FUJITSU COMPUTERS SIEMENS

Reliant UNIX ONLINE Documentation

Reliant UNIX 5.45

VERITAS File System (VxFS) V2.1

RM200, RM300, RM400, RM600

Edition March 1999

Identification: U42307-J-Z915-1-76

Copyright and Trademarks

1 Preface

File system types supported by Reliant UNIX ® at present are *VxFs*, *ufs*, *nfs* and *hs*. This manual describes the features of the VERITAS ™ file system and provides instructions for the organization and administration of the file system for the system administrator.

1.1 Audience

The audience for this manual is system administrators responsible for installing, configuring, and maintaining the VERITAS File System (VxFS) on Reliant UNIX ® systems. The manual also provides information for programmers. The VERITAS File System is transparent to users so they do not need documentation on it.

1.2 Scope of this guide

This manual describes the VERITAS File System which was developed by the Veritas™ Software Corporation.

With Reliant UNIX ® 5.45 the VERITAS File System Version 2.1 is available. This revised version of the manual includes the description of the new features as far as they are available with Reliant UNIX ® 5.45.

The manual contains the following chapters:

Chapter 2: The VxFS – a survey

This chapter contains a discussion of VxFS features, which includes:

- conversion of file systems to VxFS
- fast file system recovery
- on-line file system administration
  (fragmentation limitation and file system resizing)
- on-line backup
- extended application interface
  (expanded application facilities, support of large files and file systems, support of long file names)
- extended *mount* options
  (enhanced data integrity modes, temporary file system mode)
- enhanced file system performance
- extent-based allocation (with adjacent data blocks on the hard disk)

Chapter 3: Verifying installation

The chapter discusses the installation verification of kernel and utility components.

Chapter 4: The VxFS disk structure

This chapter discusses the VxFS disk structure, which includes:

- the superblock
- the VxFS intent log
- allocation units

Chapter 5: On-line backup

This chapter describes the online backup facility provided with VxFS.

Chapter 6: Performance and tuning

This chapter discusses ways to optimize and efficiently utilize the file system using the VxFS utilities.

Chapter 7: Application interface

This chapter details cache advisories and provides information about extent sizes and file allocation flags.

Chapter 8: Utilities and commands

This chapter describes the utilities and commands used with VxFS.

Chapter 9: Messages

This chapter contains detailed information about error messages, diagnostic messages and recommended user actions to rectify system problems when using the commands described in chapter 8.

Appendix: Table of error messages

The appendix contains an alphabetical listing of the error messages generated by the VxFS file system utilities and kernel. This list is included to lessen the task of locating error messages and their meanings.

1.3 Display conventions

The following fonts are used throughout this document

italic

The names of files, commands, utilities and options appear in text in *italic* font.

Options appear in italic font.

constant width

Screen output, as for example to be expected as a result of a command that has been executed, appears in constant width font.□
Literals that are typed exactly as they appear are also indicated with constant width font.

bold

When command syntax is being described, the fixed parts of a command line (command name, options) are indicated with **bold** font.□
Statements which are to be emphasized are printed with **bold** font.

!

Warning: The text calls attention to errors that may happen and offers hints to avoid these. □

i

Additional information, notes and tips□

1.4 List of relevant terms and acronyms

Below you will find a list of important terms or abbreviations concerned with the VxFS file system.

Allocation

Allocation of a block in the file system. "storage" allocated here is taken to be storage space on the hard disk.

allocation unit

self-contained unit in the VxFS file system for administering the Inodes and blocks; the concept corresponds to "cylinder group" in the *ufs* file system

Application interface

Interface between application program and system kernel for programming file access is implemented using system calls (z.□B. *open*(2), *read*(2), *write*(2), *ioctl*(2), *close*(2))

Bitmap

Sequence of bits; each bit represents the status (free/occupied) of a resource (e.□g. **Block** or **Inode**)

Block

Basic unit in which storage space is assigned on the hard disk.

Block based allocation

Storage allocation model of the *ufs* file system. The files are assigned blocks of constant length (e.g. 16Kbyte). For physical I/O this block size also. To keep the "offset" for small blocks to the minimum the blocks are administered in smaller logical fragments (e.g. 2 Kbyte).

Block device

Buffered access to a device from the system standpoint (e.g. a hard disk partition)

Block size

integer multiple of the **sector size** (=512 bytes); VxFS supports block sizes of 1, 2, 4 und 8 Kbytes.

