VERITAS File System application development. VERITAS encourages posting questions related to code development, and actively participates in discussions. To contribute to this forum, go to <http://forums.veritas.com/discussions/forum.jspa?forumID=104>.

The VERITAS File System Software Developer's Kit

1

The VERITAS File System Software Developer's Kit provides developers with the information necessary to use the application programming interfaces (APIs) for features of the VERITAS File System. These APIs are provided with the VxFS Software Developer's Kit.

Most of the APIs covered in this document are available in VxFS 4.0 and subsequent releases. The APIs in [Chapter 5 "VxFS I/O"](#) are available in VxFS 4.0, subsequent releases, and several releases prior.

This chapter provides an overview of VxFS APIs that are provided with the SDK, and are described in detail in later chapters. The following topics are introduced in this chapter:

- ◆ [VERITAS File System Software Developer's Kit Features](#)
- ◆ [Application Interface](#)
- ◆ [File Change Log](#)
- ◆ [Multi-Volume Support](#)
- ◆ [VxFS I/O](#)
- ◆ [Software Developer's Kit Packages](#)
- ◆ [Required Libraries and Header Files](#)
- ◆ [Compiling Environment](#)



VERITAS File System Software Developer's Kit Features

The SDK features include:

- ◆ File Change Log
- ◆ Multi-volume support
- ◆ Named Data Streams
- ◆ VxFS I/O

Application Interface

The API library interfaces highlighted in this SDK are the `vxfsutil` library and VxFS IOCTL directives. The library contains a collection of API interface calls that can be used by applications to take advantage of the features of the VxFS file system. Manual pages are available for all of the API interfaces. The library contains APIs for the following features:

<code>inotopath</code>	Inode-to-path lookup
<code>nattr</code>	Named Data Stream
<code>FCL</code>	File Change Log
<code>MVS</code>	Multi-volume support
Caching Advisories	IOCTL directives
Extents	IOCTL directives
Freeze/Thaw	IOCTL directives

The VxFS API library, `vxfsutil`, can be installed independent of the VERITAS File System product. This library is implemented using a stub library and dynamic library combination. Applications are compiled with the stub library `libvxfsutil.a`, making the application portable to any VxFS target environment. The application can then be run on a VxFS target, and the stub library will find the dynamic library provided with the VxFS target.

The stubs library will use a default path for the location of the `vxfsutil.so` dynamic library. In most cases, the default path should be used. However, the default path can be overridden by setting the environment variable, `LIBVXFSUTIL_DLL_PATH`, to the path of the `vxfsutil.so` library. This structure allows an application to be deployed with minimal issues related to compatibility with other releases of VxFS.

File Change Log

The VxFS File Change Log (FCL) tracks changes to files and directories in a file system. The File Change Log can be used by applications such as backup products, web crawlers, search and indexing engines, and replication software that typically scan an entire file system searching for modifications since a previous scan. See [“File Change Log”](#) on page 7 for more information.

Multi-Volume Support

The multi-volume support (MVS) feature allows a VxFS file system to use multiple VxVM volumes as underlying storage. Administrators and applications can control which files go where, to maximize effective performance while minimizing cost. This feature can be used only in conjunction with VERITAS Volume Manager, and some of the functionality requires additional license keys. See [“Multi-Volume Support”](#) on page 27 for more information.

VxFS I/O

VxFS conforms to the System V Interface Definition (SVID) requirements and supports user access through the Network File System (NFS). Applications that require performance features not available with other file systems can take advantage of VxFS enhancements.

Software Developer's Kit Packages

Two packages comprise the SDK: `VRTSfssdk` and `VRTSfsmnd`. The `VRTSfssdk` package contains libraries, header files, and sample programs in source and binary formats that demonstrate usage of the VxFS API interfaces. The `VRTSfsmnd` package contains this document and the API man pages.

These packages can be obtained separately from the licensed VxFS package, for use to develop and compile applications utilizing the VxFS API interface. To run the applications or sample programs, a licensed VxFS target is required.



The directory structure in the `VRTSfssdk` package is as follows:

<code>src</code>	Contains several subdirectories with sample programs and GNU-based <code>Makefile</code> files on each topic of interest.
<code>bin</code>	Contains symlinks to all the sample programs in the sources directory for easy access to binaries.
<code>include</code>	Contains the header files for API library and <code>ioctl</code> interfaces.
<code>lib</code>	Contains the pre-compiled <code>vxfsutil</code> API interface stubs library.
<code>libsrc</code>	Contains the source code for the <code>vxfsutil</code> API interface stubs library.

Required Libraries and Header Files

The `VRTSfssdk` package is installed in the `/opt` directory. The associated libraries and header files are installed in the following locations:

- ◆ `/opt/VRTSfssdk/4.1/lib/libvxfsutil.a`
- ◆ `/opt/VRTSfssdk/4.1/lib/sparcv9/libvxfsutil.a`
- ◆ `/opt/VRTSfssdk/4.1/include/vxfsutil.h`
- ◆ `/opt/VRTSfssdk/4.1/include/fcl.h`
- ◆ `/opt/VRTSfssdk/4.1/include/sys/fs/vx_ioctl.h`

There are also symlinks to these files from the standard VERITAS paths, which are `/opt/VRTS/lib` and `/opt/VRTS/include`. The standard paths are the default paths in the latest releases of VxFS and the VxFS SDK.

Compiling Environment

Sample programs are installed by the SDK package with compiled binaries. If you would like to recompile with a different compiler, follow these steps.

The required tools for compiling the `src` or `libsrc` directory are as follows:

- ◆ `gmake` or `make` command
- ◆ `gcc` compiler or `cc` command

To compile the `src` and `libsrc` directories:

1. Edit the `make.env` file and modify it with the path to your compiler.
2. Change to the `src` or `libsrc` directory and run the `gmake` or `make` command:

```
# gmake
```

3. After writing the application, it can be compiled as follows:

```
# gcc -I /opt/VRTS/include -L /opt/VRTS/lib -ldl -o MyApp \
MyApp.c libvxfsutil.a
```

The requirements for running the sample programs are as follows:

- ◆ A target system with the appropriate version of `VRTSvxfs` installed
- ◆ Root permission, required for some programs
- ◆ A mounted VxFS 4.x file system
- ◆ Some programs require a file system to be mounted on a volume set





File Change Log

2

The VxFS File Change Log (FCL) tracks changes to files and directories in a file system. Applications that can make use of the FCL are those that are typically required to scan an entire or a subset of a file system to discover changes since the last scan, such as backup utilities, webcrawlers, search engines, and replication programs.

The File Change Log records file system changes such as creates, links, unlinks, renaming, data appended, data overwritten, data truncated, extended attribute modifications, holes punched, and miscellaneous file property updates.

Note The FCL keeps track of the fact that data has changed, not the actual data. It is the responsibility of the application to examine the files that have changed data to determine which data has changed.

Topics in this chapter include:

- ◆ [File Change Log File](#)
- ◆ [Record Types](#)
- ◆ [Superblock](#)
- ◆ [Record Details](#)
- ◆ [Programmatic Interface](#)
- ◆ [Reverse Path Name Lookup](#)



File Change Log File

The FCL stores changes in a sparse file, referred to as the FCL file, in the file system namespace. The FCL file is always located in `/mount_point/lost+found/changelog`. The FCL file behaves like a regular file, but some user-level operations are prohibited, such as writes. The standard system calls `open(2)`, `lseek(2)`, `read(2)` and `close(2)` can access the data in the FCL file. Other system calls are not allowed on the FCL file.

VxFS tracks changes to the file system by appending the FCL file with information pertaining to those changes. The FCL file could be used to track space usage when a file system gets close to being full. The FCL file could be searched for recent file create or overwrite records to note new files or files that have grown recently. Depending on the application's needs, the search can be done on the entire FCL file. Alternately, the search can be performed on a portion of the FCL file that corresponds to a specific time frame. In both cases, the search identifies changes to the file system captured in the entire FCL file.

Another use is to track space usage by looking for files created with particular names. For example, if users are downloading *.mp3 files that are taking up too much space, the FCL file could be read to find files created with the name *.mp3, or the tail end of the FCL file could be read to find file system operations that eventually resulted in a full file system.

For backup applications, searching the FCL file could reduce the need for full file system scans to detect recent changes as part of an incremental backup process. VxFS creates and logs an FCL record for every update operation performed on an FCL-enabled file system.

A file's history can be traced by scanning the FCL file and coalescing FCL record sequences for a file. Related FCL records from a file's creation, attribute changes, write records and the file's deletion can be used to track the file's history.

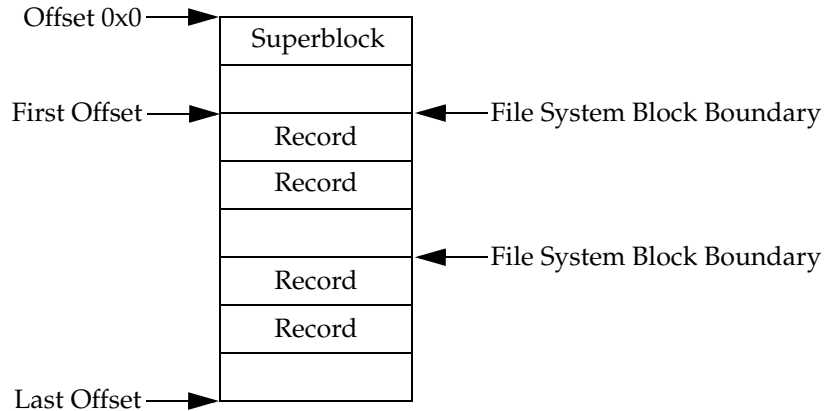
By default, FCL logging is deactivated and can be activated on a per file system basis using the `fcntladm` command. The FCL state is persistent across reboots. When deactivated, the FCL file can still be accessed. When FCL is re-activated, logging continues and records are appended to the file. As file system changes occur, new FCL records are appended to the FCL file and the superblock is updated.

The FCL file contains both the information about the FCL file, which is stored in the FCL superblock, and changes to files and directories in the file system, stored as FCL records. The superblock, which is stored in the first block of the FCL file, describes the state of the FCL file. The superblock can be read at file offset 0 in the FCL file. The remainder of the FCL file is a series of records that describe changes to the file system.

The superblock indicates whether FCL logging is enabled, what time it was activated, and the offsets of the FCL record. When parsing the FCL File, the superblock should be read and the values stored by the application. The superblock should be periodically reread to note any changes in state from previous to current.

When attempting to read FCL records, two important fields from the superblock are the first valid offset (*fc_off*) and last valid offset (*fc_loff*). *fc_off* specifies the offset in the FCL file where the first valid record is logged. *fc_loff* specifies the offset in the FCL file where the last record is logged. The FCL file should be read sequentially. Reading beyond the last valid offset of the FCL file has the same result as reading beyond the end-of-file of a regular file; zero bytes are returned.

The format of the FCL file is described in the figure below.



In most cases, each 32-byte entry in the FCL file is an FCL record. It is self-contained and can be processed by identifying the record type using the record type field and parsing the remaining fields within the record. The `VX_FCL_LINK`, `VX_FCL_RENAME`, and `VX_FCL_UNLINK` records are exceptions to this because of file names. These records are fully described in [“Records With a Trailing File Name”](#) on page 16.

The `VX_FCL_LINK`, `VX_FCL_RENAME`, and `VX_FCL_UNLINK` records span multiple 32-byte entries in the FCL file. For these records, multiple 32-byte entries in the FCL file contain the file name character string. The first of the multi-record FCL entries is processed as stated previously, with the exception of using the name length field to determine if one or more subsequent FCL entries need to be parsed to extract the file name. Once the file name is extracted, processing the subsequent FCL records continues as before by identifying the record type using the record type field.

While FCL records are at 32-byte increments within the FCL file because of the use of FCL entries for file names, there are cases when a 32-byte entry record contains only characters from a file name, in which case the record type field is invalid. For this reason, random access to the FCL file for processing records is not recommended. The type field in each FCL record should be used to know what action or sequence of records may need to be parsed.



