Veritas™ File System Programmer's Reference Guide

Solaris

5.0

N18542F

Veritas File System Programmer's Reference Guide

Veritas File System 5.0
PN: N18542F

Symantec Corporation
20330 Stevens Creek Blvd.
Cupertino, CA 95014
www.symantec.com

Printed in the United States of America.

Third-party legal notices

Licensing and registration

Technical support

For technical assistance, visit http://support.veritas.com and select phone or email support. Use the Knowledge Base search feature to access resources such as TechNotes, product alerts, software downloads, hardware compatibility lists, and our customer email notification service.

Contents

Chapter 5 VxFS I/O Application Interface

Chapter 1

Veritas File System software developer's kit

This chapter includes the following topics:

About the software developer's kit

Veritas File System (VxFS) Software Developer's Kit (SDK) provides developers with the information necessary to use the application programming interfaces (APIs) to modify and tune various features and components 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 5.0 and subsequent releases.

The APIs in Chapter 5, "VxFS I/O Application Interface" on page 69 are available in VxFS 4.0, subsequent releases, and several releases prior.

File system software developer's kit features

This section provides an overview of the VxFS features that are accessible with the SDK.

API library interfaces

The API library interfaces highlighted in this SDK are the `vxfsutil` library and VxFS IOCTL directives. The library contains a collection of API 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:

APIs	Feature
inotopath	Inode-to-path lookup
nattr	Named Data Stream
FCL	File Change Log
MVS	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 stubs library and dynamic library combination. Applications are compiled with the stubs library `libvxfsutil.a`, making the application portable to any VxFS target environment. The application can then be run on a VxFS target, and the stubs library will find the dynamic library provided with the VxFS target.

The stubs library uses 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 15.

Multi-volume support

The multi-volume support (MVS) feature allows a VxFS file system to use multiple Veritas™ Volume Manager (VxVM) volumes as underlying storage. Administrators and applications can control where files go to maximize effective performance, while minimizing cost. This feature can be used only in conjunction with Veritas Volume Manager. In addition, some of the functionality requires additional license keys.

See "Multi-volume support" on page 49.

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 to develop and compile applications. The `VRTSfsmnd` package contains this Guide and the API man pages.

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

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

The `VRTSfssdk` and `VRTSfsmnd` packages can be obtained separately from the VxFS package. To run the applications or sample programs, a licensed VxFS target is required. In addition, the VxFS license of the required features should be installed on the target system.

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/5.0/lib/libvxfsutil.a`
- `/opt/VRTSfssdk/4.1/lib/sparcv9/libvxfsutil.a`
- `/opt/VRTSfssdk/5.0/include/vxfsutil.h`
- `/opt/VRTSfssdk/5.0/include/sys/fs/fcl.h`
- `/opt/VRTSfssdk/5.0/include/sys/fs/vx_ioctl.h`

There are also symlinks to these files from the standard Veritas paths: `/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. 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/5.0 file system. Some may require a file system mounted on a Veritas Volume Set.

Note: Some programs may require special volume configurations (volume sets). In addition, some programs require a file system to be mounted on a volume set.

Recompiling with a different compiler

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

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

To recompile 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, compile it as follows:
   ```
   # gcc -I /opt/VRTS/include -L /opt/VRTS/lib -ldl -o MyApp \
   MyApp.c libvxfsutil.a
   ```

To compile the src or libsrc directory, edit the `/opt/VRTSfssdk/5.0/make.env` file as follows:

1. Select the compiler path on your local system. choose and edit CC to the path on your system.
   `CC=/opt/SUNWspro/bin/cc` (Whatever path is appropriate)
   `#CC=/usr/local/bin/gcc`
2. Change to the `src` or `libsrc` and type:
   ```
   # gmake (or make)
   ```

Chapter 2

File Change Log

This chapter includes the following topics:

About the File Change Log file

The VxFS File Change Log (FCL) tracks changes to files and directories in a file system. Applications that typically use the FCL are usually required to:

- Scan an entire file system or a subset
- Discover changes since the last scan

These applications may include: backup utilities, webcrawlers, search engines, and replication programs.

Note: The FCL tracks when the data has changed and records the change type, but does not track the actual data changes. It is the responsibility of the application to examine the files to determine the changed data.

Recorded changes

The File Change Log records file system changes including:

- Creates
- Links
- Unlinks
- Renaming
- Data appended
- Data overwritten
- Data truncated
- Extended attribute modifications
- Holes punched
- Miscellaneous file property updates

Note: The FCL is supported only on disk layout Version 6 and 7.

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, however, some user-level operations are prohibited, such as writes.

Note: The standard system calls `open`(2), `lseek`(2), `read`(2) and `close`(2) can access the data in the FCL file. All other system calls such as `mmap`(2), `unlink`(2), `ioctl`(2) etc. are not allowed on the FCL file.

Warning: For compatibility with future VxFS releases, the FCL file might be pulled out of the namespace, and these standard system calls may no longer work. Therefore, it is recommended that all new applications be developed using the programmatic interface.

See "Programmatic interface" on page 26.

Using the FCL file

VxFS tracks changes to the file system by appending the FCL file with information pertaining to those changes. This enables you to do the following:

- Use the FCL to determine the sequence of operations that have been performed on the file system in general or on a specific file after a particular point in time. For instance, an incremental backup application can scan the FCL file to determine which files have been added or modified since the file system was last backed up.
- Configure the FCL to track additional information such as file opens, I/O statistics, access information (for example, user ID) along with other changes
  You can then use this information to gather the following:
  - Space usage statistics to determine how the space usage for different types of data
  - Usage profile for the different files on a file system across different users to help determine which data has been recently accessed and by whom

  Note: These are new features for the VxFS 5.0 release.

Space usage

You can use the FCL to track space usage when a file system gets close to being full. The FCL file can be searched for recently created files (file creates) or write records to determine newly added files or existing files that have grown recently. Depending on the application needs, the search can be done on the entire FCL file, or on a portion of the FCL file corresponding to a specific time frame.

Additionally, you can look for files created with particular names. For example, if users are downloading *.mp3 files that are taking up too much space, the FCL file can be read to find files created with the name *.mp3.

Full system scan reductions

VxFS creates and logs an FCL record for every update operation performed on an FCL-enabled file system. These operations include: creates, deletes, rename, mode changes, and writes. Therefore, incremental backup applications or applications which maintain an index of a file system based on the filename, file attributes, or content, can avoid a full system scan by reading the FCL file to detect the files that have changed since the previous backup or previous index update.

File history traces

You can trace a file's history by scanning the FCL file and coalescing FCL record sequences for a file. You can also use the related FCL records from a file's creation, attribute changes, write records and the file's deletion to track the file's history.

FCL logging activation

By default, FCL logging is deactivated and can be activated on a per file system basis using the `fcladm` command.

See the `fcladm`(1M) manual page.

When the FCL is activated, new FCL records are appended to the FCL file as the file system changes occur. When the FCL is turned off, further recording stops, but the FCL file remains at the `lost+found/changelog`. You can only remove an FCL file by using the `fcladm` command.

The FCL has an associated version that represents the layout or the internal representation of the FCL, along with the list of events recorded in the FCL. Whenever a new version of VxFS is released, the following occurs:

- There may either be additional events recorded in the FCL
- The internal representation of the FCL may change

This results in the FCL version getting updated. For instance, in VxFS 4.1, the default was Version 3. In Vxfs 5.0, the default is Version 4. FCL Version 4 records additional sets of events that are not available in Version 3 (such as file opens). To provide backward compatibility for applications developed on VxFS 4.1, VxFS 5.0 provides an option to specify an FCL version (either 3 or 4) during activation. Depending on the specified version, the logging of the new record types is either allowed or disallowed.

The logging of most of the newly added records in VxFS 5.0 (for example, file opens, I/O statistics etc.) is optional and is turned off by default. Recording of these events can be enabled or disabled using the *set|clear* options of the `fcladm` command.

The FCL meta-information comprising the file system state, version, and the set of events being tracked is persistent across reboots and file system unmounts or mounts. The version and event information is also persistent across re-activations of the FCL.