Character device

Unbuffered access to a device from the system standpoint (e.g. a hard disk partition)

Contiguous file

File which comprises a single extent and for which the data blocks are therefore contiguous.

Direct extent

the **extents** of a file which are entered directly in **Inode**. The first 10 extents of a file are direct extents. All further extents are administered in the indirect levels of the inode, i. e. they are entered into separate indirect **blocks** which leads to a higher administrative load (incl. additional I/O). For high performance all data of a file should be administered in direct extents.

Extent

formed by one or more **blocks** of the file system for which there are consectutive block numbers; an extent is characterized by the block number of the first block and the **extent size** in number of subsequent blocks.

extent based allocation

Storage allocation model of the VxFS file system. The files are assigned **blocks** of constant length (e.g. 1 Kbyte). When this is done, an attempt is made to group these blocks into **extents** of variable length. For physical I/O this provides larger units so that with large extents the bandwidth of the I/O controller can be fully utilized (for Reliant UNIX max. 64 Kbytes per I/O). This offers a performance advantage over *ufs* file systems.

Extent map

Sequence of **bitmaps**, which administer the reservation of the blocks within an allocation area.

Extent size

Blocks which form an **extent**.

File offset

read Position in a file from which the data is read or written; is implicitly incremented with (2) or *write*(2) by the length of the data transfer and can be explicitly modified with *lseek*(2).

Fragmentation

Measure for the "organization" of a file system as regards the block reservation and thereby for the option of being able to form large **extents** as well; as fragmentation increases the performance of a file system deteriorates. The fragmentation of a file system can be checked with *fsadm*(1M) and reduced if necessary with reorganisation.

Header

Data structure which gives an overview of the administration unit (e.g. allocation unit)

Indirect block, double

Block in the file system which contains the block numbers of single indirect blocks. The block number of the indirect block double is entered in the Inode of the file. The double indirect block is used to address data which can no (longer) be administered in the **single indirect block**.

Indirect block, single

Block in the file system which contains the block numbers of constant-length data. The block number of the single indirect block is entered in the Inode of the file. The single indirect block is used to address data blocks which can no (longer) be administered in the direct extents.

indirect extent
: the extents of a file which are being administered in indirect levels of the inode. They are entered into **single** and **double indirect blocks**.

Inode
: Information unit for a file in which all of the data of importance for the administration of the file are stored, e.g. file type, file size, access rights, **extents**, **indirect blocks** etc.. Each file is uniquely assigned via the inode number to an inode.

Inode map
: **Bitmap** for administration of the **inode** within an **allocation area**

intent log
: reserved memory area at the start of the file system on the hard disk in which all not yet completed changes to the file system structure are logged. After a system error, utility *fsck*(1M) can be used to very quickly re-establish the file system integrity using this intent log.

mount
: activate a file system

Preallocation
: allows allocation of **blocks** for a file before this has been created with *write*(2) system calls; Feature of the VxFS-file system

recovery
: After a system error *fsck*(1M) is used, together with **intent log** to very quickly re-establish the file system integrity without checking it completely. After a disk error however a full check of the file system is .

Sector
: Basic unit of a hard disk

Sector size
: 512 bytes

Snapshot file system
: Write-protected file system with a consistent map of the "suspended" status of the initial file systems, with which online data backup was enabled

Snapped file system
: a **mounted** file system for which a **snapshot file system** was created.

Super block
: Information unit for the file system. The super block contains the main information for the structure of the file system, e.g. status of the file system, file system size, block size, available resources etc.

throughput
: Data transferred per unit of time

Transaction
: Changes to the file system structure which are recorded in the **intent log** on the hard disk

ufs
: "unix file system", file system type in SVR4 which was developed from the "fast file system" (ffs) of Berkeley Software Distribution (BSD)

Unmount
: deactivate a file system

VxFS
: "veritas file system", file system type developed by VERITAS

2 The VxFS – a survey

The VERITAS file system (VxFS) is an extent-based file system for use on Reliant UNIX systems. VxFS provides enhancements that increase the usability of Reliant UNIX and makes Reliant UNIX systems more viable for use in the commercial marketplace. The main features of the VxFS file system are:

- fast file system recovery
- on-line administration
- on-line backup
- extended application interface □
  (expanded application facilities, support of large files and file systems, support of long file names)
- extended *mount* options □
  (enhanced data integrity modes, temporary file system mode)
- enhanced file system performance
- extent-based allocation

Limits

The VxFS file system supports all *ufs* file systems features and facilities except for the following:

- support of user specific quotas of the number of inodes and disk blocks
- support of hard linking, removing or renaming "." and ".." directory entries □
  (These operations are disallowed on "." and ".." to preserve filesystem sanity.) □
  Symbolic links on these directory entries are supported.