Record Types

The following table lists actions that generate FCL record types:

Action to Create an FCL Record	Record Type
Add a link to an existing file or directory	VX_FCL_LINK
Appending write to a file	VX_FCL_DATA_EXTNDWRITE
Create a file or directory	VX_FCL_CREATE
Create a name data stream directory	VX_FCL_CREATE
Create a symbolic link	VX_FCL_SYMLINK
Perform an mmap on a file in a shared and writable mode	VX_FCL_DATA_OVERWRITE
Promote a file from a Storage Checkpoint	VX_FCL_UNDELETE
Punch a hole into a file	VX_FCL_HOLE_PUNCHED
Remove a file or directory	VX_FCL_UNLINK
Remove a named data stream directory	VX_FCL_UNLINK
Rename a file or directory	VX_FCL_RENAME
Rename a file to an existing file	VX_FCL_UNLINK
	VX_FCL_RENAME
Set file attributes (allocation policies, ACLs, and extended attributes)	VX_FCL_EATTR_CHG
Set file extent reservation	VX_FCL_INORES_CHG
Set file extent size	VX_FCL_INOEX_CHG
Set file group ownership	VX_FCL_IGRP_CHG
Set file mode	VX_FCL_IMODE_CHG
Set file size	VX_FCL_DATA_TRUNCATE
Set file user ownership	VX_FCL_IOWN_CHG
Set <i>mtime</i> of a file	VX_FCL_MTIME_CHG
Truncate a file	VX_FCL_DATA_TRUNCATE
Write to an existing block in a file	VX_FCL_DATA_OVERWRITE

Special Records

There are two record types, `VX_FCL_HEADER` and `VX_FCL_NO_CHANGE`, that are used for special purposes within the FCL file. Neither record indicates any specific file system change. They are used for the special purpose of formatting the FCL file.



The FCL file is written in chunks the size of file system blocks. The `VX_FCL_HEADER` record is always the first record written at the beginning of each file system block within the FCL file. Use the `statvfs()` system call to lookup the value of the file system block size stored in the `f_bsize` field of the `statvfs` structure.

In addition, the `VX_FCL_HEADER` record can be used in conjunction with trailing file name records. If the name length field is non-zero, the header record is being used to indicate that a multi-record filename has crossed a file system block boundary with the FCL file. This would indicate the record following the header contains a portion of the filename.

The `VX_FCL_NO_CHANGE` record is used as a filler record.

The following are special records:

<code>VX_FCL_HEADER</code>	Found at the beginning of a file system block and usually has a full or partial name trailing it. Without a trailing name, this record marks the beginning of a block. The file system block size and the current offset are needed to locate the next record.
<code>VX_FCL_NO_CHANGE</code>	Only follows trailing file names. This is a no-op record.

Obsolete Records

These records are older record types and have been deprecated. They are superseded by the use of `VX_FCL_INOEX_CHG`, `VX_FCL_INORES_CHG`, `VX_FCL_IMODE_CHG`, `VX_FCL_IOWN_CHG`, `VX_FCL_IGRP_CHG`, and `VX_FCL_MTIME_CHG` record types in FCL version 2 and later.

The following are obsolete records.:

<code>VX_FCL_INFO_CHG</code>	Recorded changes to file regular attributes, such as file mode, ownership, atime, and mtime.
<code>VX_FCL_ITIMES_CHG</code>	Recorded changes to file atime and mtime.



Typical Record Sequences

The life cycle of a file in a file system is recorded in the FCL file from creation to deletion. When creating a file, the following is a typical sequence of FCL records written to the log:

```
VX_FCL_CREATE
VX_FCL_DATA_EXTNDWRITE
VX_FCL_DATA_OVERWRITE
```

When writing a file, the following is a typical sequence of FCL records written to the log for every write operation:

```
VX_FCL_DATA_EXTNDWRITE
VX_FCL_DATA_OVERWRITE
```

When moving or deleting a file, the following is a typical sequence of FCL records written to the log:

```
VX_FCL_RENAME
VX_FCL_HEADER (optional depending on file name length)
VX_FCL_UNLINK
VX_FCL_HEADER (optional depending on file name length)
```

Superblock

Applications need to read the superblock each time they open the FCL. The activation time (*fc_atime*) should be saved after each read. At the next read, the saved values can be compared with the current values to determine if the FCL is currently enabled, if the FCL was disabled and re-enabled since the last reference.

The FCL superblock is written to the FCL the first time it is activated. It is defined as:

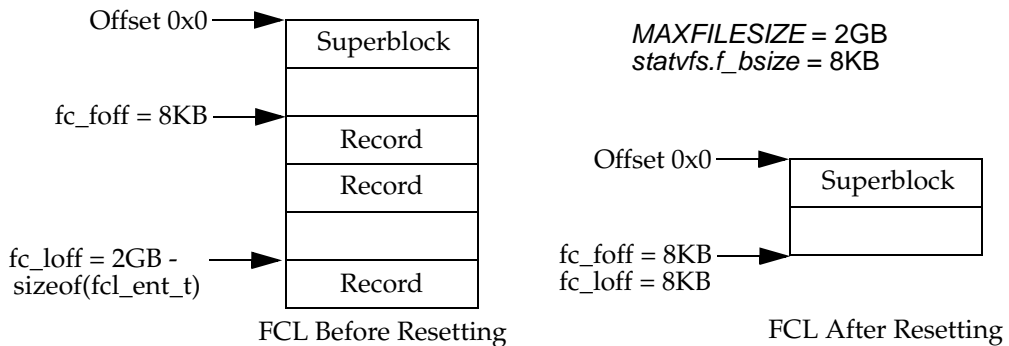
```
struct fcl_sb {
    uint32_t      fc_magic;           /* FCL magic number */
    uint32_t      fc_version;         /* FCL version number */
    fcl_state_t   fc_state;           /* FCL state */
    uint32_t      fc_sync;            /* FCL sync count */
    fcl_timeval32_t fc_atime;          /* FCL activation time */
    uint64_t      fc_foff;             /* first valid offset in
                                       FCL file */
    uint64_t      fc_loff;             /* last valid offset in FCL
                                       file */
    char          fc_oldfile[64];      /* reserved */
    uint32_t      fc_padding[2];       /* padding */
                                           /* 0x70 is length */
};
```

The superblock fields are defined below:

<i>fc_magic</i>	Defined as <code>VX_FCL_MAGIC</code> . It is used to verify that the file is really the FCL file.
<i>fc_version</i>	Incremented when the FCL structure is changed. Version numbers are compared to ensure compatibility.
<i>fc_state</i>	Indicates whether the FCL is activated (ON) or deactivated (OFF).
<i>fc_sync</i>	Incremented each time the FCL is synchronized via the <code>vxfs_fcl_sync()</code> API or <code>fcladm sync</code> command. It is also incremented when a file system Storage Checkpoint is created. See the <i>VERITAS File System Administrator's Guide</i> for information on Storage Checkpoints.
<i>fc_atime</i>	The time stamp of when the FCL was activated. If the activation time of the last read is different from that of the current read, the FCL has been deactivated at least once between the two reads.
<i>fc_foff</i>	The offset of the first valid FCL record. It is initialized to the value of the file system block size, which will be located at the second block in the FCL file. Because an FCL file can be punched with a hole to release unneeded space, the value of <code>fc_foff</code> will change to the offset of the first file system block after the punched region. The punched hole is described later. If the application's last read offset is less than the FCL first valid offset, the application has already missed some records.
<i>fc_loff</i>	The offset in the FCL file indicating the end of the last record. The next FCL record written into the FCL will be written at this offset. When it crosses the system maximum file size, the FCL resets. The file is truncated to 1 block, which contains the superblock. The activation time is reset to the current time. The first and last valid offsets are reset to the file system block size.
<i>fc_padding[2]</i>	Reserved.



The following figure illustrates an FCL resetting after the last valid offset crosses the value of `MAXFILESIZE`.



Record Details

While the FCL is activated, each file or directory change is recorded in the file. An FCL record is represented by the `fcl_ent` structure.

```
struct fcl_ent {
    uint64_t      fe_ino;           /* inode number */
    uint64_t      fe_dino;         /* parent inode number */
    fcl_timeval32_t fe_time;       /* time stamp */
    uint32_t      fe_gen;          /* inode gen count */
    uint16_t      fe_type;         /* change type */
    uint16_t      fe_nmlen;        /* file name length */
                                /* 0x20 is length */
};
```

The descriptions of the fields in the structure are as follows:

- fe_ino* Inode number of the file being changed. To generate the full path name of the changed object, inode number and generation count (*fe_gen*) could be used with the `vxfs_inotopath_gen()` API.
- fe_dino* Inode number for the directory containing the file being changed. The parent directory inode and generation count can be used in the same way to identify the full path name of the parent directory. Adding the trailing file name yields the object's full name.

<i>fe_gen</i>	<p>Generation count of the file changed. The generation count in combination with either inode is passed to <code>vxfs_inotopath_gen()</code> to provide the exact full path name of the object. Without the generation count, the returned path name can be of a re-used inode. In most cases, the generation count refers to the <i>fe_ino</i> field. In the case of a <code>VX_FCL_UNLINK</code> record, this field contains the generation count of the parent inode (<i>fe_dino</i>).</p> <p>In the case of a rename, the <i>fe_gen</i> field of the <code>VX_FCL_HEADER</code> or <code>VX_FCL_NO_CHANGE</code> record, which follows the file name, contains the generation count of the parent directory.</p>
<i>fe_time</i>	A time stamp that provides an approximation of when the change was recorded in the FCL file. Use the <code>ctime()</code> call to interpret this field.
<i>fe_type</i>	Record change type that indicates whether the change was a creation, unlink, write, file attributes change, or other change.
<i>fe_nmlen</i>	Name length field. This field is valid only when the record is either <code>VX_FCL_LINK</code> , <code>VX_FCL_UNLINK</code> , <code>VX_FCL_RENAME</code> , or <code>VX_FCL_HEADER</code> . It specifies the partial or full length of the trailing file name. The current FCL offset plus the name length, rounded up to 32 bytes, yields the offset of the next valid record.

The enum structure below fully defines all of the record types of the *fe_type* field in the `fcl_ent` structure.

```
typedef enum {
    VX_FCL_NO_CHANGE,           /* no change */
    VX_FCL_INFO_CHG,           /* file info change */
    VX_FCL_CREATE,             /* file create */
    VX_FCL_LINK,               /* file link added */
    VX_FCL_UNLINK,             /* file unlink/file deleted */
    VX_FCL_RENAME,             /* file rename */
    VX_FCL_UNDELETE,           /* file undelete */
    VX_FCL_DATA_EXTNDWRITE,     /* file data extending write*/
    VX_FCL_DATA_OVERWRITE,     /* file data overwrite */
    VX_FCL_DATA_TRUNCATE,      /* file data truncate */
    VX_FCL_EATTR_CHG,          /* file extended attribute
                                change*/
    VX_FCL_HOLE_PUNCHED,       /* file hole punched */
    VX_FCL_HEADER,              /* block header record */
    VX_FCL_SYMLINK,             /* symbolic link created */
    VX_FCL_INOEX_CHG,           /* inode extent attributes
                                changed */
    VX_FCL_INORES_CHG,         /* inode reservation
                                changed */
    VX_FCL_IMODE_CHG,           /* inode mode changed */
    VX_FCL_IOWN_CHG,           /* inode owner changed */
}
```



```
VX_FCL_IGRP_CHG,          /* inode group changed */
VX_FCL_ITIMES_CHG,        /* inode times changed */
VX_FCL_MTIME_CHG,        /* inode mtime changed */
VX_FCL_MAX,               /* maximum FCL type + 1 */
} fcl_chgtype_t;
```

New FCL record types may be periodically added; applications should be written accordingly. The introduction of new record types will result in the *fc_version* field in the FCL superblock being incremented.

Records With a Trailing File Name

There are two special records types, *VX_FCL_NO_CHANGE* and *VX_FCL_HEADER*, that are used as filler records in the FCL file. They contain no file system change information, but need to be processed to find the next record. These records are used to pad the FCL file for *VX_FCL_LINK*, *VX_FCL_UNLINK*, and *VX_FCL_RENAME* records because of a variable length record or when spanning a block boundary.