FCL file layout

In VxFS 4.1, the internal layout of the FCL file was exposed to the user and the applications were expected to access the FCL using standard file system interfaces such as `open`(2), `read`(2), and `lseek`(2). However, this methodology may lead to future compatibility issues. For example, if the underlying FCL layout and the FCL version changes, the application must be changed and recompiled to accommodate these changes.

VxFS 5.0 introduces a new programming interface, which provides improved compatibility, even when the on-disk FCL layout changes. With this API, the FCL layout is not a concern for applications. Consequently, this section provides only a rudimentary description of the FCL layout.

The FCL file is usually a sparse file containing the FCL superblock and the FCL records. The first information block in the FCL file is the FCL superblock. This block may be followed by an optional hole as well as the FCL records which contain information about the changes in the file system.

Figure 2-1 depicts the FCL file format.

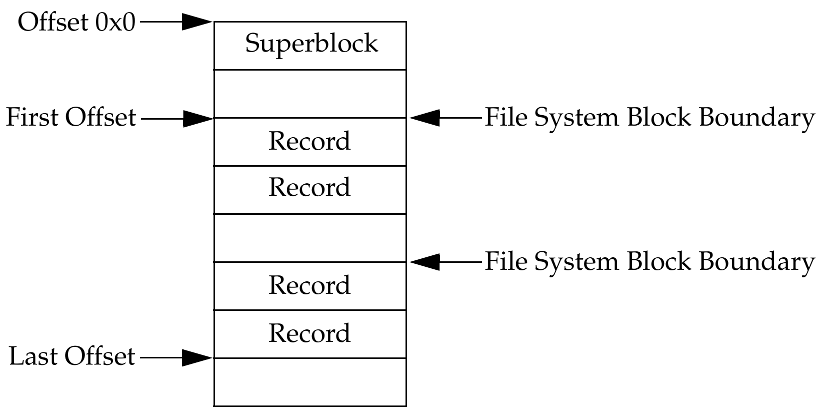


Figure 2-1 FCL file format

FCL superblock

Changes to files and directories in the file system are stored as FCL records. The superblock, which is currently stored in the first block of the FCL file, describes the state of the FCL file. The superblock indicates the following:

- Whether FCL logging is enabled
- What time it was activated
- The current offsets of the first and last FCL records
- The FCL file version
- The event mask for the set of events currently being tracked
- The time that the event mask was last changed

The FCL file containing just the superblock is created when FCL is first activated using the `fcladm on` command. The superblock gets deleted only when the FCL file is removed using the `fcladm rm` command.

When the FCL is activated using `fcladm on`, the state in the superblock and its activation time are changed. Whenever any file system activity results in a record being appended to the FCL file, the last offset gets updated.

As the FCL file grows in size, depending on the file system tunables (`fcl_maxalloc` and `fcl_keeptime`), the oldest records at the start of the FCL file are thrown away to free up some space, as the first offset gets updated. When the set of events tracked in the FCL is changed using `fcladm` *`set|clear`*, it results in the event mask and the event mask change time getting updated, as well. An event mask change also results in a event mask change record containing the old event mask and the new one being logged in the FCL file.

FCL record

The FCL records contain information about these typical changes:

- The inode number of the file that has changed
  See "Inodes" on page 46.
- The time of change
- The type of change
- Optional information depending on the record type
  Depending on the record type, the FCL may also include the following:
  - A parent inode number
  - A filename for file deletes or links, etc.

- A command name for a file open record
- The actual statistics for an I/O statistics record, etc.

See Figure 2-1 on page 20.

Record types

Table 2-1 lists actions that generate FCL record types.

Table 2-1 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 named 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

Table 2-1 FCL record types

Action to create an FCL record	Record type
Set mtime 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`
Open a file	VX_FCL_FILEOPEN
Write I/O statistics of a file to FCL	VX_FCL_FILESTATS
Change the set of events tracked in the FCL	VX_FCL_EVNTMSK_CHG

Note: When the `fcladm on` command activates the FCL, all the events listed in Table 2-1 are recorded by default, except `fileopen` and `filestat`. Access information for each of these events is also not recorded by default. Use the *`set`* option of the `fcladm` command to record opens, I/O statistics and access information.
See the `fcladm`(1M) manual page.

The record types Table 2-1 belong to `fcl_chgtype.t`. `fcl_chgtype.t`, which is an enumeration that is defined in the `fcl.h` header file in Table 2-2 on page 39.

Special records

The following record types are no longer visible through the API:

- `VX_FCL_HEADER`
- `VX_FCL_NOCHANGE`
- `VX_FCL_ACCESSINFO`

Typical record sequences

The life cycle of a file in a file system is recorded in the FCL file from creation to deletion. The following is a typical sequence of FCL records written to the log for creating an FCL file:
`VX_FCL_CREATE`
`VX_FCL_FILEOPEN` (if tracking file opens is enabled)
`VX_FCL_DATA_EXTNDWRITE`
`VX_FCL_IMODE_CHG`

When writing a file, one of the following FCL records is written to the log for every write operation. The record depends on whether the write is past the current end of the file or within the file.

```
VX_FCL_DATA_EXTNDWRITE
VX_FCL_DATA_OVERWRITE
```

The following shows a typical sequence of FCL records written to the log, when file “a” is renamed to “b” and both files are in the file system:

`VX FCL_UNLINK` (for file “b” if it already exists)

`VX_FCL_RENAME` (for rename from “a” to “b”)

FCL tunables

You can set four FCL tunable parameters using 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 the 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 and 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 should be set to any value greater than the time between FCL scans. For example, if the FCL is scanned every 24 hours, `fcl_keeptime` could be set to 25 hours. This prevents FCL records from being purged before they are read and processed.
`fcl_maxalloc`	Specifies the maximum amount 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 is updated in the FCL superblock. 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 multiple overwrite, extending write, or truncation records for the same inode. This helps to reduce the number of repetitive records in the FCL. The `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.
`fcl_ointerval`	Specifies the time interval in seconds within which subsequent opens of a file do not produce an additional FCL record. This helps to reduce number of repetitive file-open records logged in the FCL, especially in the case of frequent accesses through NFS. If the tracking of access information is also enabled, a subsequent file open event within `fcl_ointerval` might produce a record, if the latter open is by a different user. Similarly, if an inode goes out of cache and returns, or if there is an FCL sync, there might be more than one file open record within the same open interval. The default value is 600 seconds. Tuning recommendations: If the application using file-open records only needs to know if a file has been accessed by any user from the last time it scanned the FCL, `fcl_ointerval` can be set to a time period in the range of the time between the scans. If the application is interested in tracking every access, the tunable can be set to zero. In the case where the file system is extensively accessed over NFS, depending on the platform and the NFS implementation, there might be a large number of file open records logged. In such cases, it is recommended to set the tunable to a higher value to avoid flooding the FCL with repetitive records.

How tunables handle FCL growth size

Figure 2-2 illustrates an example of record purging as an FCL file grows in size. The FCL file on the left contains 8K blocks and no holes. When activity occurs on the file system, it is recorded in the FCL and the growth results in the FCL file on the right.

When the FCL size reaches the maximum allowable size that is specified by the `fcl_maxalloc` tunable, older records are purged and space is freed. The FCL program only purges records that are older than a time specified by `fcl_keeptime`.

The freed space is always in units of an internal hole size. In Figure 2-2, the file system frees up space in the FCL in 8K units. When the FCL file surpasses the maximum allocation for the first time and the number of older records is 20K, the program purges 16K. This leaves a 16K hole following the FCL superblock. The first valid offset in the FCL superblock is then updated to 24K.