2.1 Conversion of file systems

Other file systems can be converted using the following steps:

(1)
Back up the contents of the file system to tape or some holding directory in the system.

(2)
Unmount the file system.

(3)
Create a VxFS file system on this partition or on another larger one. Optionally label it using the command *labelit*(1M).

(4)
Mount the newly created file system with type *vxfs*.

(5)
Restore the contents from the tape or some disk temporary holding directory.

(6)
If there is an entry for this file system in */etc/vfstab*, modify the type field to *vxfs*.

2.2 Fast file system recovery

The *ufs* file system uses the full structural verification of the *fsck* utility as the only means to recover from a system failure. The *fsck* utility for *ufs* checks the entire structure and corrects any inconsistencies that are found. For large disk configurations, this process can be very time consuming.

The VxFS file system provides recovery by utilizing a tracking feature called **intent log**. The intent log is written on a reserved area of the disk. All pending changes to the file system structure are recorded in the intent log. The recovery with *fsck* lasts only a few seconds when using the intent log.

Upon recovery from a system failure, the VxFS *fsck* utility scans the intent log, nullifying or completing file system operations that were active when the system failed. The file system may then be mounted without completing a structural check of the entire file system. Except for the fact that VxFS file system recovery is completed in a few seconds, the intent log recovery feature is not readily apparent to either the user or the system administrator.

VxFS V2.1 uses an intent log format that differs from the one of former file system versions. The intent log format will be adapted when a file system is being mounted on Reliant UNIX 5.45 and VxFS V2.1 for the first time.

Please never mount a file system on Reliant UNIX 5.44
and VxFS V1.2 after a crash, if it had been mounted on Reliant UNIX 5.45 and therefore already has the intent log format of VxFS V2.1. Even in this case the *fsck* command would try to deal with the unfinished modifications to the file system. But this might destroy the file system, because *fsck* can't read the intent log due to the old format.

A full structural mode of the *fsck* utility is still provided and must be run to recover from a disk hardware failure.

2.3 On-line file system administration

The VxFS file system includes the ability to perform defragmentation while the file system remains accessible to users, and file system resizing without another *mkfs*. The following sections contain detailed information about these features.

2.3.1 Limiting fragmentation

Free resources are originally aligned in the most efficient order possible and are allocated to files in a way that is considered by the system to provide optimal performance. When a file system is active for extended periods of time, files grow and shrink, are allocated and deleted. Over time, the original ordering of free resources is lost. As this process continues, the file system tends to spread further and further along the disk, leaving unused "gaps" or fragments between areas that are in use. This process is known as **fragmentation**. Fragmentation leads to degraded performance because the file system has fewer choices when selecting an extent to assign to a file.

The *ufs* file system uses the concept of **cylinder groups**. Cylinder groups are self-contained sections of a file system that are composed of inodes, data blocks, and bitmaps that indicate free inodes and data blocks. Allocation strategies in *ufs* attempt to place inodes and data blocks in proximity. This strategy limits, but does not eliminate the fragmentation process.

The VxFS file system provides the on-line administration utility *fsadm* to resolve the problem of fragmentation. One of the functions of the *fsadm* utility is to perform defragmentation on a mounted file system. To accomplish defragmentation, the *fsadm* utility

- removes unused space from directories,
- makes all small files contiguous,
- consolidates free blocks for file system use.

The *fsadm* utility may be run on demand and it should be scheduled regularly as a *cron* task.

2.3.2 Resizing the file system

When a file system is created, it is assigned a specific number of blocks and inodes. Changes of system usage requirements may result in file systems that are too small to support the desired system usage or are overly large for their intended use.

With a *ufs* file system there are traditionally three solutions to the problem of a file system that is too small:

1. Give some users a different file system.
2. Take a subdirectory of the file system and move it to a new file system.
3. Copy the entire file system to a larger file system.

When a file system is too large, standard file systems make reclaiming the unused space a matter of off-loading the contents of the file system and rebuilding it to a new size. The solution provided by the *ufs* system requires that the file system is unmounted. Users are unable to access the file system while it is modified.