FCL records are written one after another, sometimes interspersed with trailing file names. FCL record types are enumerated in *fcl_chgtype_t*. There are 3 special records, *VX_FCL_LINK*, *VX_FCL_UNLINK*, and *VX_FCL_RENAME*, where file or directory names trail the records. In those records, the file name length field is non-zero. In addition to the name length field, other fields in the record provide additional details about the change:

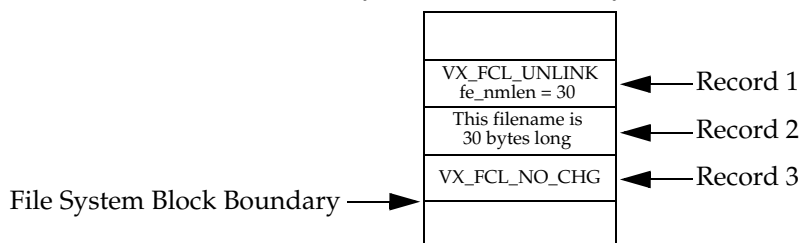
<i>VX_FCL_LINK</i>	The trailing name is the new link's name.
<i>VX_FCL_UNLINK</i>	The trailing name is the name of the unlinked file.
<i>VX_FCL_RENAME</i>	The trailing name is the previous name of the file.

File names have a maximum size of *MAXNAMELEN*. Thus, a trailing file name in the FCL could span one or more FCL records. In all cases, file names are padded with zeroes at the end of the name to a 32-byte boundary to ensure that all FCL records begin on a 32-byte aligned offset.

The following sections provide examples of how file names with trailing records are managed with the FCL_UNLINK record. These same examples apply equally to the VX_FCL_LINK and VX_FCL_RENAME records.

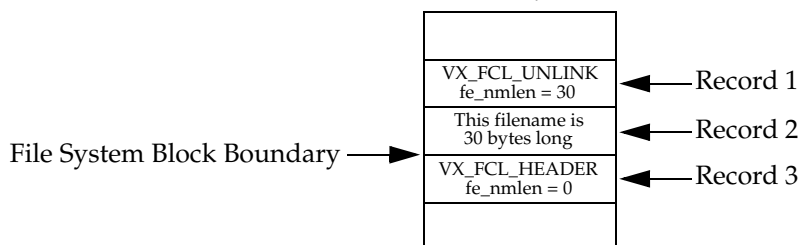
- ◆ VX_FCL_UNLINK record, file name, and VX_FCL_NO_CHANGE record fit completely in the remainder of the current file system block

Record 1 is the VX_FCL_UNLINK record. The record name length (*fe_nmlen*) contains the length of the full name. Record 2 contains the 30-byte file name, which can be read entirely from this record. Record 3 contains the VX_FCL_NO_CHG record, which is used to fill the current file system block boundary.



- ◆ VX_FCL_UNLINK record and file name fit in the remaining of the current file system block

Record 1 is the VX_FCL_UNLINK record. The record name length (*fe_nmlen*) contains the length of the full name. Record 2 contains the 30-byte file name, which can be read entirely from this record. Record 3 contains a VX_FCL_HEADER record, which is written at offset 0 of the next block and its *fe_nmlen* field is set to 0.

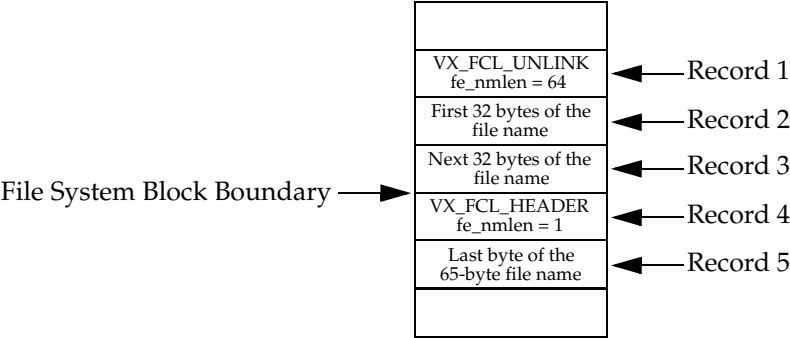


◆ File name crosses the file system block boundary

Record 1 is the VX_FCL_UNLINK record. The record name length, *fe_nmlen*, will be set to the length of the partial file name that fits between the preceding FCL record (record 1) and the file system block boundary. The partial file name is written to records 2 and 3.

Record 4 is a VX_FCL_HEADER record, which is written at offset 0 of the next block, and its *fe_nmlen* field is set to the remaining file name length. Record 5 contains the remaining file name bytes.

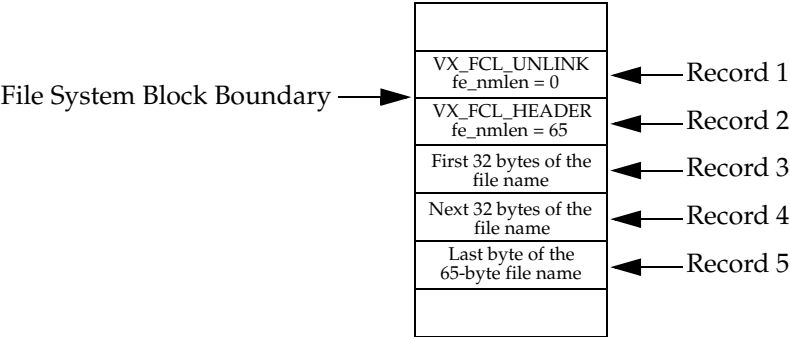
The full file name has to be assembled from reading records 2, 3, and 5.



◆ Insufficient space to write at least 32 bytes of the file name in the current file system block

Record 1 is the VX_FCL_UNLINK record. The record name length, *fe_nmlen*, will be set 0. Record 2 is a VX_FCL_HEADER record, which is written at offset 0 of the next block, and its *fe_nmlen* field is set to the file name length. The partial file name is written to records 3 and 4. Record 5 contains the remaining file name bytes.

The full file name is written after the VX_FCL_HEADER record. The full file name has to be assembled from reading records 3, 4, and 5.



Records Without a Trailing File Name

The remaining FCL records do not have trailing file names and should be read sequentially.

Tunables

There are 3 FCL tunable parameters that can be set via the `vxtunefs` command (see the `vxtunefs(1m)` manual page).

fcl_keeptime Specifies the duration in seconds that FCL records stay in the FCL file before they can be purged. The first records to be purged are the oldest ones, which are located at the beginning of the file. Additionally, records at the beginning of the file can be purged if allocation to the FCL file exceeds *fcl_maxalloc* bytes. The default value is 0. Note that *fcl_keeptime* takes precedence over *fcl_maxalloc*. No hole is punched if the FCL file exceeds *fcl_maxalloc* bytes but the life of the oldest record has not reached *fcl_keeptime* seconds.

Tuning recommendation: The *fcl_keeptime* tunable parameter needs to be tuned only when the administrator wants to ensure that records are kept in the FCL for *fcl_keeptime* length of time. The *fcl_keeptime* parameter generally should be set to a value that is twice the time between FCL scans. For example, if the FCL is scanned every 24 hours, *fcl_keeptime* should be set to 48 hours. This prevents FCL records from being purged before they are read and processed.

fcl_maxalloc Specifies the maximum number of space in bytes to be allocated to the FCL file. When the space allocated exceeds *fcl_maxalloc*, a hole is punched at the beginning of the file. As a result, records are purged and the first valid offset (*fc_off*) is updated. The minimum value of *fcl_maxalloc* is 4MB. The default value is *fs_size*/33.

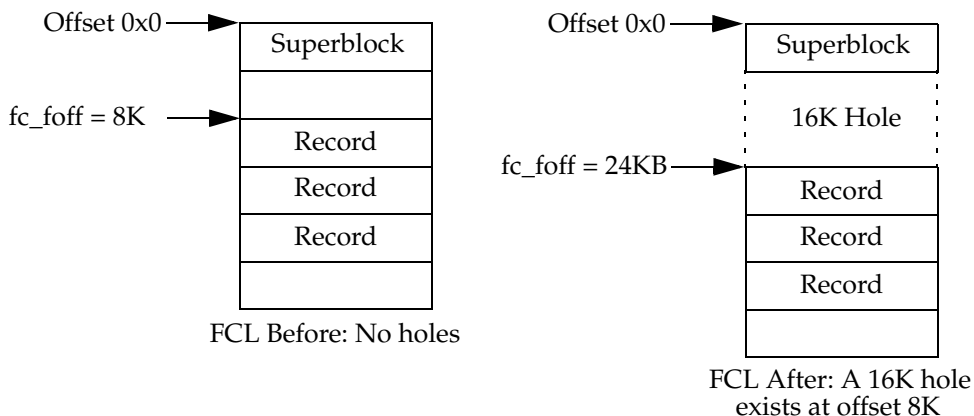


fcl_winterval

Specifies the time in seconds that must elapse before the FCL records an overwrite, extending write, or a truncate. This helps to reduce the number of repetitive records in the FCL. *fcl_winterval* time-out is per inode. If an inode happens to go out of cache and returns, its write interval is reset. As a result, there could be more than one write record for that file in the same write interval. The default value is 3600 seconds.

Tuning recommendation: The *fcl_winterval* tunable parameter should be set to a value that is less than the time between FCL scans. For example, if the FCL is scanned every 24 hours, *fcl_winterval* should be set to less than 24 hours. This ensures that there is at least one record in the FCL for each file being overwritten, extended, or truncated between scans.

In the following figure, a 16K extent is punched at the beginning of the FCL file.



Programmatic Interface

The standard system calls `open(2)`, `lseek(2)`, `read(2)` and `close(2)` can be used on the FCL file at `/mount_point/lost+found/changellog`. Additionally, there is one programmatic interface exposed through `libvxfsutil: vxfs_fcl_sync()`. The following is the syntax for the `vxfs_fcl_sync()` API:

```
int vxfs_fcl_sync(char *fname, uint64_t *offp);
```

The `vxfs_fcl_sync()` API has two parameters: a pointer to the FCL file name, *fname*, and the address of a 64-bit offset, *offp*.

The `vxfs_fcl_sync()` API sets a synchronization point within the FCL file. This synchronization point defines a clear offset into the FCL file that applications can use as a stopping point when reading the FCL. The application saves this offset to use it as a starting point the next time the application reads more of the FCL file.

Setting a synchronization point in the FCL also resets the FCL data write interval for all of the files in the file system. The FCL data write interval is a VxFS tunable used to limit the number of FCL records for file data changes during a specified time interval. This limitation applies to all files in all read-write mounted filesets for a given file system volume or device. See the `vxtunefs(1M)` manual page for more information. By resetting the FCL data write interval, an application is certain to find least one FCL data change record for each file in the file system that is being overwritten, extended, or truncated the next time that the application reads the FCL file from *offp*.

Multiple applications can use the `vxfs_fcl_sync()` function simultaneously. The synchronization point that is set does not affect the operation of other applications using the FCL. Multiple applications issuing a call to `vxfs_fcl_sync()` obtain different offsets to use as their own synchronization points.

Synchronizing the FCL is one of the critical steps in reading and processing the FCL. The prototype is available in the `/opt/VRTSfssdk/4.1/include/vxfsutil.h` header file. See the `vxfs_fcl_sync(3)` manual page for more information.

▼ Sample steps to read and process the FCL

1. Open the `/mountpoint/lost+found/changelog` file.
2. Read the superblock from offset 0.
3. Check the FCL state. If the state is OFF, then you are done. Otherwise, proceed.
4. Record the activation time.
5. Compare the activation time to that of the last time the FCL log was read. If the times are different, then the FCL was deactivated and reactivated since the last time the application read the FCL log. It is likely that some changes to the files in the file system were not recorded in the FCL log. The application cannot depend on the FCL to identify the modified files. Instead, it needs to rescan the file system and re-establish a baseline before using the FCL log again.
6. If this is not the first time the application read the FCL log, check if the last offset read by the application is greater than the FCL's first valid offset. If the last offset read is not greater than the FCL's first valid offset, it is possible that holes were punched in front of the FCL and records in the punched region were not read. Rescan the file system and re-establish a baseline before using the FCL again.