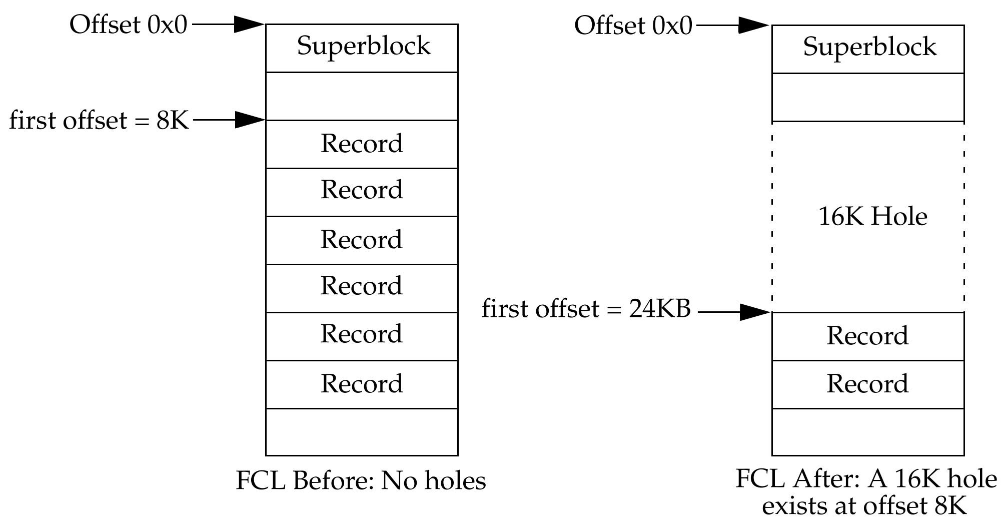


Figure 2-2 FCL record purging example

Programmatic interface

In addition to the existing programmatic interface exposed through `libvxfsutil: vxfs_fcl_sync`,VxFS 5.0 provides a new set of programmatic interfaces which replace the mechanism to access an FCL file via the set of standard system calls: `open`(2), `lseek`(2), `read`(2) and `close`(2). This API provides the following improvements.

Ease of use

The API reduces the need to write additional code to parse FCL entries. Most of the on-disk FCL records are of a fixed size and contain only the default information such as the inode number or time stamp. However, some records can be of variable sizes (for example, a file remove or rename record). These records contain additional information such as the name of the file removed or renamed.

To ensure that the first few bytes at the start of any file system block is always a valid FCL record (if the filename crosses a block boundary), the file system block may be split across multiple on-disk records. Previously, you were required to write additional code to assemble these records to get the filename. The VxFS 5.0 API provides a mechanism to directly read a single assembled logical record. This makes it easier for applications using the API. The API also lets the application specify a filter to indicate a subset of the events of interest and return only required records.

Backward compatibility

The API lets applications read the FCL independent of the FCL layout changes. For example, consider a scenario where an application directly accesses and interprets the on-disk FCL records. If the next VxFS release adds new records or changes the way the records are stored in the FCL file, the application needs to be rewritten or at least recompiled to accommodate for the changes (under previous VxFS versions).

With an intermediate API, the on-disk layout of FCL is hidden from the application, so even if the disk layout of FCL changes, the API internally translates the data returns the expected output record to the user. The user application can then continue without a recompilation or a rewrite.This insulates programs from FCL layout changes and provides greater compatibility for existing applications.

API functions

The API uses the following type of functions:

- Functions for accessing FCL records
- Functions for seeking offsets and time stamps

Functions for accessing FCL records

These are general functions for accessing FCL records:

`vxfs_fcl_open`	Opens the FCL file and returns a handle which can be used for further operations. All subsequent accesses of the FCL file through the API must use this handle.
`vxfs_fcl_close`	Closes the FCL file and cleans up resources associated with the handle
`vxfs_fcl_getinfo`	Returns the FCL version number along with the state (on/off) of the FCL file
`vxfs_fcl_read`	Reads FCL records of interest to the user into a buffer passed in by the user
`vxfs_fcl_copyrec`	Copies an FCL record. If the source record contains pointers, it relocates them to point to the new location.

Functions for seeking offsets and time stamps in the FCL

Users have the option to seek to a particular point in the File Change Log based on the offset from where they left off, or to the first record after a specified time. The following functions can seek offsets and time stamps in the FCL:

`vxfs_fcl_getcookie`	Returns an opaque structure (referred hereinafter as a cookie) which embeds the current FCL activation time and the current offset. This cookie can be saved and later passed into `vxfs_fcl_seek` to continue reading from where the application left off last time.
`vxfs_fcl_seek`	Extracts data from the cookie passed and seeks to the specified offset. A cookie is embedded with the FCL activation time and file offset
`vxfs_fcl_seektime`	Seeks to the first record in the FCL after the specified time

vxfs_fcl_open

`int vxfs_fcl_open(`*`char *pathname, int flags, void **handle`*`);`

This function opens the FCL file and returns a handle which should be used for all further accesses to the FCL through the API (for example, `vxfs_fcl_read`, `vxfs_fcl_seek`, etc.).

`vxfs_fcl_open` has two parameters: *`*pathname`* and *`**handle`*. The *`*pathname`* can either be a pointer to an FCL filename or a mount point. If *`*pathname`* is a mount point, `vxfs_fcl_open` automatically determines if the FCL is activated on the mount point and opens the FCL file associated with the mount point (currently *`mount_point/`*`lost+found/changelog`).

`vxfs_fcl_open` then determines if it is a valid FCL file, and if the FCL file version is compatible with the library. The `vxfs_fcl_open` function then assimilates meta-information about the FCL file into an opaque internal data structure and populates *`**handle`* with a pointer.

Just like the `lseek`(2) and `read`(2) system calls, the FCL file *`**handle`* has an internal offset to indicate the position in the file from where the next read starts. When the FCL file is successfully opened, this offset is set to the first valid offset in the FCL file.

Return Value

Upon successful completion, a “0” is returned to the caller and the handle is non-NULL. Otherwise, the API returns a non-zero value is and the handle is set to NULL. The global value **errno** is also set to indicate the error.

vxfs_fcl_close

`vxfs_fcl_close` closes the FCL file referenced by the handle. All data structures allocated with this handle are cleaned. You should not use this handle after a call to `vxfs_fcl_close`.

Parameters

`void vxfs_fcl_close(`*`void *handle`*`)`

`*handle` is a valid handle returned by the previous call to `vxfs_fcl_open`.

vxfs_fcl_getinfo

`int vxfs_fcl_getinfo(`*`void *handle, struct fcl_info*fclinfo`*`);`

The `vxfs_fcl_getinfo` function returns information about the FCL file in the FCL information structure pointed to by *`fcl_info`*. It obtains this information from the FCL superblock.

```
struct fcl_info {
   uint32_tfcl_version;
```

```
    uint32_tfcl_state;
};
```

An intelligent application that is aware of the record types associated with each FCL version can use *fcl_version* to determine whether the FCL file contains the needed information. For example, with a Version 3 FCL, an intelligent application can infer that there is no access information in the FCL record. In addition, if the `fcl_state` is `FCLS_OFF`, the application can also infer that there are no records added to the FCL file due to file system activity.

Return Values

A "0" indicates success; otherwise, the **errno** is set to error and a non-zero value is returned.

vxfs_fcl_read

This function lets the application read the actual file or directory change information recorded in the FCL as logical records. Each record returns a `struct fcl_record` type. `vxfs_fcl_read` lets the application specify a filter comprising a set of desired events.

Parameters

```
int vxfs_fcl_read(void *hndl, char *buf, size_t *bufsz,
uint64_t eventmask, uint32_t *nentries);
```

Input

This function has the following input:

- **hndl* is a pointer returned by a previous call to `vxfs_fcl_open`
- **buf* is a pointer to a buffer of size at least **bufsz*
- **bufsz* specifies the buffer size
- *eventmask* is a bit-mask that specifies a set of events which is of interest to the application. It should be a "logical or" of a set of event masks specified in the `fcl.h` header. For example, if the *eventmask* is (`VX_FCL_CREATE_MASK | VX_FCL_UNLINK_MASK`), `vxfs_fcl_read` returns only file create and delete records. If an application needs to read all all the records listed in Table 2-1 on page 21, it can specify a default *eventmask* mask as `FCL_ALL_V4_EVENTS`. This returns all valid Version 4 FCL records in the FCL file.