The VxFS file system utility *fsadm* provides a mechanism to solve these problems without unmounting the file system or interrupting user's productivity. *fsadm* enables the VxFS file system to be resized while it is mounted. However, since the VxFS file system can only be mounted on **one** device, expanding the file system means that the underlying device must also be expandable while the file system is mounted.

Virtual disks cannot be reconfigured while being active (opened by the VxFS file system). If a virtual disk must be expanded, the VxFS file system must be unmounted first. Then an additional partition can be added at the end of a concatenated virtual disk. After reconfiguration with *dkconfig*(1M) the VxFS file system can be mounted again and resized with *fsadm*(1M). For details and an example refer to Section "Resizing a file system".

There is no need to use the *mkfs* command.□

For additional information about the functionality of virtual disk refer to the manual "Virtual Disks".

2.4 On-line backup

The VxFS file system provides two different methods for performing on-line backup of data:

- using a **snapshot**
- using virtual disks (mirrored disks)

Backup with a snapshot

The first method uses the "snapshot" feature of VxFS. A snapshot image of a file system is created by the *snapof* option of the *mount* command. The snapshot file system gives access to the data of the "snapped" original file system for the point in time that the snapshot was made. The original file system is active furthermore and can be changed.

When changes are made to the snapped file system, the old data is first copied to the snapshot so that it is retained. When the snapshot is read, if the data was changed the old data is returned, otherwise the current data from the snapped file system is returned. Backups are made by:

- copying selected files from the snapshot file system
- backing up the entire file system using the *volcopy* utility
- using the *fscat* utility to present a raw disk backup image of the snapped file system at the time the snapshot was made

For detailed information about performing on-line backups, see Chapter "On-line backup".

Backup with virtual disks

Another method of backing up the system uses virtual disks (mirrored disks) to make an exact copy of a file system to a second plex. The file system is frozen and the plex is detached and associated into a new volume which can then be mounted to backup selected files or stored directly to tape.

This approach is not recommended since it is somewhat slower than the snapshot approach and requires a great deal more disk space.

Additional information about virtual disks and backup procedures is located in the manual Virtual Disks.

2.5 Extended application interface

The VxFS file system interface conforms to SVID requirements and supports access using NFS and RFS. Any application running on a *ufs* file system should function identically on a VxFS file system. Remember that VxFS does **not** support hard links on directories.

2.5.1 Expanded application facilities

The VxFS file system provides the following facilities frequently associated with commercial applications:

- preallocate space for a file
- specify fixed extent size
- specify the form of caching for files and file systems
- specify the expected access pattern to a file

As these facilities are provided using VxFS-specific *ioctl*(2) system calls (refer to the manual pages for further information), existing Reliant UNIX system applications do not use these facilities. Custom applications can use these facilities to receive the benefits of the resulting performance improvement. For portability reasons, these applications should check what file system type they are using (ref *statvfs*(2)).

2.5.2 Support of large files and file systems

The maximum size of a file in a VxFS file system is restricted to 2^{44}-1 byte. This corresponds to 16 Tbyte - 1 data block (bsize). This limit is supported by Reliant UNIX from Version 5.44.

The VxFS file system does not have inherent limitations on the maximum number of concurrently-mounted file systems or concurrently accessed files.

2.5.3 Support of long file names

VxFS supports file names up to 255 characters in length. Files which use this facility can't be transferred to a number of other file systems without renaming.

2.6 Extended mount options

The VxFS file system supports extended mount options to specify:

- enhanced data integrity modes
- temporary file system mode

Details pertaining to the *mount* options can be found in Section "Choosing mount options".

2.6.1 Enhanced data integrity modes

The *ufs* file system is "buffered" in the sense that resources are allocated to files and data is written asynchronously to files. File systems are buffered in this way to provide better performance. In general, the buffering schemes work well without compromising data integrity.

If a system failure occurs while allocating space to a file, uninitialized data or data from another file may be present in the extended file after reboot. Also, data□written shortly before the system failure may be lost.

Using blkclear for data integrity

In environments where users prefer performance over absolute data integrity, the preceding situation is not of great concern. However, for environments where data integrity is preferred to performance, the VxFS file system provides a *mount -o blkclear* option that guarantees that uninitialized data never appears in a file.