7. Synchronize the FCL via the `vxfs_fcl_sync()` API, which returns the value of `sync_offset`. Read the FCL from the last offset read by the application up to the offset specified by `sync_offset`.
8. Save this offset as the offset last read by the application.
9. Reread the superblock to confirm that the offsets from which the application has read FCL records are still valid.

The following sample code fragment reads the FCL superblock, checks that the state of the FCL is `VX_FCLS_ON`, issues a call to `vxfs_fcl_sync()` to obtain a finishing offset to read to, determines the first valid offset in the FCL file, then reads the entries in 8K chunks from this offset. The section `process fcl entries` is what an application developer must supply to process the entries in the FCL.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <fcl.h>
#include <vxfsutil.h>

#define FCL_READSZ 8192

char* fclname = "/mnt/lost+found/changelog";

int
read_fcl(fclname)
    char* fclname;
{
    struct fcl_sb fclsb;
    uint64_t off, lastoff;
    size_t size;
    char buf[FCL_READSZ], *bufp = buf;
    int fd;
    int err = 0;

    if ((fd = open(fclname, O_RDONLY)) < 0) {
        return ENOENT;
    }
    if ((off = lseek(fd, 0, SEEK_SET)) != 0) {
        close(fd);
        return EIO;
    }
    size = read(fd, &fclsb, sizeof (struct fcl_sb));
```



```

    if (size < 0) {
        close(fd);
        return EIO;
    }
    if (fclsb.fc_state == VX_FCLS_OFF) {
        close(fd);
        return 0;
    }
    if (err = vxfs_fcl_sync(fclname, &lastoff)) {
        close(fd);
        return err;
    }
    if ((off = lseek(fd, fclsb.fc_foff, SEEK_SET)) !=
        fclsb.fc_foff) {
        close(fd);
        return EIO;
    }
    while (off < lastoff) {
        if ((size = read(fd, bufp, FCL_READSZ)) <= 0) {
            close(fd);
            return errno;
        }
        /* process fcl entries */
        off += size;
    }
    close(fd);
    return 0;
}

```

Reverse Path Name Lookup

The reverse path name lookup feature obtains the full path name of a file or directory from the inode number of that file or directory. The inode number is provided as an argument to the `vxfs_inotopath_gen()` application programming interface library function. See the `vxfs_inotopath_gen(3)` online manual page for more information.

The reverse path name lookup feature can be useful for a variety of applications, such as for clients of the VxFS file change log feature, in backup and restore utilities, and for replication products. Typically, these applications store information by inode numbers because a path name for a file or directory can be very long, thus the need for an easy method of obtaining a path name.



An inode is a unique identification number for each file in a file system. An inode contains the data and metadata associated with that file, but does not include the file name to which the inode corresponds. It is therefore relatively difficult to determine the name of a file from an inode number. The `ncheck` command provides a mechanism for obtaining a file name from an inode identifier by scanning each directory in the file system, but this process can take a long period of time. The VxFS reverse path name lookup feature obtains path names relatively quickly.

Note Because symbolic links do not constitute a path to the file, the reverse path name lookup feature cannot track symbolic links to files.

A file inode number, generation count, and, in the case of a `VX_FCL_LINK`, `VX_FCL_UNLINK`, or `VX_FCL_RENAME` record, trailing file name, when combined with the use of reverse path name lookup, can generate full path names for each FCL record.

The `vxfs_inotopath_gen()` API takes a mount point name, inode number, and inode generation count and returns a buffer that contains one or more (in the case of multiple links to an inode) full path names representing the inode. The inode generation count parameter ensures that the returned path name is not a false value of a re-used inode. Because of this, use the `vxfs_inotopath_gen()` API whenever possible.

The `vxfs_inotopath()` API is included only for backward compatibility. The `vxfs_inotopath()` API does not take the inode generation count.

The following is the syntax for the `vxfs_inotopath()` and `vxfs_inotopath_gen()` APIs:

```
int vxfs_inotopath(char *mount_point, uint64_t inode_number,
                  int all, char ***bufp, int *inentries)

int vxfs_inotopath_gen(char *mnt_pt, uint64_t inode_number,
                      uint32_t inode_generation, int all,
                      char ***bufp, int *nentries)
```

For the `vxfs_inotopath()` call, the *all* argument must be 0 to obtain a single path name or 1 to obtain all path names. The *mount_point* argument specifies the file system mount point. Upon successful return, *bufp* points to a two-dimensional character pointer containing the path names and *nentries* contains the number of entries. Each entry of the returned two-dimensional array is `MAXPATHLEN` in size and must be freed by the calling application.

The `vxfs_inotopath_gen()` call is identical to the `vxfs_inotopath()` call, except that it uses an additional parameter, *inode_generation*. The `vxfs_inotopath_gen()` function returns one or more path names associated with the given inode number, if the *inode_generation* passed matches the current generation of the inode number. If the generations differ, it returns with an error. Specify *inode_generation*=0 when the generation count is unknown. This is equivalent to using the `vxfs_inotopath()` call.

The `vxfs_inotopath_gen()` and `vxfs_inotopath()` calls are supported only on Version 6 and later disk layouts.





Multi-Volume Support

3

The multi-volume support (MVS) feature allows a VxFS file system to use multiple VxVM volumes as underlying storage instead of the traditional single volume per file system. These different volumes can have different characteristics, such as performance, redundancy, or cost, or they could be used to isolate different parts of the file system from each other for performance or administrative purposes.

Administrators and applications can control which files and metadata go into which volumes by using allocation policies. Each file system operation that allocates space examines the applicable allocation policies to see which volumes are specified for that operation. Allocation policies normally only affect new allocations, but there are also interfaces to move existing data to match a new allocation policy.

There are several levels of policies that can apply to each allocation:

- ◆ Per-file policies
- ◆ Per-Storage-Checkpoint policies
- ◆ Per-file-system policies

The most specific allocation policy in effect for a given allocation operation will be used.

The MVS APIs fall into three basic categories:

- ◆ Manipulation of volumes within a file system
- ◆ Manipulation of allocation policy definitions
- ◆ Application of allocation policies

Each of the APIs is also available via options to the `fsvoladm(1M)` and `fsapadm(1M)` commands.



Topics in this chapter include:

- ◆ [Uses for Multi-Volume Support](#)
- ◆ [Volume APIs](#)
- ◆ [Allocation Policy APIs](#)
- ◆ [Data Structures](#)
- ◆ [Examples](#)

Uses for Multi-Volume Support

Possible uses for the multi-volume support feature include the following:

- ◆ Controlling where files are stored so that specific files or file hierarchies can be assigned to different volumes.
- ◆ Separating Storage Checkpoints so that data allocated to a Storage Checkpoint is isolated from the rest of the file system.
- ◆ Separating file system metadata from file data.
- ◆ Encapsulating volumes so that a volume appears in the file system as a file. This is particularly useful for databases that are running on raw volumes.
- ◆ Migrating files off a volume so that the volume can be replaced or serviced.
- ◆ Implementing a storage optimization application that periodically scans the file system and modifies the allocation policies in response to changing patterns of storage use.



Volume APIs

The volume APIs can be used to add volumes to a file system, remove volumes from a file system, list which volumes are in a file system, and retrieve information on usage and availability of space in a volume.

Multi-volume file systems can only be used with VxVM volume sets. Volume sets are administered via the `vxvset` command. See the *VERITAS Volume Manager Administrator's Guide* for more information.

Examples of Volume Set Operations

- ◆ Convert `myvol1` to a volume set:

```
# vxvset make myvset myvol1
```
- ◆ Add `myvol2` to the volume set `myvset`:

```
# vxvset addvol myvset myvol2
```
- ◆ List the volumes of `myvset`:

```
# vxvset list myvset
```
- ◆ Remove `myvol2` from `myvset`:

```
# vxvset rmvol myvset myvol2
```

Querying the Volume Set for a File System

- ◆ Query all the volumes associated with the file system:

```
vxfs_vol_enumerate(fd, &count, infop);
```
- ◆ Query a single volume:

```
vxfs_vol_stat(fd, vol_name, vol_size);
```


Modifying a Volume Within a File System

- ◆ Grow or shrink a volume:

```
vxfs_vol_resize(fd, vol_name, new_vol_size);
```

- ◆ Remove a volume from a file system:

```
vxfs_vol_remove(fd, vol_name);
```

- ◆ Add a volume to a file system:

```
vxfs_vol_add(fd, new_vol_name, new_vol_size);
```

Volume Encapsulation

- ◆ Encapsulate an existing raw volume and make the volume contents appear as a file in the file system:

```
vxfs_vol_encapsulate(encapsulate_name, vol_name, vol_size);
```

- ◆ De-encapsulate an existing raw volume to remove the file from the file system:

```
vxfs_vol_deencapsulate(encapsulate_name);
```

See the *VERITAS File System Administrator's Guide* for more information.



Allocation Policy APIs

To make full use of multi-volume support features, VxFS provides support for allocation policies that allow files or groups of files to be assigned to specified volumes within the volume set.

An allocation policy specifies a list of volumes and the order in which to attempt allocations. A policy can be assigned to a file, file system, or Storage Checkpoint created from a file system. When policies are assigned to objects in the file system, you must specify how the policy maps to both metadata and file data. For example, if a policy is assigned to a single file, the file system must know where to place both the file data and metadata. If no policies are specified, the file system places data randomly.

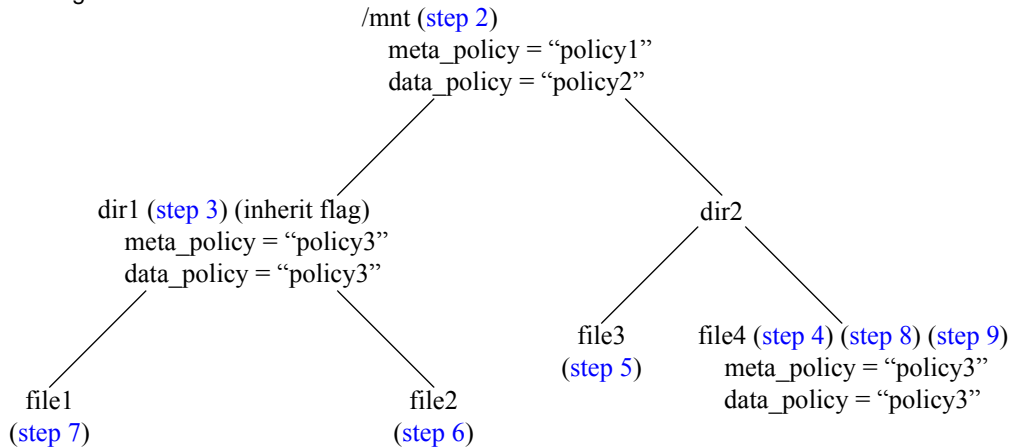
The allocation policies are defined per file system and are persistent. There is no fixed limit on the number of allocation policy definitions in a file system. Once a policy is assigned, new file allocations are governed by the policy. For files allocated before a policy was defined or assigned or when a policy on a file has been changed, the policy can be enforced, causing the file to be re-allocated to the appropriate volumes. Allocation policies can be inherited by a newly created file from its parent directory. This is accomplished by specifying the `FSAP_INHERIT` flag when assigning the policy to the parent directory.

Currently, there is no interface for determining where an existing file is currently allocated. However, these APIs can be used to assign and enforce a policy on a file to assure that the blocks are allocated properly.

An Illustration of File Allocation

The following figure shows how you might use the allocation policies to direct file allocations. In the figure, the `/mnt` file system has 3 volumes in its volume set: `vol-01`, `vol-02`, and `vol-03`. These volumes correspond to `policy1`, `policy2`, and `policy3`, respectively.

Directing File Allocations



▼ Directing File Allocations

The following are actions taken in the steps shown in the figure:

1. Create the allocation policies on the `/mnt` file system.
2. Assign the data and metadata allocation policies to the `/mnt` file system as `policy1` and `policy2`.
3. Assign the data and metadata allocation policies to `dir1` with the `INHERIT` flag, with both as `policy3`.
4. Create `file4` (100MB), which becomes allocated to `vol-02`.
5. Create `file3` (10MB), which becomes allocated to `vol-02`.
6. Create `file2` (100MB), which becomes allocated to `vol-03`.



7. Create `file1` (100MB), which becomes allocated to `vol-03`.
8. Assign the data and metadata allocation policies to `file4`, with both as `policy3`.
9. Enforce the allocation policies on `file4`, which reallocates the file to `vol-03`.

In the figure, the file system has a policy assignment that allocates data as directed by `policy1` and metadata as directed by `policy2`. These policies cause files to be allocated on `vol-01` and `vol-02`, except for `dir1`, which has overriding assignments for allocation on `vol-03`.

When the `file3` and `file4` files are created, they will be allocated on `vol-02` as directed by the `policy1` and `policy2` assignments. When `file1` and `file2` are created, they will be allocated on `vol-03`, as specified by `policy3`.

When `file4` is created, the initial allocation is on `vol-01` and `vol-02`. To move `file4` to `vol-03`, assign `policy3` to `file4` and enforce that policy on the file. This reallocates `file4` to `vol-03`.

Creating and Assigning Policies

The following is an example for creating and assigning a policy using the multi-volume API:

1. Define a policy for a file system.