Note: If VX_FCL_EVNTMASKCHG_MASK is set in *eventmask* and the records returned by vxfs_fcl_read contain a VX_FCL_EVNTMASK_CHG record, it is always the last record in the buffer. This lets the application readjust the *eventmask* if required. In addition, if the application discovers from the *eventmask* change record that a particular event is no longer recorded, it can decide to stop further reading.

- **nentries* specifies the number of entries that should be read into the buffer in this call to vxfs_fcl_read. If **nentries* is "0," vxfs_fcl_read reads as many entries as will fit in the buffer.

Output

**buf* contains **nentries* FCL records of the struct fcl_record type if there is no error.

If the requested number of entries cannot fit in a buffer of the passed size, an FCL_ENOSPC error is returned. In this case, **bufsz* is updated to contain the buffer size required for the requested number of records. The application may use this to reallocate a larger sized buffer and invoke vxfs_fcl_read again. **bufsz* is not changed if there is no error.

**nentries* is updated to contain the number of entries read in the buffer when vxfs_fcl_read is called and there is no error. **nentries* and the returned value are both zero when the application has reached the end of file and there are no more records to be read.

Return Values

A "0" indicates success; a non-zero indicates an error.

Note: FCL_ENOSPC is returned if there is not enough space in the buffer to store the current record. The minimum size the buffer must be is returned in **bufsz.*

After a successful call to vxfs_fcl_read, the current file position is advanced, so that the next call to vxfs_fcl_read reads the next set of records.

vxfs_fcl_copyrec

vxfs_fcl_copyrec copies an FCL record of length *len* from **src* to **tgt* and relocates the pointers in the target FCL record to point to copies of the data in **src*. A special operation to copy FCL records is needed because a simple memory copy of the FCL record from **src* to **tgt* leaves the pointers in **tgt* pointing to the data in source FCL record, instead of copies of the data. This may cause problems when the memory for the source FCL record is re-used.

Parameters

`int vxfs_fcl_copyrec(`*`struct fcl_record *src`*`,`*`struct fcl_record *tgt`*`,`*`size_t len`*`);`

- *`*src`* must point to a valid *`fcl_record`* structure
- The *`len`* passed should be the length of the source FCL record, represented by the *`fr_reclen`* field of the source FCL record.
- *`*tgt`* must point to a non-null memory address used to hold the target FCL record.

Note: The caller of `vxfs_fcl_copyrec` must ensure that the space required to hold the target FCL record is allocated (i.e., at least the *`len`* bytes).

---

vxfs_fcl_getcookie

The `vxfs_fcl_getcookie` and `vxfs_fcl_seek` functions are effective methods for remembering a position in the FCL file that the application has processed earlier. This then can be used as a restarting point. This is a highly useful tool for applications.

See "vxfs_fcl_seek" on page 32.

The `vxfs_fcl_getcookie` function returns an opaque `fcl_cookie` structure which embeds information comprising the current activation time of the FCL file and an offset indicating the current position in the FCL file. This cookie can be passed into `vxfs_fcl_seek` to seek to the position in the FCL file defined by the cookie.

A typical incremental backup or index-update program can read to the end of the FCL file and perform actions based on the FCL records. The application can get information about the current position in the FCL file using `vxfs_fcl_getcookie` and then store the cookie in a persistent structure such as a file. The next time the application needs to perform an incremental operation, it reads the cookie and passes it to `vxfs_fcl_seek` to seek to the point where it left off. This enables the application to read only the new FCL records.

Parameters

`int vxfs_fcl_getcookie(`*`void *handle, struct fcl_cookie *cookie`*`)`

- *`*handle`* is the FCL file handle returned by a call to `vxf_fcl_open`
- *`*cookie`* is a pointer to an opaque data block defined as follows:

```
struct fcl_cookie {
    char    fc_bytes[24];
};
```

The data stored in the cookie is internal to the VxFS library. The application should not assume any internal representation for the cookie or tamper with the data in the cookie.

vxfs_fcl_seek

You can use `vxfs_fcl_seek` to seek to the start or end of an FCL file depending on the flag passed to it.

See "vxfs_fcl_getcookie" on page 31.

Parameters

`int vxfs_fcl_seek(`*`void *handle, struct fcl_cookie *cookie, int where`*`)`

- The **handle* should be the same handle that was returned by the most recent call to `vxfs_fcl_open`. This is not necessarily the same handle used in `vxfs_fcl_getcookie`. The application may open the FCL file, get the cookie, and close the FCL file in one session, and then open the FCL file and submit the saved cookie in a later session. For each open session on the FCL file, the valid handle is the one returned by `vxfs_fcl_open` for that session.
- The **cookie* should point to a valid cookie that has returned from a call to `vxfs_fcl_getcookie:`
  - On the same FCL file
  - One of its checkpoints
  - One of the dumped or restored copies of the same FCL file.

  It is the responsibility of the user application to decide which FCL file is valid for a particular cookie and to use them in a sensible combination.

  Note: **cookie* may be NULL if *where* is `FCL_SEEK_SET` or `FCL_SEEK_END`.

- *where* should be one of `FCL_SEEK_SET`, `FCL_SEEK_END`, `FCL_SEEK_COOOKIE`.
  - If *where* is `FCL_SEEK_SET` or `FCL_SEEK_END`, the **cookie* argument is ignored and `vxfs_fcl_seek` seeks to either the start or end of the FCL file respectively, that is, where the first FCL record starts or where the last record ends
  - If *where* is `FCL_SEEK_COOKIE`, `vxfs_fcl_seek` extracts the activation time and offset stored in the **cookie*

  If the FCL has been de-activated (switched off) from the time the application last did a `vxfs_fcl_getcookie` function, or if the record at the offset contained in the **cookie* was purged during by a hole-punch, `vxfs_fcl_seek` returns an `FCL_EMISSEDRECORD` error. If not, `vxfs_fcl_seek` sets the current file position to the offset contained in the cookie. Further calls to `vxfs_fcl_read` return records from this offset.

Return values

A "0" indicates success; a non-zero indicates an error.

Note: `vxfs_fcl_seek` returns `FCL_EMISSEDRECORD` if the FCL has been reactivated, i.e., the activation time in FCL is different than that passed in the cookie, or the first valid offset in the FCL file is greater than the offset present in the cookie.

---

vxfs_fcl_seektime

The `vxfs_fcl_seektime` function seeks to the first record in the FCL file that has a time stamp greater than or equal to the specified time.

Parameters

`int vxfs_fcl_seektime(`*`void *handle, struct fcl_timeval time`*`)`

- *`*handle`* is a valid handle returned by a previous call to `vxfs_fcl_open`
- *`time`* is an `fcl_time_t` structure type defined as follows:

```
struct fcl_time {
    uint32_t tv sec;
    unit32_t tv_nsec;
} fcl_time t;
```

Note: The time specified in `fcl_time_t` may be in seconds or nanoseconds, while the time that is returned by a standard system call such as `gettimeofday` may be in seconds or microseconds. Therefore, a conversion may be needed.

`vxfs_fcl_seektime` assumes that the entries in the FCL are in a non-decreasing order of the time stamps and does a faster-than-linear (binary) search to determine the FCL record with a time stamp greater than the specified time. This means that `vxfs_fcl_seektime` can seek to a different record when compared to a seek done through a linear search.

As a result, the `vxfs_fcl_seektime` interface is not 100% reliable. Under the following circumstances, the time stamps in the FCL might be out-of-order:

- If the system time is modified

- If the FCL file is on a cluster-mounted file system and the times on the different nodes are out-of-sync

---

Warning: On a cluster file system, you must use a mechanism to keep the system clocks in sync (for example, Network Time Protocol—NTP), to help ensure that the `vxfs_fcl_seektime` interface is kept reasonably accurate.

Return values

`vxfs_fcl_seektime` returns "0" on success. If there are no records in the FCL file newer than the time indicated in the *`time`* parameter, `vxfs_fcl_seektime` returns `EINVAL`.

vxfs_fcl_sync

The `vxfs_fcl_sync` function sets a synchronization point within the FCL file. This function is kept for backward compatibility.