Using closesync for data integrity

The *mount -o mincache=closesync* option is useful in desktop environments where users are likely to shut off the power on the machine without halting it first.□In this mode, any changes to the file which are not in the intent log are flushed to disk when the file is closed.

ℹ Users should be aware that there are performance penalties associated with these options.

2.6.2 Temporary file system mode

On many installed Reliant□UNIX systems, temporary file system directories (for example, */tmp* and */usr/tmp*) are commonly used to hold files that do not need to be retained when the system reboots. Since such file systems are temporary, there is no need for the underlying file system to maintain a high degree of structural integrity for these directories.

The VxFS file system provides a *-o nolog* option that allows the user to specify that no intent logging is required for this temporary data. With this option enabled, system performance is considerably improved (refer to Section "Enhanced file system performance").

2.7 Enhanced file system performance

The *ufs* file system uses block-based allocation schemes and provide sufficient random access to files, and acceptable latency on small files. For large files, however, throughput is limited by this block-based architecture: because of the fixed data block size (e.g. 16 Kbyte for *ufs* file system on RM600) less data than possible is transfered every physical I/O operation (at present 64 Kbyte per I/O operation with the RM disk controller).

The VxFS file system addresses this file system performance issue by using a different allocation scheme and by providing increased user control of allocation and I/O and caching policies. The following features are unique to VxFS:

- extent-based allocation
- data synchronous I/O
- direct I/O
- interface for caching advisories
- enhanced directory features
- explicit file alignment, extent size, and preallocation controls

To provide a conceptual understanding of how the VxFS allocation scheme differs from block-based allocation, an overview description of this architecture is covered in the following section, "Extent-based allocation". Details on the use of all of the preceding features can be found in Section "Cache advisories".

2.8 Extent-based allocation

An *extent* is defined as one or more adjacent blocks of data within the file system. When storage is added to a file on a VxFS file system, it is grouped in extents, as opposed to being allocated a block at a time as is done in the *ufs* file system.

Using extents means that disk I/O to and from a file can be done in units of multiple blocks. This type of I/O can occur if storage is allocated in units of consecutive blocks. For sequential I/O, multiple block operations are considerably faster than block-at-a-time operations. Almost all disk drives accept I/O operations of multiple blocks. The constant value *VX_MAXIO* in */usr/include/sys/fs/vx_param.h* specifies the amount of data which can be transfered per I/O operation (64 Kbyte for RM systems).

Extent allocation makes the interpretation of addressed blocks from the inode structure only slightly different from that of block-based inodes. The *ufs* file system inode structure contains the addresses of 12 direct blocks, one indirect block, and one double indirect block. An indirect block contains the addresses of other blocks. This allows an inode to directly address 12 blocks. With one indirect block there can be addressed *block_size*/4 more blocks, e. g. 2048 blocks with a block size of 8 Kbyte.

The VxFS file system inode is similar to the *ufs* inode. It contains 10 direct extents, which are pairs of starting block addresses and lengths in blocks. The VxFS file system also has one single indirect address and one double indirect address. These indirect address extents are allocated in multiple blocks. The size of all indirect address extents for a file must be the same size, and this value is stored in the inode. Directory inodes always use an 8 Kbyte extent size. By default, regular file inodes use an 8 Kbyte indirect address extent size.

3 Verifying installation

VxFS is the standard file system under Reliant UNIX. Only the root file system is a *ufs* file system

The VxFS installation (like *ufs*) contains

- a kernel component and
- a set of administrative utilities.

Verification of each component will be covered in the sections Kernel component verification and Utility component verification. Use *pkginfo* and *pkgparam* to verify the package installation.

3.1 Kernel component verification

To verify that the kernel component of the VxFS file system is installed, use the *crash*(1M) command to display a list of file system types (FSTypes) installed in the kernel. The following command:

```
# crash <<!
> vfssw
> !
```

should produce on a system the following screen output for example:

```
dumpfile = /dev/mem, namelist = /stand/unix, outfile = stdout
> FILE SYSTEM SWITCH TABLE SIZE = 13
SLOT  NAME     FLAGS□
1□□□□□bsdsfs□□□□42000
2□□□□□fdfs□□□□□□82001
3□□□□□fifofs□□□□42001
4□□□□□bs□□□□□□□□1
5□□□□□msockfs□□□42000
6□□□□□namefs□□□□82000
7□□□□□nfs□□□□□□□21000
8□□□□□proc□□□□□□82001
9□□□□□sockfs□□□□42000
10□□□□specfs□□□□42000
11□□□□ttyfs□□□□□42000
12□□□□ufs□□□□□□□11000
13□□□□vxfs□□□□□□11001
```

In addition to the familiar File System Types (FSTypes), *crash* also lists certain internal FSTypes, such as *specfs*, that have no user interface. If the name *vxfs* does not appear in the list produced by *crash*, the kernel portion of the file system has not been installed to the kernel (missing *idbuild*) or the system has not been restarted with the new kernel after *idbuild*.