```
vxfs_ap_define(fd, fsap_info_ptr, 0);
```

2. Assign a policy to a file system.

```
vxfs_ap_assign_fs(fd, data_policy, meta_policy);
```

3. Assign a policy to a file or directory.

```
vxfs_ap_assign_file(fd, data_policy, meta_policy, 0);
```

4. Assign a policy to a Storage Checkpoint

```
vxfs_ap_assign_ckpt( fd, check_point_name, data_policy,  
                    meta_policy)
```

Querying the Defined Policies

- ◆ Query all policies on a file system.
`vxfs_ap_enumerate(fd, &count, fsap_info_ptr);`
- ◆ Query a single defined policy.
`vxfs_ap_query(fs, fsap_info_ptr);`
- ◆ Query a file for its assigned policies.
`vxfs_ap_query_file(fs, data_policy, meta_policy, 0);`
- ◆ Query a Storage Checkpoint for its assigned policies.
`vxfs_ap_query_ckpt(fd, check_point_name, data_policy,
meta_policy)`

Enforcing a Policy on a File

- ◆ Enforce a policy. This may cause the file to be reallocated to another volume.
`vxfs_ap_enforce_file(fd, data_policy, meta_policy);`



Data Structures

For more information, see the `vxfsutil.h` header file and `libvxfsutil.a` library file.

```
#define FSAP_NAMESZ      64
#define FSAP_MAXDEVS    256
#define FSDEV_NAMESZ    32

struct fsap_info {          /* policy structure */
    char ap_name[FSAP_NAMESZ]; /* policy name */
    uint32_t ap_flags;      /* FSAP_CREATE | FSAP_INHERIT |
                             FSAP_ANYUSER */
    uint32_t ap_order;      /* FSAP_ORDER_AS GIVEN |
                             FSAP_ORDER_LEASTFULL |
                             FSAP_ORDER_ROUNDROBIN */
    uint32_t ap_ndevs;      /* number of volumes */
    char ap_devs[FSAP_MAXDEVS][FSDEV_NAMESZ];
                             /* volume names associated with this
                             policy */
};

struct fsdev_info {        /* volume structure */
    int dev_id;            /* a number from 0 to n */
    uint64_t dev_size;     /* size in bytes of volume */
    uint64_t dev_free;
    uint64_t dev_avail;
    char dev_name[FSDEV_NAMESZ]; /* volume name */
};
```

Examples

The following examples assume there is a volume set, `volset`, with the volumes `vol-01`, `vol-02`, and `vol-03`. The file system mount point `/mnt` is mounted on `volset`.

Defining and Assigning Allocation Policies

The following pseudocode provides an example of using the allocation policy APIs to define and assign allocation policies.

- ◆ Reallocate an existing file's data blocks to a specific volume (`vol-03`)

```
/* Create a data policy for moving file's data */

strcpy((char *) ap.ap_name, "Data_Mover_Policy");
ap.ap_flags = FSAP_CREATE;
ap.ap_order = FSAP_ORDER_AS GIVEN;
ap.ap_nd devs = 1;
strcpy(ap.ap_devs[0], "vol-03");

fd = open("/mnt", O_RDONLY);
vxfs_ap_define(fd, &ap, 0);

file_fd = open ("/mnt/file_to_move", O_RDONLY);
vxfs_ap_assign_file(file_fd, "Data_Mover_Policy", NULL, 0);

vxfs_ap_enforce_file(file_fd, "Data_Mover_Policy", NULL);
```



- ◆ Create policies to allocate new files under directory `dir1` according to the policies

In this example, the files are under `dir1`, the metadata will be allocated to `vol-01`, and file data will be allocated to `vol-02`.

```
/* Define 2 policies */

/* Create the RAID5 policy */

strcpy((char *) ap.ap_name, "RAID5_Policy");
ap.ap_flags = FSAP_CREATE | FSAP_INHERIT;
ap.ap_order = FSAP_ORDER_ASGIVEN;
ap.ap_ndevs = 1;
strcpy(ap.ap_devs[0], "vol-02");

fd = open("/mnt", O_RDONLY);
dir_fd = open("/mnt/dir1", O_RDONLY);

vxfs_ap_define(fd, &ap, 0);

/* Create the mirror policy */

strcpy((char *) ap.ap_name, "Mirror_Policy");
ap.ap_flags = FSAP_CREATE | FSAP_INHERIT;
ap.ap_order = FSAP_ORDER_ASGIVEN;
ap.ap_ndevs = 1;
strcpy(ap.ap_devs[0], "vol-01");

vxfs_ap_define(fd, &ap, 0);

/* Assign policies to the directory */

vxfs_ap_assign_file(dir_fd, "RAID5_Policy", "Mirror_Policy", 0);

/* Create file under directory dir1 */
/* Meta and data blocks for file1 will be allocated on
   vol-01 and vol-02 respectively. */

file_fd = open("/mnt/dir1/file1");
write(file_fd, buf, 1024);
```


Using Volume APIs

The following pseudocode provides an example of using the volume APIs.

- ◆ Shrink or grow a volume within a file system

To grow a volume, use the `vxresize` command to grow the physical volume. Then, use the `vxfs_vol_resize()` call to grow the file system.

```
/* stat volume "vol-03" to get the size information */

fd = open("/mnt");
vxfs_vol_stat(fd, "vol-03", infop);

/* resize (shrink/grow) accordingly. This example shrinks
   the volume by half */

vxfs_vol_resize(fd, "vol-03", infop->dev_size / 2);
```

- ◆ Encapsulate a raw volume `vol-03` as a file (`encapsulate_name`) in a the file system `/mnt`. The volume must first be added to the volume set before encapsulation.

```
/* Take the raw volume vol-03 and encapsulate it. The volume's
   contents will be accessible through the given path name. */

vxfs_vol_encapsulate("/mnt/encapsulate_name", "vol-03",
                    infop->dev_size);

/* Access to the volume is through writes and reads of file
   "/mnt/encapsulate_name" */

encap_fd = open("/mnt/encapsulate_name");
write(encap_fd, buf, 1024);
```

- ◆ De-encapsulate the raw volume `vol-03` known as `encapsulate_name` in the file system `/mnt`

```
/* Use de-encapsulate to remove raw volume. After de-encapsulation
   vol-03 is still part of volset, but is not an active part of
   the file system. */

vxfs_vol_deencapsulate("/mnt/encapsulate_name");
```

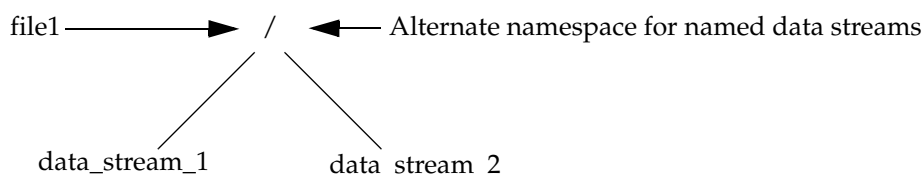




Named Data Streams

Named data streams associate multiple data streams with a file. The default (unnamed) data stream can be accessed through the file descriptor returned by the `open()` function called on the file name. The other data streams are stored in an alternate name space associated with the file. The following figure illustrates the alternate namespace associated with a file.

Alternate Namespace



In the figure, the `file1` file has two named data streams: `data_stream_1` and `data_stream_2`.

Every file can have its own alternate namespace to store named data streams. The alternate namespace can be accessed through the named data stream APIs supported by VxFS.

Access to the named data stream can be done through a file descriptor using the named data stream library functions. Applications can open the named data stream to obtain a file descriptor and perform `read()`, `write()`, and `mmap()` operations using the file descriptor. These system calls work as though they are operating on a regular file. The named data streams of a file are stored in a hidden named data stream directory inode associated with the file. The hidden directory inode for the file can be accessed only through the named data stream application programming interface.

There are no VxFS-supplied administrative commands to use this feature. A VxFS API is provided for creating, reading, and writing the named data streams of a file.

This feature is compatible with the Solaris 10 administrative commands.

Note Named data streams are also known as *named attributes*.



Topics in this chapter include:

- ◆ [Uses for Named Data Streams](#)
- ◆ [Named Data Streams Programmatic Interface](#)
- ◆ [Listing Named Data Streams](#)
- ◆ [Namespace for Named Data Streams](#)
- ◆ [Behavior Changes in Other System Calls](#)
- ◆ [Example](#)
- ◆ [Programmatic API](#)
- ◆ [Programmer's Reference](#)

Uses for Named Data Streams

Named data streams allow applications to attach information to a file that appears to be hidden. An administrative program could use this to attach file usage information, backup information, and so on. An application could use this feature to hide or collect file attachments. For example, a multi-media document could have all text, audio clips, and video clips organized in one file rather than in several files. A document being reviewed by multiple people could have each person's comments attached to the file as a named data stream.

Named Data Streams Programmatic Interface

The following standard system calls can manipulate named data streams:

<code>open()</code>	Opens a named data stream.
<code>read()</code>	Reads a named data stream.
<code>write()</code>	Writes a named data stream.
<code>getdents()</code>	Reads directory entries and puts in a file system independent format.
<code>mmap()</code>	Maps pages of memory.
<code>readdir()</code>	Reads a directory.

VxFS named data stream functionality is available through several application programming interface functions.

The `vxfs_nattr_open()` function works similar to the `open()` system call, except that the path is interpreted as a named data stream to a file descriptor. If the `vxfs_nattr_open()` operation completes successfully, the return value is the file descriptor associated with the named data stream. The file descriptor can be used by other input/output functions to refer to that named data stream. If the path of the named data stream is set to "." the file descriptor returned points to the named data stream directory vnode. The syntax for the `vxfs_nattr_open()` API is as follows:

```
int vxfs_nattr_open(int fd, char *path, int oflag);
```



The `vxfs_nattr_link()` function creates a new directory entry for the existing named data stream and increments its link count by one. There is a pointer to an existing named data stream in the named data stream namespace and a pointer to the new directory entry created in the named data stream namespace. The syntax for the `vxfs_nattr_open()` API is as follows:

```
int vxfs_nattr_link(int sfd, char *spath, char *tpath);
```

The `vxfs_nattr_unlink()` function removes the named data stream at a specified path. The calling function must have write permission to remove the directory entry for the named data stream. The syntax for the `vxfs_nattr_unlink()` API is as follows:

```
int vxfs_nattr_unlink(int fd, char *path);
```

The `vxfs_nattr_rename()` function changes a specified namespace entry at `path1` to a second specified namespace at `path2`. The specified paths are resolved relative to a pointer to the named data stream directory vnodes. The syntax for the `vxfs_nattr_rename()` API is as follows:

```
int vxfs_nattr_rename(int sfd, char *old, char *tnew);
```

The `vxfs_nattr_utimes()` function sets the access and modification times of the named data stream. The syntax for the `vxfs_nattr_utimes()` API is as follows:

```
int vxfs_nattr_utimes(int sfd, const char *path,  
                      const struct timeval times[2]);
```

See the `vxfs_nattr_open(3)`, `vxfs_nattr_link(3)`, `vxfs_nattr_unlink(3)`, `vxfs_nattr_rename(3)`, and `vxfs_nattr_rename(3)` manual pages for more information.

Listing Named Data Streams

The named data streams for a file can be listed by calling `getdents()` on the named data stream directory inode. For example:

```
fd = open("foo", O_RDWR);          /* open file foo */  
afd = vxfs_nattr_open(fd, "stream1",  
                      O_RDWR|O_CREAT, 0777); /* create named data stream  
                                              stream1 for file foo */  
write(afd, buf, 1024);             /* writes to named stream file */  
read(afd, buf, 1024);             /* reads from named stream file */  
dfd = vxfs_nattr_open(fd, ".", O_RDONLY); /* opens named stream  
                                              directory for file foo */  
getdents(dfd, buf, 1024);          /* reads directory entries for  
                                    named stream directory */
```

The reverse name lookup call resolves a stream file to a pathname. The resulting pathname's format is similar to the following:

```
/mount_point/file_with_named_data_stream/./data_stream_file_name
```

Namespace for Named Data Streams

Names starting with "\$vxfs:" are reserved for future use. Creating a data stream in which the name starts with "\$vxfs:" fails with an EINVAL error.

Behavior Changes in Other System Calls

Though the named data stream directory is hidden from the namespace, it is possible to open the name data stream directory inode with a `fchdir()` or `fchroot()` call. Some of the attributes (such as ".") are not defined for a named data streams directory. Any operation that accesses these fields can fail. Attempts to create directories, symbolic links, or device files on a named data stream directory will fail. `VOP_SETATTR()` called on a named data stream directory or named data stream inode will also fail.

An alternative method for reading the hidden directory using the `fchdir()` call is as follows:

```
fd = open(filename, O_RDONLY)
dfd = vxfs_nattr_open(fd, ".", O_RDONLY, mode)

fchdir(dfd);
dirp = opendir(".");
readdir_r(dirp, (struct dirent *)&entry, &result);
```

Note The usage section of the `getcwd(3C)` man page states that applications should exercise care when using the `chdir(2)` call in conjunction with `getcwd()`. The current working directory is global to all threads within a process. If more than one thread calls `chdir()` to change the working directory, a subsequent call to `getcwd()` could produce results that are unexpected.



Example

Using the API calls, a file, `named_stream_file`, was created with 20 named data streams. Using the `ls` command on the file displays the following:

```
# ls -al named_stream_file
-r-xr-lr-x    1 root  other   1024 Aug 12 09:49 named_stream_file
```

The named data streams are not displayed by the `ls` command. When named data streams are created, they are organized in a hidden directory. Using the `getdents()` or `readdir_r()` system call, you can query the `named_stream_file` file for its directory contents, which contains the 20 named stream files.

Attribute Directory contents for `/vxfstest1/named_stream_file`

```
0x1fff root  other 1K Thu Aug 12 09:49:17 2004 .
0x565 root  other 1K Thu Aug 12 09:49:17 2004 ..
0x177 root  other 1K Thu Aug 12 09:49:17 2004 stream0
0x177 root  other 1K Thu Aug 12 09:49:17 2004 stream1
0x177 root  other 1K Thu Aug 12 09:49:17 2004 stream2
.
.
.
0x177 root  other 1K Thu Aug 12 09:49:17 2004 stream17
0x177 root  other 1K Thu Aug 12 09:49:17 2004 stream18
0x177 root  other 1K Thu Aug 12 09:49:17 2004 stream19
```


Programmatic API

The named data streams API uses a combination of standard system calls and VxFS API calls to utilize its functionality. The following is pseudo code for creating the above example:

```
/* Create and open a file */
if ((fd = open("named_stream_file", O_RDWR | O_CREAT | O_TRUNC,
mode)) < 0) {
    sprintf(error_buf, "%s, Error Opening File %s ", argv[0],
filename);
    perror(error_buf);
    exit(-1);
}

/* Write to the regular file as usual */
write(fd, buf, 1024);

/* Create several named data streams for file named_stream_file */
for (i = 0; i < 20; i++) {
    sprintf(attrname, "%s%d", "stream", i);
    nfd = vxfs_nattr_open(fd, attrname, O_WRONLY | O_CREAT, mode);
    if (nfd < 0) {
        sprintf(error_buf,
            "%s, Error Opening Attribute file %s/./%s ",
            argv[0], filename, attrname);
        perror(error_buf);
        exit(-1);
    }
    /* Write some data to the stream file */
    memset(buf, 0x41 + i, 1024);
    write(nfd, buf, 1024);
    close(nfd);
}
```

Programmer's Reference

When using the `cp`, `tar`, `ls` or similar commands to copy or list a file with named data streams, the file will be copied or listed, but the attached named data streams will not be copied or listed.

Note The Solaris 9 operating environment and later provide the `-@` option that may be specified with these commands to manipulate the named data streams.





Unlike the other VxFS APIs described in this document, the APIs described in this chapter are available in previous releases of VxFS on all platforms. The exception is the API that provides concurrent I/O access through the VxFS caching advisories, which is available on VxFS 4.1 and later releases.

Topics in this chapter include:

- ◆ [Freeze/Thaw](#)
- ◆ [Caching Advisories](#)
- ◆ [Extents](#)



Freeze/Thaw

Freezing a file system is a necessary step for obtaining a stable and consistent image of the file system. Consistent file system images can be obtained and used with a file system snapshot tool. The freeze operation flushes all buffers and pages in the file system cache that contain dirty metadata and user data. The operation then suspends any new activity on the file system until the file system is thawed.

VxFS provides ioctl interfaces to application programs to freeze and thaw VxFS file systems. The interfaces are `VX_FREEZE`, `VX_FREEZE_ALL`, and `VX_THAW`.

The `VX_FREEZE` ioctl operates on a single file system. The program performing this ioctl can freeze the specified file system and block any attempts to access the file system until it is thawed. The file system will thaw once the time-out value, specified with the `VX_FREEZE` ioctl, has expired, or the `VX_THAW` ioctl is operated on the file system.

The `VX_THAW` ioctl operates on a frozen file system. It can be used to thaw the specified file system before the freeze time-out period has elapsed.

The `VX_FREEZE_ALL` ioctl interface freezes one or more file systems. The `VX_FREEZE_ALL` ioctl operates in an atomic fashion when there are more than one file systems specified with a freeze operation. VxFS blocks access to the specified file systems simultaneously and disallows a user-initiated write operation that may modify more than one file system with a single write operation. Because `VX_FREEZE_ALL` can be used with a single file system, `VX_FREEZE_ALL` is the preferred interface over the `VX_FREEZE` ioctl.

The execution of the `VX_FREEZE` or `VX_FREEZE_ALL` ioctls will result in a clean file system image that can be mounted after the image is split off from the file system device. In response to a freeze request, all modified file system metadata is flushed to disk with no pending file system transactions in the log that must be replayed before mounting the split off image.

Both the `VX_FREEZE` and `VX_FREEZE_ALL` interfaces can be used to freeze locally mounted file systems, or locally or remotely mounted cluster file systems. See the following table for compatibility with VxFS releases:

Freeze/Thaw Compatibility With VxFS Releases

	VxFS 3.5	VxFS 4.0	VxFS 4.1
<code>VX_FREEZE</code>	Local File System	Local File System Cluster File System	Local File System Cluster File System
<code>VX_FREEZE_ALL</code>	Local File System	Local File System	Local File System Cluster File System



When freezing a file system, care should be taken with choosing a reasonable time-out value for freeze to reduce impact to external resources targeting the file system. User or system processes and resources are blocked while the file system is frozen. If the specified time-out value is too large, resources will be blocked for an extended period of time.

During a file system freeze, any attempt to get a file descriptor from the root directory of the file system for use with the `VX_THAW` ioctl will cause the calling process to be blocked as the result the frozen state of the file system. The file descriptor must be acquired before issuing the `VX_FREEZE_ALL` or `VX_FREEZE` ioctl.

File systems frozen with the `VX_FREEZE_ALL` ioctl before the time-out has expired can be thawed by issuing the `VX_THAW` ioctl for each file system.

The programming interface is as follows:

```
include <sys/fs/vx_ioctl.h>

int    timeout;
int    vxfs_fd;

/*
 * A common mistake is to pass the address of "timeout".
 * Do not pass the address of timeout, as that would be interpreted
 * as a very long timeout period
 */
if (ioctl(vxfs_fd, VX_FREEZE, timeout)) {
    perror("ERROR: File system freeze failed");
}
```

For multiple file systems:

```
int    vxfs_fd[NUM_FILE_SYSTEMS];
struct vx_freezeall freeze_info;

freeze_info.num = NUM_FILE_SYSTEMS
freeze_info.timeout = timeout;
freeze_info.fds = &vxfs_fd[0];

if (ioctl(vxfs_fd[0], VX_FREEZE_ALL, &freeze_info)) {
    perror("ERROR: File system freeze failed");
}