Before the availability of the VxFS 5.0 API to access the FCL file, applications would typically call `vxfs_fcl_sync` to get the FCL to a stable state and set an offset in the FCL file to use as a reference point to stop reading. The application would then store the offset and use it to determine files changes since the last FCL read time. A `vxfs_fcl_sync` call would then that if a file had been written to or opened, there would be at least one corresponding write or open record in the FCL after the synchronization offset. This would happen even if the time specified by `fcl_winterval` or `fcl_ointerval` had not elapsed since the last record was written.

With the VxFS 5.0 API FCL access, synchronization is now done automatically when the FCL file is opened through `vxfs_fcl_open`. The `vxfs_fcl_open` function sets a synchronization point and determines a reference end offset internally.

Parameters

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

- *`*fname`* is a pointer to the FCL filename
- *`*offp`* is the address of a 64-bit offset

`vxfs_fcl_sync` brings the FCL file to a stable state and updates *`*offp`* with an offset that can be used by the application as a reference point.

FCL record

An application reads the FCL file through the `vxfs_fcl_read` function. `vxfs_fcl_read` performs the following tasks:

- Reads the data from the FCL file
- Assembles the data into fcl_record structures
- Fills the buffer passed in by the application with these records

Each `fcl_record` structure represents a logical event recorded in the FCL. It is defined as the following:

```
struct fcl_record {
  uint32_t   fr_reclen;            /* Record length */
  uint16_t   fr_op;                /* Operation type. */
  uint16_t  fr_unused1;            /* unused field */
  uint32_t   fr_acsinfovalid : 1;    /* fr_acsinfo field valid */
  uint32_t  fr_newnmvalid : 1; /* fr_newfilename field is valid */
  uint32_t   fr_pinogenvalid : 1; /* fr_fr_pinogen field is valid */
  uint32_t  fr_unused2 : 29;        /* Future use */
  uint64_t  fr_inonum;             /* Inode Number. */
  uint32_t  fr_inogen;             /* Inode Generation Count. */
  fcl_time_t fr_time;              /* Time. */
  union fcl_vardata {
    char                    *fv_cmdname;
    struct fcl_nminfo        fv_nm;
    struct fcl_iostats      *fv_stats;
    struct fcl_evmaskinfo    fv_evmask;
} fr_var;
  uint64_t              fr_tdino;              /* Target dir ino */
  char                   *fr_newfilename;        /* For rename */
  struct fcl_acsinfo      *fr_acsinfo;            /* Access Info */
};

struct fcl_nminfo {
  uint64_tfn_pinonum;/* Parent Inode Number. */
  uint32_tfn_pinogen;/* Parent Inode Gen cnt. */
  char*fn_filename;
};

struct fcl_evmaskinfo {
  uint64_toldmask;/* Old event mask. */
  uint64_tnewmask;/* New event mask. */
};
```

Defines

These defines are provided for easier access:

```
#define fr_cmdname     fr_var.fv_cmdname
#define fr_stats       fr_var.fv_stats
#define fr_oldmask     fr_var.fv_evmask.oldmask
#define fr_newmask     fr_var.fv_evmask.newmask
#define fr_pinonum     fr_var.fv_nm.fn_pinonum
#define fr_pinogen     fr_var.fv_nm.fn_pinogen
#define fr_filename    fr_var.fv_nm.fn_filename
```

fcl_iostats structure

VxFS 5.0 lets you gather statistics such as the number of reads /writes occurring on a file. You can enable this through the `fiostat` command. The gathered stats are maintained in a per-file in-core structure and the File Change Log acts as a persistent backing store for the statistics. The stats are written to the FCL under the following circumstances:

- When the in-core structures need to be freed
- When the stats are reset
- At periodic intervals

These statistics can be read from the FCL as `VX_FCL_FILESTAT` records. Each record contains information as defined by the following `fcl_iostat` structure:

```
struct fcl_iostats {
    uint64_t  nbytesread; /* Number of bytes read from the file*/
    uint64_t  nbyteswrite;/* Number of bytes written to the file*/
    uint32_t  nreads;     /* Number of reads from the file */
    uint32_t  nwrites;    /* Number of writes to the file */
    uint32_t  readtime;   /* Total time in seconds for the reads */
    uint32_t  writetime;  /* Total time in seconds for the writes */
    struct {
        uint32_t   tv_sec;
        uint32_t   tv_nsec;
    } lastreset;/*      Last reset time for the stats */
    uint32_tnodeid;      /* Node from which the record was written */
    uint32_treset;       /* Stats have been written due to a reset */
};
```

Each iostat record in the FCL contains I/O statistics accumulated over the time interval from the *`lastreset`* time to when the FCL record is written. Over a period of time, the cumulative statistics and aggregate can be computed by the following:

- Traversing the FCL
- Looking for records of type `VX_FCL_FILESTATS`

For example, computing the aggregate for the total number of reads over a period of time requires traversing a set of FCL files to obtain I/O statistics records. This informations contains a sequence of records of the type `VX_FCL_FILESTATS` with the same *`lastreset`* time followed by another sequence of records with a later *`lastreset`* time for a specific file.

The aggregation considers values only from the latest record from records with the same *`lastreset`* time and then sums up the number of reads for each such record.

fcl_acsinfo structure

When tracking access-info is enabled, VxFS logs the access information such as:

- The real and effective user and group ID of the accessing application
- The node from where the file was accessed
- The process id of the user application along with each record

When the application reads the FCL, the information is returned in the `fr_acsinfo` field. The `fr_acsinfo` points to an FCL`_acsinfo` structure, defined as follows:

```
struct fcl_acsinfo {
    uint32_tfa_ruid;
    uint32_tfa_rgid;
    uint32_tfa_euid;
    uint32_tfa_egid;
    uint32_tfa_pid;
    uint32_tfa_nodeid;
};
```

Note: The `accessinfo` is not returned as a separate record type but as additional information along with the other records. In addition, the `accessinfo` information is not always present with every record (if `accessinfo` is not enabled). However, even when `accessinfo` is enabled in some file system internal operations, (such as truncating a file), the access information may not be present. To help determine if access information is available, the FCL record contains a flag called `fcl_acsinfovalid` which is non-zero only if the `accessinfo` is present with a particular record.

Several of the fields in the `fcl_acsinfo` structure are pointers and need memory to store the actual contents. This is handled by storing the actual data immediately after the FCL record, and updating the pointer to point to the data. The record length *`fr_reclen`* field is updated to account for the whole data. Thus, each FCL record returned by `vxfs_fcl_read` is a variable size record, whose length is indicated by *`fr_reclen_field`*. Figure 2-3 on page 38 shows how the data is laid out in a sample link record.

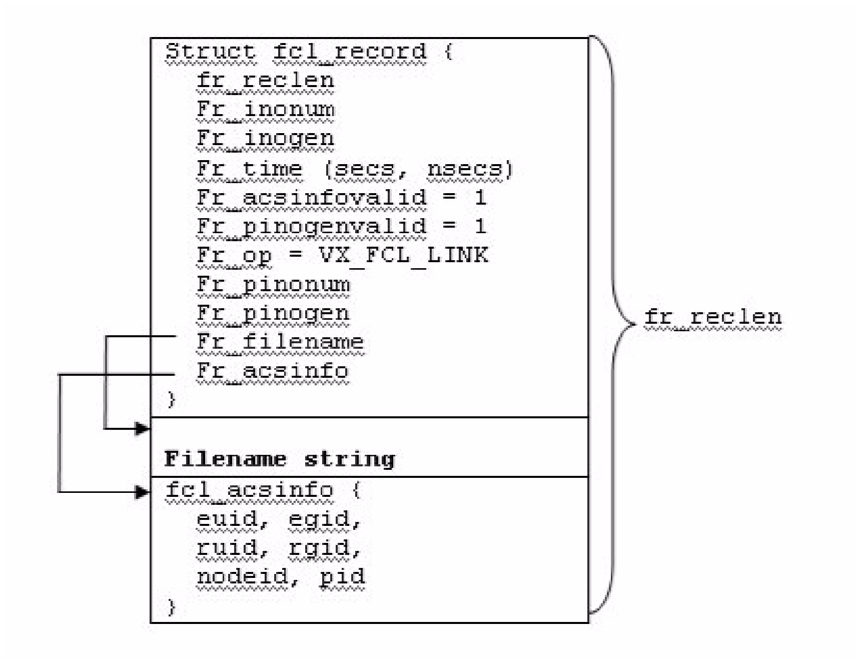


Figure 2-3 Sample link record

The following code sample traverses the set of records returned by a call to `vxfs_fcl_read` and prints the user ID:

```
Struct fcl_record*fr;
Char       *tbuf;
…
        error = vxfs_fcl_read(fh, buf, &bufsz,
                         FCL_ALL_V4_EVENTS,
                         &nentries);
    tbuf = buf;
    while (--nentries) {
        fr = (struct fcl_record *)tbuf;
if (fr->fr_acsinfovalid) {
        printf("Uid %ld\n", fr->fr_acsinfo->uid;
}
        tbuf += fr->fr_reclen;
    }
```

Note: FCL_ALL_V4_EVENTS are event masks.