3.2 Utility component verification

VxFS is similar to *ufs* in that file system-specific utilities are found in two directories: */etc/fs* and */usr/lib/fs*. Each of these directories contain a series of subdirectories; each of these subdirectories contain file system administrative utilities.

It is possible that one of these two directories is a symbolic link to the other, or that the directories themselves may be separate but the individual utilities are linked to the corresponding name in the other directory.

The standard UNIX system commands (e. g. *ls*, *mv*, *cp*) have got additional functionality to support the extent attributes that can be given to files on a VxFS file system.

Unlike earlier SINIX versions, the VxFS-specific functionality of commands *cp*, *cpio*, *ls* und *mv* is integrated into the corresponding standard command. Commands *getext* and *setext* are provided under */usr/bin*.

Check whether directory /*etc/fs/vxfs* contains the following files:

df   *fsadm*   *fsck*   *fsck1*   *fsck2* *fsdb* *fstyp* *mkfs* *mount*

Directory */usr/lib/fs/vxfs* additionally contains the following files:

ff *fscat* *labelit* *ncheck*

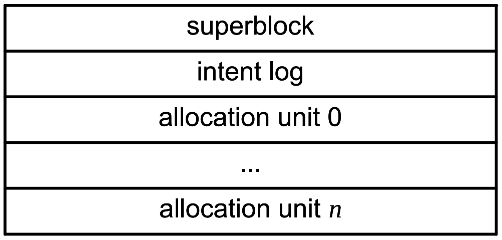
volcopy *vxdump* *vxrestore*

4 The VxFS disk structure

This chapter provides a description of the VxFS disk layout. It begins by defining the VxFS disk structure. It is illustrated in the following figure and is composed of:

- the superblock
- the intent log
- one or more allocation units

These elements are described in Section "List of relevant terms and acronyms" and are discussed in detail in the following sections.



superblock

intent log

allocation unit 0

...

allocation unit *n*

Disk space is allocated by the system in 512-byte sectors. An integral number of sectors in powers of 2 are grouped to form a block. VxFS allocates disk space to files in multiples of blocks. VxFS supports block sizes of 1024, 2048, 4096, and 8192 byte. The block size may be specified as an argument to the *mkfs* utility; the default block size is 1024 byte. Block size may vary between VxFS file systems mounted on the same system.

The VxFS disk structure supports up to 2^{44} - 1 files, up to 2^{44} - 1 disk blocks, up to 2^{44} - 1 bytes per file (16 Tbytes).

4.1 The superblock

The *superblock* is in a fixed location offset from the start of the partition or logical volume by 1024 bytes. The superblock contains

- creation and modification dates,
- label information,
- information about the size and layout of the file system and
- the count of available resources.

4.2 The intent log

In the event of system failure, the VxFS file system uses intent logging to guarantee file system integrity. This technique is based on transaction processing theories, and it requires that changes in the file system are written to the disk in a predefined order.

The intent log is a circular activity log with a default size of 256 sectors on a RM system. It is located behind the superblock. This log is a record of the intention of the system to update a file system structure. An update to the file system structure (a **transaction**) is divided into subfunctions based on the data structure type to be updated (e.g. management of extents, inodes, directories). A composite log record of the transaction is created that contains the subfunctions that constitute the transaction.

Example

The creation of a file that would expand the directory in which the file is contained, will produce a transaction consisting of the following subfunctions:

- a free extent map update for the allocation of the new directory block
- a directory block update
- an inode modification for the directory size change
- an inode modification for the new file
- a free inode map update for the new allocation of the new file

VxFS maintains log records in the intent log for all pending changes to the file system structure, and ensures that the log records are written to disk in **advance** of the changes to the file system. In the event of a system failure, the pending changes to the file system are either nullified or completed by the *fsck* utility (see also Section "Fast file system recovery").