for (i = 0; i < NUM_FILE_SYSTEMS; i++)
    if (ioctl(vxfs_fd[i], VX_THAW, NULL)) {
        perror("ERROR: File system thaw failed");
    }
```



Caching Advisories

VxFS allows an application to set caching advisories for use when accessing files. A caching advisory is the application's preferred choice for accessing a file. The choice may be based on optional performance achieved through the specified advisory or to ensure integrity of user data. For example, a database application may choose to access the files containing database data using direct I/O, or the application may choose to benefit from the file system level caching by selecting a buffered I/O advisory. The application chooses which caching advisory to use.

To set a caching advisory on a file, open the file first. When a caching advisory is requested, the advisory is recorded in memory. This implies that caching advisories do not persist across reboots or remounts. Some advisories are maintained on a per-file basis, not a per-file-descriptor basis, meaning that the effect of setting such an advisory through a file descriptor will impact other processes' access to the same file. This also means that conflicting advisories cannot be in effect for accesses to the same file. If two applications set different advisories, both applications use the last advisory set on the file. VxFS does not coordinate or prioritize advisories.

Some advisories are not cleared from memory after the last close of the file. The recording of advisories remain in memory for as long as the file system metadata used to manage access to the file remains in memory. The removal of file system metadata for the file from memory is not predictable.

All advisories are set using the file descriptor, returned via the `open()` and `ioctl()` calls using the `VX_SETCACHE` `ioctl` command. For details on the use of the `ioctl` commands, see the `vxfsio(7)` manual page.

The caching advisories are described in the following sections.

Direct I/O

Direct I/O is an unbuffered form of I/O for accessing files. If the `VX_DIRECT` advisory is set, the user is requesting direct data transfer between the disk and the user-supplied buffer for reads and writes. This bypasses the kernel buffering of data, and reduces the CPU overhead associated with I/O by eliminating the data copy between the kernel buffer and the user's buffer. This also avoids taking up space in the buffer cache that might be better used for something else, such as application cache. The direct I/O feature can provide significant performance gains for some applications.

For an I/O operation to be performed as direct I/O, it must meet certain alignment criteria. The alignment constraints are usually determined by the disk driver, the disk controller, and the system memory management hardware and software. The file offset must be aligned on a sector boundary (`DEV_BSIZE`). All user buffers must be aligned on a long or sector boundary. If the file offset is not aligned to sector boundaries, VxFS will perform a regular read or write instead of a concurrent read or write.

If a request fails to meet the alignment constraints for direct I/O, the request is performed as data synchronous I/O. If the file is currently being accessed by using memory mapped I/O, any direct I/O accesses are done as data synchronous I/O.

Because direct I/O maintains the same data integrity as synchronous I/O, it can be used in many applications that currently use synchronous I/O. If a direct I/O request does not allocate storage or extend the file, the inode is not immediately written.

The CPU cost of direct I/O is about the same as a raw disk transfer. For sequential I/O to very large files, using direct I/O with large transfer sizes can provide the same speed as buffered I/O with much less CPU overhead.

If the file is being extended or storage is being allocated, direct I/O must write the inode change before returning to the application. This eliminates some of the performance advantages of direct I/O.

The direct I/O advisory is maintained on a per-file-descriptor basis.

Concurrent I/O

Concurrent I/O (`VX_CONCURRENT`) is a form of I/O for file access. This form of I/O allows multiple processes to read or write to the same file without blocking other `read()` or `write()` operations. POSIX semantics requires `read()` and `write()` operations to be serialized on a file with other `read()` and `write()` operations. With POSIX semantics, a read will either read the data before or after the write occurred. With the `VX_CONCURRENT` advisory set on a file, the reads and writes are not serialized similar to character devices. This advisory is usually used by applications that require high performance for accessing data and do not perform overlapping writes to the same file. An example is database applications. Such applications perform their own locking at the application level to avoid overlapping writes to the same region of the file.

It is the responsibility of the application or threads to coordinate write activities to the same file when using the `VX_CONCURRENT` advisory to avoid overlapping writes. The consequence of two overlapping writes to the same file is unpredictable. The best practice for applications is to avoid simultaneous write operations to the same region of the same file.

If the `VX_CONCURRENT` advisory is set on a file, VxFS performs direct I/O for reads and writes to the file. As is the case with direct I/O, concurrent I/O also has the same direct I/O alignment requirements (see “[Direct I/O](#)” on page 52). When concurrent I/O is enabled, the read and write behaves as follows:

- ◆ The `write()` system call acquires a shared read-write lock instead of an exclusive lock.
- ◆ The `write()` system call performs direct I/O to the disk instead of copying and then writing the user data to the pages in the system page cache.



- ◆ The `read()` system call acquires a shared read or write lock and performs direct I/O from disk instead of reading the data into pages in the system page cache and copying from the pages to the user buffer.
- ◆ The `read()` and `write()` system calls will not be atomic. The application must ensure that two threads will not write to the same region of a file at the same time.

VxFS 4.1 introduces support for the `VX_CONCURRENT` caching advisory. This is the first release that provides this API.

Concurrent I/O can be set through the file descriptor and `ioctl()` operation using the `VX_SETCACHE` `ioctl` command with the `VX_CONCURRENT` advisory flag. Only the `read()` and `write()` operations occurring through this file descriptor use concurrent I/O. `Read()` and `write()` operations occurring through other file descriptors will still follow the POSIX semantics. The `VX_CONCURRENT` advisory can be set via the `VX_SETCACHE` `ioctl` descriptor on a file.

Concurrent I/O is a licensable feature of VxFS.

Unbuffered I/O

The I/O behavior of the `VX_UNBUFFERED` advisory is the same as the `VX_DIRECT` advisory set with the alignment constraints as direct I/O. However, for unbuffered I/O, if the file is being extended, or storage is being allocated to the file, metadata updates on the disk for extending the file is not performed synchronously before the write returns to the user. The `VX_UNBUFFERED` advisory is maintained on a per-file-descriptor basis.

Other Advisories

The `VX_SEQ` advisory is a per-file advisory that indicates that the file is being accessed sequentially. A process setting this advisory on a file through its file descriptor will impact the access pattern of other processes currently accessing the same file. When a file with `VX_SEQ` advisory is being read, the maximum read-ahead is performed. When a file with `VX_SEQ` advisory is written, sequential write access is assumed and the modified pages with write operations are not immediately flushed. Instead, modified pages remain in the system page cache and those pages are flushed at some distance point behind the current write point (flush behind).

The `VX_RANDOM` advisory is a per-file advisory that indicates that the file is being accessed randomly. A process setting this advisory on a file through its file descriptor will impact the access pattern of other processes currently accessing the same file. This advisory disables read-ahead with read operations on the file, and disables flush-behind on the file, as described above. The result of disabling flush behind is that the modified pages in the system page cache from the recent write operations are not flushed to the disk until the system pager is scheduled and run to flush dirty pages. The rate at which the system pager is scheduled is based on availability of free memory and contention.

Note The `VX_SEQ` and `VX_RANDOM` are mutually exclusive advisories.

Extents

In general disk space is allocated in 512-byte or 1024-byte (`DEV_BSIZE`) sectors to form logical blocks. VxFS supports logical block sizes of 1024, 2048, 4096, and 8192 bytes. The default block size is 1K for file systems up to 2 TB in size, and 8K for other file system sizes. Users can choose any block when creating file systems using the `mkfs` command. VxFS allocates disk space to files in groups of one or more adjacent blocks called extents. An extent is a set of one or more consecutive logical blocks. Extents allow disk I/O to take place in units of multiple blocks if storage is allocated in consecutive blocks. For sequential I/O, multiple block operations are considerably faster than block-at-a-time operations.

VxFS uses an aggressive allocation policy for allocating extents to files. It also allows an application to pre-allocate space or request contiguous space. This results in improved I/O performance and less file system overhead for performing allocations. For an extending write operation, the policy attempts to extend the previously allocated extent by the size of the write operation or larger. Larger allocation is attempted when consecutive extending write operations are detected. If the last extent cannot be extended to satisfy the entire write operation, a new disjoint extent is allocated. This policy leaves excess allocation that will be trimmed at the last close of the file or if the file is not written to for some amount of time. The file system can still be fragmented with too many non-contiguous extents, especially file systems of smaller size.



Extent Attributes

VxFS allocates disk space to files in groups of one or more extents. In general, the internal allocation policies of VxFS attempt to achieve two goals: allocate extents for optimum I/O performance and reduce fragmentation. VxFS allocation policies attempt to balance these two goals through large allocations and minimal file system fragmentation by allocating from space available in the file system that best fits the data. These extent-based allocation policies provide an advantage over block-based allocation policies. Extent based policies rarely use indirect blocks with allocations and eliminate many instances of disk access that stem from indirect references.

VxFS allows control over some aspects of the extent allocation policies for a given file via two administrative tools, `setext(1)` and `getext(1)`, and an API. The application-imposed policies associated with a file are referred to as extent attributes. VxFS provides APIs that allow an application to set or view extent attributes associated with a file and preallocate space for a file.

Attribute Specifics

There are two basic extent attributes associated with a file: *reservation* and *fixed extent size*. You can preallocate space to the file by manipulating a file's reservation, or override the default allocation policy of the file system by setting a fixed extent size. Other policies determine the way these attributes are expressed during the allocation process. You can specify that:

- ◆ The space reserved for a file must be contiguous
- ◆ No allocations are made for a file beyond the current reservation
- ◆ An unused reservation is released when the file is closed
- ◆ Space is allocated, but no reservation is assigned
- ◆ The file size is changed to incorporate immediately the allocated space

Some of the extent attributes are persistent and become part of the on-disk information about the file, while other attributes are temporary and are lost after the file is closed or the system is rebooted. The persistent attributes are similar to the file's permissions and are written in the inode for the file. When a file is copied, moved, or archived, only the persistent attributes of the source file are preserved in the new file.

Reservation: Preallocating Space to a File

Space reservation is used to make sure applications do not fail because the file system is out of space. An application can preallocate space for all the files it needs before starting to do any work. By allocating space in advance, the file is optimally allocated for performance, and file accesses are not slowed down by the need to allocate storage. This allocation of resources can be important in applications that require a guaranteed response time. With very large files, use of space reservation can avoid the need to use indirect extents. It can also improve performance and reduce fragmentation by guaranteeing that the file consists of large contiguous extents.

VxFS provides an API to preallocate space to a file at the time of the request rather than when data is written into the file. Preallocation, or reservation, prevents any unexpected out-of-space condition on the file system by ensuring that a file's required space is associated with the file before data is written to the file. Storage can be reserved for a file at any time, and reserved space to a file is not allocated to other files in the file system. The API provides the application the option to change the size of the file to include the reserved space.

Reservation does not perform zeroing of the allocated blocks to the file. Therefore, this facility is limited to applications running with appropriate privileges, unless the size of the file is not changed with the reservation request. The data that appears in the newly allocated blocks for the file may have been previously contained in another file.

Reservation is a persistent attribute for the file saved on disk. When this attribute is set on a file, the attribute is not released when the file is truncated. The reservation must be cleared through the same API, or the file must be removed to free the reserved space. At the time of specifying the reservation, if the file size is less than the reservation amount, space is allocated to the file from the current file size up to the reservation amount. When the file is truncated, space below the reserved amount is not freed.

Fixed Extent Size

VxFS uses the I/O size of write requests and the default allocation policy for allocating space to a file. For some applications, the default allocation policy may not be optimal. Setting a fixed extent size on a file overrides the default allocation policies for that file. Applications can set a fixed extent size to match the application I/O size so that all new extents allocated to the file are of the fixed size. By using a fixed extent size, an application can reduce allocation attempts and guarantee optimal extent sizes for a file. With the fixed extent size attribute, an extending write operation will trigger VxFS to extend the previously allocated extent by the fixed extent size amount to maintain contiguity of the extent. If the last extent cannot be extended by the fixed extent size amount, a new disjoint extent is allocated. The size of a fixed extent should factor in the size of file I/O appropriate to the application. Do not use small fixed extent size to eliminate the advantage with extent-base allocation policies.



Another use of a fixed extent size occurs with sparse files. VxFS usually performs I/O in multiples of the system-defined page size. When allocating to a sparse file, VxFS allocates space in multiples of the page size according to the amount of page I/O in need of allocation. If the application always does sub-page I/O, the use of fixed extent size in multiples of the page size reduces allocations.

Applications should not use a large fixed extent size. Allocating a large fixed extent may fail due to the unavailability of an extent of that size, whereas smaller extents are more readily available for allocation.

Custom applications may also use fixed extent sizes for specific reasons, such as the need to align extents to cylinder or striping boundaries on disk.

The fixed extent size attribute is specified in units of file system block size. It specifies the number of contiguous file system blocks to allocate for a new extent, or the number of contiguous blocks to allocate and append to the end of an existing extent. A file with this attribute has fixed size extents or larger extents that are a multiple of the fixed size extent.

Application Programming Interface for Extent Attributes

The current API for extent attributes is `ioctl()`. Applications can open a file and use the returned file descriptor with calls to `ioctl()` to retrieve, set, or change extent attributes. To set or change existing extent attributes, use the `VX_SETEXT` `ioctl`. To retrieve existing extent attributes, if any, use the `VX_GETTEXT` `ioctl`. Applications can set or change extent attributes on a file by providing the attribute information in the structure of type `vx_ext` and passing the `VX_SETEXT` `ioctl` and the address of the structure using the third argument of the `ioctl()` call. Applications can also retrieve existing extent attributes, if any, by passing the `VX_GETTEXT` `ioctl` and the address of the same structure, of type `vx_ext`, as the third argument with the `ioctl()` call.