See "vxfs_fcl_read" on page 29.

Record structure fields

Table 2-2 briefly describes each field of the `fcl_record` structure and indicates the record types for which it is valid.

Table 2-2 FCL record structure fields

Field	Description	Validity
`fr_reclen`	Length of the FCL record. This includes length of the FCL record structure and length of the data stored immediately following the structure. This length should be used while traversing FCL records returned in the buffer by `vxfs_fcl_read`.	Valid for all records.
`fr_inonum`	The inode number of the file being changed. To generate the full path name of the changed object, the inode number and generation count (`fr_inogen`) can be used with `vxfs_inotopath_gen`.	Valid for all FCL records except when the record is FCL_EVNTMSK_CHG. For event mask change the file is implicitly the FCL file.
`fr_op`	The operation for this FCL record, for example, creation, unlink, write, file attributes change, or other change. `fr_op` takes on one of the values for the record types listed in Table 2-1. Use this parameter to determine which fields of the FCL record are valid.	Valid for all records.
`fr_time`	The approximate time when the change was recorded in the FCL file. Use the `ctime` call to interpret this field.	Valid for all records.
`fr_inogen`	The generation count of the changed file. The generation count in combination with the inode number (of the file) is passed to `vxfs_inotopath_gen` to provide the exact full path name of the object. Without the generation count, the returned path name can be a re-used inode.	Valid for all FCL records except for event mask changes and unlinks. For event mask changes, the inode number and generation count are implicit. For unlink, the generation count is not needed to get the filename via reverse name lookup, since it is already present with the record.

Table 2-2 FCL record structure fields

Field	Description	Validity
fr_pinonum fr_pinogen fr_filename	For FCL records like file remove or rename, where the directory entry is removed, the filename cannot be determined by reverse name lookup. Similarly in the case of link record, the filename cannot be determined unambiguously. Therefore in these cases, the filename, inode number, and generation count of the parent directory (containing the file being changed) is recorded. The parent directory inode (fr_pinonum) and generation count (fr_pinogen) can be used with the reverse name lookup API to identify the full path name of the parent directory. Adding the trailing filename yields the object's full name.	Valid when the FCL record is VX_FCL_UNLINK, VX_FCL_RENAME or VX_FCL_LINK. The unlink and rename; filename and the parent inode number; and generation count, contain information about the old file that was removed. For the link, they represent the new filename.
fr_cmdname	A short name of the command which opened the file represented by fr_inonum and fr_inogen.	Valid only when the FCL record is VX_FCL_FILEOPEN.
fr_stats	A pointer to an FCL_iostat record. The fcl_iostat record contains I/O statistics such as the number of reads/ writes that happened on the file, average time for a read/ write, etc. These point-in-time records can be used to compute the aggregate or average I/O statistics for a file over a period of time.	Valid only when the FCL record is VX_FCL_FILESTATS.
fr_oldmask fr_newmask	These fields contain the old and new event masks, respectively. Each event mask is a "logical or" of a set of masks defined in fcl.h.	Valid only when the FCL record is VX_FCL_EVNTMASK_CHG.

Table 2-2 FCL record structure fields

Field	Description	Validity
`fr_acsinfo`	A pointer to an FCL_`acsinfo` structure. This structure contains information such as the user and group ID of the application that performed the particular operation, the process id and the ID of the accessing node.	Validity is determined by the `fcl_acsinfovalid` `bit-field`. It can potentially exist with all kinds of records. This is an optional field.

Copying FCL records

Each FCL record returned by `vxfs_fcl_read` is of variable size and consists of the `fcl_record` structure, followed by the additional data associated with the record. The pointers in the `fcl_record` structure point to the data stored after the fcl_record structure and the record length specifies the size of the variable sized record. However, making an in-core copy of the FCL record involves more than replicating `fr_reclen` bytes of data from the source to the copy.

A simple memory copy just copies over the pointers from the source record to the target record. This leaves the pointers in the target record pointing to data from the source. Eventually, this can cause problems when the memory for the source record is re-used or freed. The pointers in the replica must be modified to point to data in the target record. Therefore, to make an in-core copy of the FCL record, the application must use the `vxfs_fcl_copyrec` function to copy and perform the pointer relocation. The user application must allocate the memory needed for the copy.

Index maintenance application

This sample application is for a system that maintains an index of all files in the file system to enable a fast search similar to the `locate` program in Linux. The system needs to update the index periodically, or as required with respect to the file changes since the last index update. The following lists the basic steps to perform and shows a sample call to the FCL API.

Preparation

To prepare the application

1. Enable the FCL.

   ```
   $ fcladm on mntpt
   ```

2. Tune *`fcl_keeptime`* and *`fcl_maxalloc`* to the required values.

   ```
   $ vxtunefs -o fcl_keeptime=value
   ```

```
$ vxtunefs -o fcl_maxalloc=value
```

First run of the index maintenance application

To test est the application

1. Open the FCL file.
   ```
   $ vxfs_fcl_open(mntpt, 0, &fh);
   ```
2. Seek to the end.
   ```
   $ vxfs_fcl_seek(fh, NULL, FCL_SEEK_END);
   ```
3. Get the cookie and store it in a file.
   ```
   $ vxfs_fcl_getcookie(fh, &cookie)
   write(fd, cookie, sizeof(struct fcl_cookie));
   ```
4. Create the index.

Periodic run to update the index

To update the application

1. Open the FCL file.
   ```
   $ vxfs_fcl_open(mntpt, 0, &fh);
   ```
2. Read the cookie and seek to the cookie.
   ```
   $ read(fd, &cookie, sizeof(struct fcl_cookie))
   $ vxfs_fcl_seek(fh, cookie, FCL_SEEK_COOKIE)
   ```
3. Read the FCL file and update the index accordingly.
   ```
   $ vxfs_fcl_read(fh, buf, BUFSZ, FCL_ALL_v4_EVENTS, &nentries)
   ```
4. Get the cookie and store it back in the file.
   ```
   $ vxfs_fcl_getcookie(fh, &cookie)
   $ write(fd, cookie, sizeof(struct fcl_cookie));
   ```

Computing a usage profile

This sample application computes the usage profile of a particular file, that is, the users who have accessed a particular file in the last hour.

Initial setup

This sample application needs additional information such as tracking file opens and access information, that are available only with FCL Version 4. Be sure to enable the correct FCL version.

The following steps perform the required initial setup.

To setup up the application

1. Switch on the FCL with Version 4.
   ```
   $ fcladm -o version=4 on mntpt
   ```

Note: If this step fails, use `fcladm print` to check for an existing FCL Version 3 file. If present, remove it with `fcladm rm` and then try switching on FCL with Version 4.

In VxFS 5.0, the default FCL version is 4. If there is no existing FCL file, the `fcladm on` *`mntpt`* command automatically creates a Version 4 FCL.

---

2. Enable tracking of access information, file-opens, and I/O statistics as needed.

   ```
   $ fcladm set fileopen,accessinfo mntpt
   ```

3. Set tunables `fcl_keeptime`, `fcl_maxalloc`, and `fcl_ointerval` as required. For example:

   ```
   $ vxtunefs fcl_ointerval=value
   ```

Sample steps

The following provides sample steps for possible application use.