The VxFS intent log only records changes to the file system **structure**; except for synchronous writes changes to internal data of files are **not** logged in the intent log (see Using closesync for data integrity).

4.3 The allocation units

An ***allocation unit*** in the VxFS file system is similar in concept to the *ufs* ,,cylinder group". The VxFS allocation unit is depicted in the following figure:

allocation unit header
free resource summaries
free inode map (imap)
extended inode operations map (eimap)
free extent map (emap)
inode list
padding
data blocks

The number and size of allocation units can be specified when the file system is made. All of the allocation units, except possibly the last one, will be of equal size. The last allocation unit can have a partial set of data blocks to allow use of all available blocks on a partition.

Allocation unit header

The allocation unit header contains a copy of the file systems superblock that is used to verify that the allocation unit matches the superblock of the file system.

Free resource summaries

The free resource summaries contain the number of inodes with extended operations pending, number of free inodes, and number of free extents in the allocation unit.

Free inode map

The free inode map is a bitmap that indicates which inodes are free and which are allocated. A free inode is indicated by the bit being on. Inodes zero and one of the first allocation unit are reserved by the file system; inode two is the inode for the *root* directory; inode three is the inode for the *lost+found* directory.

Extended inode operations map

The extended inode operations map is in the same format as the free inode map. The extended inode operations map keeps track of inodes on which operations would remain pending for too long to reside in the intent log. To prevent the intent log from wrapping and the transaction getting overwritten, the required operations are stored in the affected inode. This map is then updated to identify the inodes that have extended operations which need to be completed. These updates allow the *fsck* utility to quickly identify which inodes had extended operations pending at the time of a system failure.

Free extent map

The free extent map is a series of independent 512-byte bitmaps that are each referred to as a free extent map section. The first region of 2048 bits represents a section of 2048 one-block extents. A free block is represented by a set bit. The second region of 1024 bits represent a section of 1024 two-block extents. This sectioning continues for all powers of 2 up to the single bit that represents one 2048 block extent.

The one-block bitmaps always represent the true allocation of blocks from the allocation unit. The remaining bitmaps remap these same blocks, in a "binary-buddy" scheme, in increasingly larger-sized groups. As smaller extents are needed, the larger groups of blocks mapped by the buddy maps are broken apart to create the smaller extents.

Inode list

The inode list is a list of the data structures (inodes) that contain information about the files. Each inode stores for example

- the file length
- a number of link count
- owner and group IDs
- access privileges and
- data to the extent blocks that contain the file's data (the block number of the first block of the extent and the length of the extent).

There is one inode in the list for every file.

Padding

It may be desirable to align data blocks to a physical boundary. To facilitate this, the system administrator may specify that a gap be left between the end of the inode list and the first data block (see option *aupad* of the *mkfs*(1M) command in Section "mkfs(1M)").

Data blocks

The balance of the allocation unit is occupied by data blocks.

5 On-line backup

The VxFS file system provides a mechanism for taking snapshot images of mounted file systems which is useful for making backups. The snapshot is a consistent view of the file system "snapped" at the point in time the snapshot is made. Selected files can be backed up from the snapshot using standard utilities such as *cpio* or *cp*, or the entire file system image can be backed up using the *volcopy*, *vxdump*, or *fscat* utilities.

A snapshot file system is always read-only and exists only as long as it and the file system that has been snapped are mounted. A snapped file system cannot be unmounted until any corresponding snapshots are unmounted. A snapshot file system ceases to exist when unmounted. While it is possible to have multiple snapshots of a file system made at different times, it is not possible to make a snapshot of a snapshot.

This chapter describes the creation of snapshot file systems and gives some examples of backing up all or part of a file system using the snapshot mechanism.

5.1 Disk structure of snapshot file systems

A snapshot file system consists of:

- a superblock
- a blockmap
- data copied from the snapped file system

The superblock is the same as the superblock of a normal VxFS file system except that the magic number is different and many of the fields are meaningless.

Starting at the block immediately after the superblock is the *blockmap*. It contains one entry (of type *daddr_t*) for each block on the snapped file system. Initially all entries are zero. If data of the snapped file system is changed, first the "old" data is copied to the data block of the snapshot file system. The appropriate entry in the blockmap is changed to contain the block number on the snapshot file system that holds the data from the snapped file system.

After the blockmap are the actual data blocks used by the snapshot file system. These are filled by any data copied from the snapped file system, starting from the front of the data block area.