```
struct vx_ext {
    off_t    ext_size;        /* extent size in fs blocks */
    off_t    reserve;        /* space reservation in fs blocks */
    int      a_flags;        /* allocation flags */
}
```

The `ext_size` argument is set to specify a fixed extent size. The value of fixed extent size is specified in units of the file system block size. Be sure the file system block size is known before setting the fixed extent size. If a fixed extent size is not required, use zero to allow the default allocation policy to be used for allocating extents. The fixed extent allocation policy takes effect immediately after successful execution of the `VX_SETEXT` `ioctl`. An exception is with files that already contain indirect blocks, in which case the fixed extent policy has no effect unless all current indirect blocks are freed via file truncation.

The *reserve* argument can be set to specify the amount of space preallocated to a file. The amount is specified in units of the file system block size. Be sure the file system block size is known before setting the preallocation amount. If a file has already been preallocated, its current reservation amount can be changed with the `VX_SETEXT` ioctl. If the specified reserve amount is greater than the current reservation, the allocation for the file is increased to match the newly specified reserve amount. If the reserve amount is less than the current reservation, the reservation amount is decreased and the allocation is reduced to the newly set reservation amount or the current file size. Note that file preallocation requires root privilege, unless the size of the file is not changed (see the `VX_CHGSIZE` flag), and the preallocation size cannot be increased beyond the `ulimit` of the requesting process. See the `ulimit(2)` manual page for more information.

Allocation Flags

Allocation flags can be specified with `VX_SETEXT` ioctl for additional control over allocation policies. Allocation flags are specified in the *a_flag* argument of `vx_ext` structure to determine:

- ◆ Whether allocations are aligned
- ◆ Whether allocations are contiguous
- ◆ Whether the file can be written beyond its reservation
- ◆ Whether an unused reservation is released when the file is closed
- ◆ Whether the reservation is a persistent attribute of the file
- ◆ When the space reserved for a file will actually become part of the file.

Allocation flags with Reservation

The `VX_TRIM`, `VX_NOEXTEND`, `VX_CHGSIZE`, `VX_NORESERVE` and `VX_CONTIGUOUS` flags can be used to modify reservation requests. Note that `VX_NOEXTEND` is the only flag that is persistent; the other flags may have persistent effects, but they are not returned by the `VX_GETTEXT` ioctl. The non-persistent flags remain active for a file in the file system cache until the file is no longer accessed and is removed from the cache.

Reservation Trimming

The `VX_TRIM` flag specifies that the reservation amount must be trimmed to match the file size when the last close occurs on the file. At the last close, the `VX_TRIM` flag is cleared and any unused reservation space beyond the size of the file is freed. This can be useful if an application needs enough space for a file, but it is not known how large the file will become. Enough space can be reserved to hold the largest expected file, and when the file has been written and closed, any extra space will be released.



Non-Persistent Reservation

If reservation is not desired to be a persistent attribute, the `VX_NORESERVE` flag can be specified to request allocation of space without making reservation a persistent attribute of the file. This flag can be used by applications interested in temporary reservation but wish to free any space past the end of the file when the file is closed. For example, if an application is copying a file that is 1 MB long, it can request a 1 MB reservation with the `VX_NORESERVE` flag set. The space is allocated, but the reservation in the file is left at 0. If the program aborts for any reason or the system crashes, the unused space past the end of the file is released. When the program finishes, there is no clean up because the reservation was never recorded on disk.

No Write Beyond Reservation

The `VX_NOEXTEND` flag specifies that any attempt to write beyond the current reservation must fail. Writing beyond the current reservation requires the allocation of new space for the file. To allocate new space to the file, the space reservation must be increased. This can be used similar to the function of the `ulimit` command to prevent a file from using too much space.

Contiguous Reservation

The `VX_CONTIGUOUS` flag specifies that any space allocated to a file must satisfy the requirement of a single extent allocation. If there is not one extent large enough to satisfy the reservation request, the request fails. For example, if a file is created and a 1 MB contiguous reservation is requested, the file size is set to zero and the reservation to 1 MB. The file will have one extent that is 1 MB long. If another reservation request is made for a 3 MB contiguous reservation, the new request will find that the first 1 MB is already allocated and allocate a 2 MB extent to satisfy the request. If there are no 2 MB extents available, the request fails. Extents are, by definition, contiguous. Note that because `VX_CONTIGUOUS` is not a persistent flag, space will not be allocated contiguously for restoring a file that was previously allocated with the `VX_CONTIGUOUS` flag.

Include Reservation in the File Size

A reservation request can affect the size of the file to include the reservation amount by specifying `VX_CHGFSIZE`. This flag increases the size of the file to match the reservation amount without zeroing the reserved space. Because the effect of this flag is uninitialized data in a file, which might have been previously contained in other files, the use of this flag is restricted to users with the appropriate privileges. Without this flag, the space of the reservation is not included in the file until an extending write operation requires the space. A reservation that immediately changes the file size can generate large temporary files. Applications can benefit from this type of reservation by eliminating the overhead imposed with write operations to allocate space and update the size of the file.

It is possible to use these flags in combination. For example, using `VX_CHGSIZE` and `VX_NORESERVE` changes the file size, but does not set any reservation. When the file is truncated, the space is freed. If the `VX_NORESERVE` flag is not used, the reservation is set on the disk along with the file size.

Allocation Flags With Fixed Extent Size

The `VX_ALIGN` flag can be used to specify an allocation flag for fixed extent size. This flag has no effect if it is specified with a reservation request. The `VX_ALIGN` specifies the alignment requirement for allocating future extents aligned on a fixed extent size boundary relative to the start of the allocation unit. This can be used to align extents to disk striping boundaries or physical disk boundaries. The `VX_ALIGN` flag is persistent and is returned by the `VX_GETTEXT` ioctl.

How to use Extent Attribute APIs

First, verify that the target file system is VxFS, and then determine the file system block size using the `statfs()` call. The type for VxFS is `MNT_VXFS` on most platforms, and the file system block size is returned in `statfs.f_bsize`. The block size must be known for setting or interpreting the extent attribute information through VxFS extent attribute APIs.

Each invocation of the `VX_SETEXT` ioctl affects all the elements in the `vx_ext` structure. When using `VX_SETEXT`, always use the following procedure:

1. Call the `VX_GETTEXT` ioctl to read the current settings, if any.
2. Modify the current values to be changed.
3. Call the `VX_SETEXT` ioctl to set the new values.

Caution Follow this procedure carefully. A fixed extent size may be inadvertently cleared when the reservation is changed. When copying files between VxFS and non-VxFS file systems, the extent attributes cannot be preserved. Note that the attribute values returned for a file in a `vx_ext` structure will have a different effect on another VxFS file system with different a file system block size from the source file system. Translation of attribute values for different block sizes may be necessary when copying files with attributes between two file systems of a different block size.



The following is an example code snippet for setting the fixed extent size of the `MY_PREFERRED_EXTSIZE` attribute on a new file, `MY_FILE`, assuming `MY_PREFERRED_EXTSIZE` is multiple of the file system block size:

```
#include <sys/fs/vx_ioctl.h>

struct vx_ext myext;

fd = open(MY_FILE, O_CREATE, 0644);

myext.ext_size = MY_PREFERRED_EXTSIZE;
myext.reserve = 0;
myext.flags = 0;

error = ioctl(fd, VX_SETTEXT, &myext);
```

The following is an example code snippet for preallocating `MY_FILESIZE_IN_BYTES` bytes of space on the new file, `MY_FILE`, assuming the target file system block size is `THIS_FS_BLOCKSIZE`:

```
#include <sys/fs/vx_ioctl.h>

struct vx_ext ext;

fd = open(MY_FILE, O_CREATE, 0644);

myext.ext_size = 0;
myext.reserve = (MY_FILESIZE_IN_BYTES + THIS_FS_BLOCKSIZE) +
                THIS_FS_BLOCKSIZE;
myext.flags = VX_CHGSIZE;
error = ioctl(fd, VX_SETTEXT, &myext);
```


Index

A

- Activation Time 12
- Allocation Flags 59
- Allocation Flags With Fixed Extent Size 61
- Allocation Policies 27
 - Multi-Volume Support 32
- Alternate Namespace 41
- Application Interface 2

C

- Caching Advisories 52
- close 8, 20
- Compiling Environment 5
- Concurrent I/O 53
- ctime 15

D

- Data Copy 52
- Data Transfer 52
- DEV_BSIZE 52, 55
- Direct Data Transfer 52
- Direct I/O 52

E

- enum 15
- Extent Attributes 56
- Extents 55

F

- fc_atime 12, 13
- fc_foff 9, 13, 19
- fc_loff 9, 13
- fc_magic 13
- fc_padding 13
- fc_state 13
- fc_sync 13
- fc_version 13, 16
- fchdir 45
- fchroot 45
- fcl_chgtype_t 16

- fcl_ent 14

- Fields 14

- fcl_keeptime 19

- fcl_maxalloc 19

- fcl_winterval 20

- fe_dino 14, 15

- fe_gen 15

- fe_ino 14, 15

- fe_nmlen 15

- fe_time 15

- fe_type 15

- Features 2

- File Change Log 3, 7

- File Change Log File 8

- Programmatic Interface 20

- Record Details 14

- Records With a Trailing File

- Name 16

- Records Without a Trailing File

- Name 19

- Record Types 10

- Obsolete Records 11

- Special Records 10

- Superblock 12

- Fields 13

- Tunables 19

- File Change Log File 8

- First Valid Offset 9

- Fixed Extent Size 56, 57

- Freeze/Thaw 50

- FSAP_INHERIT 32

- fsapadm 27

- fsvoladm 27

G

- getcwd 45

- getdents 43, 44

- gettext 56



H

Header Files 4

I

I/O

Direct 52

Sequential 53

Synchronous 53

ioctl 2, 52, 58

L

Last Valid Offset 9

Libraries 4

libvxfsutil 20

Logical Blocks 55

lseek 8, 20

M

mkfs 55

mmap 41, 43

Multi-Volume Support 3, 27

Allocation Policy APIs 32

Creating and Assigning Policies 34

Data Structures 36

Enforcing the Policy on a File 35

Examples 37

Examples of Volume Set Operations 30

Modifying a Volume Within a File System 31

Querying the Defined Policies 35

Querying the Volume Set for a File System 30

Uses 29

Volume APIs 30

Volume Encapsulation 31

N

Named Attributes 41

Named Data Streams 41

Behavior Changes in Other System

Calls 45

Example 46

Listing 44

Namespace 45

Programmatic Interface 43, 47

Programmer's Reference 47

ncheck 24

O

Obsolete Records 11

open 8, 20, 41, 43, 52

Other Advisories 54

R

read 8, 20, 41, 43, 53

readdir 43

Record Details 14

Record Types 10

Obsolete Records 11

Special Records 10

Records With a Trailing File Name 16

Records Without a Trailing File Name 19

Reservation 56, 57

Reverse Path Name Lookup 23

S

Sequential I/O 55

sequential I/O 53

setext 56

Software Developer's Kit 1
Packages 3

Special Records 10

statfs 61

statvfs 11

Storage Checkpoints 29

Superblock 8, 12
Fields 13

Synchronous I/O 53

U

ulimit 59

Unbuffered I/O 54

Using Extent Attribute APIs 61

V

Volume APIs 30

Volume Set 30

VOP_SETATTR 45

VRTSfsmnd 3

VRTSfssdk 3

VX_ALIGN 61

VX_CHGSIZE 59, 60

VX_CONCURRENT 53

VX_CONTIGUOUS 59, 60

VX_DIRECT 54

vx_ext 58, 61

VX_FCL_HEADER 11, 15, 16

VX_FCL_INFO_CHG 11

VX_FCL_ITIMES_CHG 11

VX_FCL_LINK 9, 15, 16

VX_FCL_MAGIC 13

VX_FCL_NO_CHANGE 11, 15, 16

- VX_FCL_RENAME 9, 15, 16
- VX_FCL_UNLINK 9, 15, 16
- VX_FREEZE 50
- VX_FREEZE_ALL 50
- VX_GETEXT 58, 61
- VX_NOEXTEND 59, 60
- VX_NORESERVE 59, 60
- VX_RANDOM 55
- VX_SEQ 54
- VX_SETCACHE 54
- VX_SETEXT 58, 61
- VX_THAW 50
- VX_TRIM 59
- VX_UNBUFFERED 54
- VxFS I/O 3, 49
 - Caching Advisories 52
 - Concurrent I/O 53
 - Direct I/O 52
 - Other Advisories 54
 - Unbuffered I/O 54
 - Extents 55
 - Allocation Flags 59
 - Allocation Flags With Fixed Extent
 - Size 61
 - API 58
 - Attribute Specifics 56
 - Extent Attributes 56
 - Fixed Extent Size 57
 - Reservation 57
 - Using Extent Attribute APIs 61
 - Freeze/Thaw 50
 - vxfs_fcl_sync 13, 20, 21
 - vxfs_inotopath 24
 - vxfs_inotopath_gen 14, 15, 23
 - vxfs_nattr_link 44
 - vxfs_nattr_open 43
 - vxfs_nattr_rename 44
 - vxfs_nattr_unlink 44
 - vxfs_nattr_utimes 44
 - vxfsio 52
 - vxtunefs 19, 21
 - vxvset 30

W

- write 41, 43, 53