1. Open the FCL file.

   ```
   vxfs_fcl_open(mntpt, 0, &fh);
   ```

2. Set up the time to perform the seek.

   a. Get current time using `gettimeofday`.

   b. Fabricate the `fcl_time_t` for the time an hour before.

   c. Seek to the record in the FCL file at that time.

   ```
   gettimeofday(&tm, NULL);
   tm.sec -= 3600
   vxfs_fcl_seektime(fh, tm);
   ```

3. Read the file with appropriate event masks until the end of file (The application is interested in only the file open records and the access information).

   a. Check if the file inode number and generation count are same as the ones being sought for each FCL record.

   b. Print information about the user who has accessed the file, if applicable.

   ```
   vxfs_fcl_read(fh, buf, BUFSZ, VX_FCL_FILEOPEN_MASK | \
   VX_FCL_ACCESSINFO_MASK, &nentries);
   ```

Off host processing

In some scenarios, a user application may choose to save the bandwidth of the actual production server and outsource the job of processing the FCL to a different system. For off-host processing, the FCL file needs to be shipped to the off-host system. Since the FCL file is not a regular file, a command such as `cp` or `ftp` does not work.

To be" shippable," the FCL file must first be dumped into a regular file using the `fcladm dump` command. The file can then be sent to the off-host system using normal file transfer programs. See the following example.

```
# fcladm -s savefile dump mntpt
# rcp savefile offhost-path
```

On the off-host system, the FCL file must be then restored using the *`restore`* option through the `fcladm` command. Unlike the original FCL file, the restored file is a regular file.

```
# fcladm -s savefile restore restorefile
```

The restored FCL file can be passed as an argument to `vxfs_fcl_open` for further use with the FCL API.

Warning: The reverse name lookup API does not work on the off-host system. The off-host processing mechanism should only be used when the application can work with the inode number and generation count, or when it has an independent method to determine the filenames from the inode number.

VxFS and FCL upgrade and downgrade

VxFS 4.1 supported only FCL Version 3. VxFS 5.0 supports both FCL Version 3 and 4, with Version 4 as the default. When a system is upgraded from VxFS 4.1 to VxFS 5.0, and the file system has FCL switched on, the existing Version 3 FCL files remains as is. VxFS 5.0 continues tracking file system changes in the Version 3 FCL exactly as it was done by VxFS 4.1.

A VxFS 4.1 application that directly accesses the FCL file using the `read`(2) system call can still continue to work in 5.0 provided that the FCL file is Version 3. However, you must develop any new applications using the API. The API has support for both FCL Versions 3 and 4.

If a new application uses the newly added record types in VxFS 5.0 such as file opens or access information, etc., the FCL needs to be at Version 4.

If you are running applications that still read FCL Version 3 directly, you cannot upgrade to FCL Version 4 until those applications are rewritten to use the new API. The API can interpret both Version 3 and Version 4, so applications can be upgraded to use the API while Version 3 is still in effect.

Converting FCL Version 3 files to Version 4

The following provides the path for moving from FCL Version 3 to 4:

1. Switch off the FCL.
   ```
   $ fcladm off mntpt
   ```
2. Remove the existing FCL file.
   ```
   $ fcladm rm mntpt
   ```
3. Re-activate with the required version.
   ```
   $ fcladm [-oversion=4] on mntpt
   ```

Downgrading VxFS versions

During data center operations, a VxFS file system might need to be migrated from a host running a newer version of VxFS to a host running an older version (for example, from VxFS 5.0 to VxFS 4.0). Such a migration might occur for offhost processing. If the FCL file created under VxFS 5.0 is FCL Version 3, it can continue to be used (as is) under VxFS 4.0. However, if the FCL file is Version 4, then it must be removed using `fcladm rm` and reactivated as FCL Version 3 before it can be used under VxFS 4.0.

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` 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:

- Clients of the VxFS file change log feature
- Backup and restore utilities
- Replication products

Typically, these applications store information by inode numbers because a path name for a file or directory can be very long and need an easy method to obtain a path name.

Inodes

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 filenames 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 filename from an inode identifier by scanning each directory in the file system, but this process can take a long 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 filename, when combined with the use of reverse path name lookup, can generate full path names for each FCL record.

vxfs_inotopath_gen

The vxfs_inotopath_gen function 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 function whenever possible.

The vxfs_inotopath function is included only for backward compatibility. The vxfs_inotopath function does not take the inode generation count.

The following is the syntax for vxfs_inotopath and vxfs_inotopath_gen:

```
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,
                      unint32_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, along with the array itself, 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 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 7 disk layouts.

Chapter 3

Multi-volume support

This chapter includes the following topics:

About multi-volume support

The multi-volume support (MVS) feature lets a VxFS file system 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.

The following levels of policies 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 is used.

The MVS APIs fall into the following 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.

See the `fsvoladm`(1M) and `fsapadm`(1M) manual pages.

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 application programmatic interfaces

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*.

Administering volume sets

The following examples show how to administer volume sets.

To convert a volume to a volume set

◆ To convert `myvol1` to a volume set, use the following function call:

```
# vxvset make myvset myvol1
```

To add a volume to a volume set

◆ To add `myvol2` to the volume set `myvset`, use the following function call:

```
# vxvset addvol myvset myvol2
```

To list volumes of a volume set

◆ To list the volumes of `myvset`, use the following function call:

```
# vxvset list myvset
```

To remove a volume from a volume set

- To remove `myvol2` from `myvset`, use the following function call:

  ```
  # vxvset rmvol myvset myvol2
  ```

Querying the volume set for a file system

The following function calls query a volume set for a file system.

To query all volumes associated with the file system

- To query all volumes associated with the file system, use the following function call:

  ```
  vxfs_vol_enumerate(fd, &count, infop);
  ```

To query a single volume

- To query a single volume, use the following function call:

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

Modifying a volume within a file system

The following function calls modify a volume within a file system.

To grow or shrink a volume

- To grow or shrink a volume, use the following function call:

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

To remove a volume from a file system

- To remove a volume from a file system, use the following function call:

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

Add a volume to a file system

- To add a volume to a file system, use the following function call:

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

Encapsulating and de-encapsulating a volume

The following function calls encapsulate a volume.

To encapsulate a raw volume

- To encapsulate an existing raw volume and make the volume contents appear as a file in the file system, use the following function call:

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

To de-encapsulate a raw volume

- To de-encapsulate an existing raw volume to remove the file from the file system, use the following function call:

```
vxfs_vol_deencapsulate(encapsulate_name);
```

Allocation policy application programmatic interfaces

To make full use of multi-volume support features, VxFS supports 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.

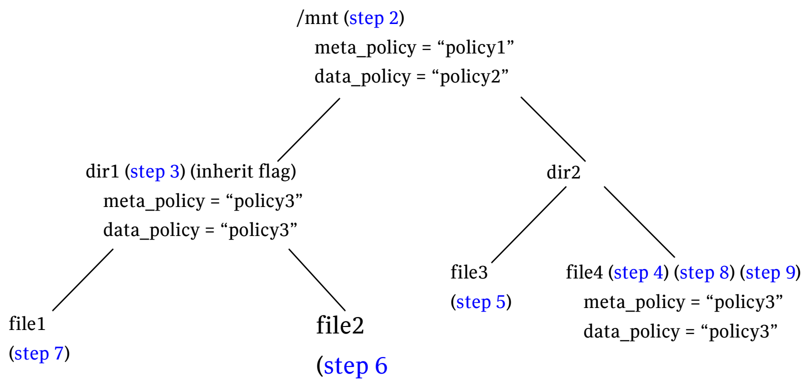
The allocation policies are defined per file system and are persistent. There is no fixed limit on the number of allocation policy definitions in a file system. Once a policy is assigned, new file allocations are governed by the policy. For files allocated before a policy was defined or assigned or when 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.

Directing file allocations

Figure 3-4 shows how you might use the allocation policies to direct file allocations.

Figure 3-4 Directing File Allocations



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.

To direct file allocations

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`.

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 are allocated on `vol-02` as directed by the `policy1` and `policy2` assignments. When `file1` and `file2` are created, they are 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 example creates and assigns a policy using the multi-volume API.

To create and assign a policy

1. To define a policy for a file system, use the following function call:
   ```
   vxfs_ap_define(fd, fsap_info_ptr, 0);
   ```
2. To assign a policy to a file system, use the following function call:
   ```
   vxfs_ap_assign_fs(fd, data_policy, meta_policy);
   ```
3. To assign a policy to a file or directory, use the following function call:
   ```
   vxfs_ap_assign_file(fd, data_policy, meta_policy, 0);
   ```
4. To assign a policy to a Storage Checkpoint, use the following function call:
   ```
   vxfs_ap_assign_ckpt( fd, checkpoint_name, data_policy,
       meta_policy);
   ```

Querying the defined policies

The following function calls query defined policies.

To query all policies on a file system

- To query all policies on a file system, use the following function call:
  ```
  vxfs_ap_enumerate(fd, &count, fsap_info_ptr);
  ```

To query a single defined policy

- To query a single defined policy, use the following function call:
  ```
  vxfs_ap_query(fs, fsap_info_ptr);
  ```

To query a file for its assigned policies

- To query a file for its assigned policies, use the following function call:
  ```
  vxfs_ap_query_file(fs, data_policy, meta_policy, 0);
  ```

To query a Storage Checkpoint for its assigned policies

- To query a Storage Checkpoint for its assigned policies, use the following function call:
  ```
  vxfs_ap_query_ckpt(fd, check_point_name, data_policy,
                                       meta_policy)
  ```

Enforcing a policy on a file

The following function call enforces a policy.

To enforce a policy on a file

- To enforce a policy on a file, use the following function call:
  ```
  vxfs_ap_enforce_file(fd, data_policy, meta_policy);
  ```
  Enforcing the policy may cause the file to be reallocated to another volume.

Data structures

You can view the fsap_info and fsdev_info data structures in the vxfsutil.h header file and libvxfsutil.a library file.

See the vxfsutil.h header file and libvxfsutil.a library file.

The data structures are provided here for quick reference:

```
#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_ASGIVEN |
                                 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 */
};
```

Using policies and application programmatic interfaces

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.

To define and assign an allocation policy to reallocate an existing file's data blocks to a specific volume

◆ To reallocate an existing file's data blocks to a specific volume (`vol-03`), create code similar to the following:

```
/* 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_ASGIVEN;
ap.ap_ndevs = 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);
```

To create policies that allocate new files under a directory

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

◆ To create policies to allocate new files under directory `dir1`, create code similar to the following:

```
/* 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 application programmatic interfaces

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

To shrink or grow a volume within a file system

1. Use the `vxresize` command to grow the physical volume.
2. To use the `vxfs_vol_resize`() call to shrink or grow the file system, create codes similar to the following:

```
/* 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);
```

To encapsulate a raw volume as a file

1. Add the volume to the volume set.
2. To encapsulate a raw volume `vol-03` as a file named `encapsulate_name` in the file system `/mnt`, create code similar to the following:

```
/* 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);
```

To de-encapsulate a raw volume

◆ To de-encapsulate the raw volume `vol-03` named `encapsulate_name` in the file system `/mnt`, create code similar to the following:

```
/* Use de-ecapsulate 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");
```

Chapter 4

Named data streams

This chapter includes the following topics:

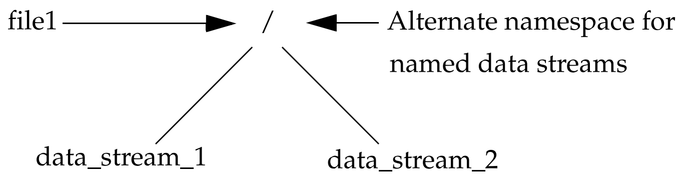
About named data streams

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.

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

Figure 4-5 illustrates the alternate namespace associated with a file.

Figure 4-5 Alternate Namespace



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.

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:

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

VxFS named data stream functionality is available through the following application programming interface functions:

`vxfs_nattr_open()`

Works similarly 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 following is the syntax for the `vxfs_nattr_open()` API:

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

`vxfs_nattr_link()`	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 calling function must have write permission to link the named data stream. The following is the syntax for the `vxfs_nattr_link()` API: `int vxfs_nattr_link(int sfd, char *spath, char *tpath);`
`vxfs_nattr_unlink()`	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 following is the syntax for the `vxfs_nattr_unlink()` API: `int vxfs_nattr_unlink(int fd, char *path);`
`vxfs_nattr_rename()`	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 following is the syntax for the `vxfs_nattr_rename()` API: `int vxfs_nattr_rename(int sfd, char *old, char *tnew);`
`vxfs_nattr_utimes()`	Sets the access and modification times of the named data stream. The following is the syntax for the `vxfs_nattr_utimes()` API: `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_utimes`(3) manual pages.

Listing named data streams

The named data streams for a file can be listed by calling `getdents()` on the named data stream directory inode, as shown in the following example.

To list named data streams

1. To list the named data streams, create code similar to the following:

```
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 */
```

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

```
/mount_point/file_with_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

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 fail. `VOP_SETATTR()` called on a named data stream directory or named data stream inode also fails.

Querying named data streams

In the following example, a file `named_stream_file` was created with 20 named data streams using the API calls.

The named data streams are not displayed by the `ls` command. When named data streams are created, they are organized in a hidden directory. For example:

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

To query named data streams

- Use the `getdents()` or `readdir_r()` system call to 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

0x1ff 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
```

Application programmatic interface

The named data streams API uses a combination of standard system calls and VxFS API calls to utilize its functionality.

The following is an example of pseudo code to query named data streams:

```
/* 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);
}
```

Command reference

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

VxFS I/O Application Interface

Topics in this chapter include:

- Freeze and thaw
- Caching advisories
- Extents

Note: 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.

Freeze and thaw

Freezing a file system temporarily blocks all I/O operations to a file system and then performs a sync on the file system. Current operations are completed and the file system is synchronized to disk. Freezing a file system is a necessary step for obtaining a stable and consistent image of the file system at the volume level.

Consistent volume-level 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 multiple 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.

The following table shows freeze/thaw compatibility with VxFS releases:

	VxFS 3.5	VxFS 4.0	VxFS 4.1	5.0
VX_FREEZE	Local File System	Local File System Cluster File System	Local File System Cluster File System	Local File System Cluster File System
VX_FREEZE_ALL	Local File System	Local File System	Local File System Cluster 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.

Use the VX_THAW ioctl to thaw file systems frozen with VX_FREEZE_ALL ioctl before the timeout period has expired.

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 optimal 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.

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 direct 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 metadata 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 generally 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 such, concurrent I/O has the same alignment requirements as direct I/O.

See "Direct I/O" on page 73.

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-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.

Concurrent I/O (CIO) 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.

CIO 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 same 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 are 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_GETEXT` 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` iotcl 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_GETEXT` 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, and the preallocation size cannot be increased beyond the `ulimit` of the requesting process.

See the `VX_CHGSIZE` flag.

See the `ulimit`(2) manual page.

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, VX_CONTIGUOUS, and VX_GROWFILE 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_GETEXT 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_CHGSIZE`. 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.

Reading the grown part of the file

When the allocation flag (`a.flag`) is set to `VX_GROWFILE`, the size of the file is changed to include the reservation. This flag reads the grown part of the file (between the current size of the file and the size after the operation succeeds). `VX_GROWFILE` has persistent affects, but is not visible as an allocation flag. This flag is visible through the `VX_GETEXT` ioctl.

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_GETEXT` 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.

To use `VX_SETEXT`

1. Call the `VX_GETEXT` 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.

Warning: 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 a different 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_PREFFERED_EXTSIZE is multiple of the file system block size:

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

struct vx_ext myext;

fd = open(MY_FILE, O_CREAT, 0644);

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

error = ioctl(fd, VX_SETEXT, &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 myext;

fd = open(MY_FILE, O_CREAT, 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_SETEXT, &myext);
```

Index