5.2 How a snapshot file system works

A snapshot file system is created by mounting an empty disk (or volume) as a snapshot of a currently-mounted file system. The blockmap and superblock are initialized and then the currently mounted file system is frozen (refer to the *VX_FREEZE* option of the *ioctl*(2) system call in *vxfsio*(7)). Once the file system to be snapped is frozen, the snapshot is enabled and mounted and the snapped file system is thawed (refer to the *VX_THAW* option of the *ioctl*(2) system call in *vxfsio*(7)). The snapshot appears as an exact image of the snapped file system at the time the snapshot was made, although not the entire snapped file system is copied but only the changed blocks since the snapping time. This method is explained in the following.

Initially, the snapshot file system satisfies *read*(2) requests by simply finding the data on the snapped file system and returning it to the requesting process. When an inode update or write changes the data in block X of the snapped file system, the old data is first read and copied to the snapshot before the snapped file system is updated. The blockmap entry for block X is changed from 0 (which indicates the data for block X can be found on the snapped file system) to the block number on the snapshot file system containing the old data.

A subsequent read request for block X on the snapshot file system will be satisfied by checking the blockmap entry for block X and reading the data from the indicated block on the snapshot file system, rather than from block X on the snapped file system. Subsequent writes to block X on the snapped file system do not result in additional copies to the snapshot file system since the old data only needs to be saved once.

All updates to the snapped file system for inodes, directories, data in files, extent maps, etc, are handled in this fashion so that the snapshot can present a consistent view of all file system structures for the snapped file system for the time when the snapshot was created. As data blocks are changed on the primary file system, the snapshot will gradually fill with data copied from the snapped file system.

The amount of disk space required for the snapshot depends on the rate of change of the snapped file system and the amount of time the snapshot is maintained. In the worst case, the snapped file system is completely full and every file is removed and rewritten. The snapshot file system would need enough blocks to hold a copy of every block on the snapped file system, plus a few blocks for the data structures which make up the snapshot file system or roughly 101 percent of the size of the snapped file system. Normally, most file systems don't undergo changes at this extreme rate.

If a snapshot file system runs out of space for changed data blocks, it is disabled and all further accesses to it fail.

5.3 Using a snapshot file system for backup

Once a snapshot file system is created, it can be used to perform a consistent backup of the snapped file system. For this purpose, standard backup programs can be used:

- backup programs that function using the standard file system tree (path- and filenames), as for example *cpio*
- backup programs that access the disk structures of the file system, as for example *volcopy* (...) and *vxdump* (...).

Other backup programs that use knowledge of the VxFS on-disk image can use the *fscat* command to obtain a raw image of the entire file system identical to that which would have been obtained by a *cat* of the disk device containing the snapped file system at the exact moment the snapshot was created. If an application requires random access to the snapshot image, it can use the *VX_SNAPREAD ioctl* described in Section "Cache advisories". The *VX_SNAPREAD ioctl* takes arguments similar to those of the read system call and returns the same results as would have been obtained by performing a read on the disk device containing the snapped file system at the exact time the snapshot was created. In both cases, however, the snapshot file system provides a consistent image of the snapped file system with all activity complete or blocked at the system call level; it is an instantaneous read of the entire file system. This is a marked contrast to the results that would be obtained by a *cat* or *read* of the disk device of an active file system.

If a complete backup of a snapshot file system is made through a utility such as *volcopy* and is later restored, it will be necessary to *fsck* the restored file system since the snapshot file system is only consistent and not clean. The file system may have some extended inode operations that must be completed, though there should be no other changes. The snapshot file system cannot be *fsck*ed since it is not writable.

5.4 Creating a snapshot file system

A snapshot file system is created by using the *snapof*= suboption of the *mount* command. The *snapsize*= suboption is the device size, or if a size smaller then the entire device is desired. The snapshot file system must be created large enough to hold any blocks on the snapped file system that may be written to while the snapshot file system exists.

! If a snapshot file system runs out of blocks to hold copied data, it will be disabled and all further access to the snapshot file system will fail.

During a period of low system usage (for example on nights and weekends), the snapshot only needs to contain two to six percent of the blocks of the snapped file system. During a busy period, the snapshot of an "average" file system might require 15 percent of the blocks of the snapped file system, though most file systems don't experience this much turnover of data over an entire day. The system administrator should manage the blocks allocated to the snapshot based on file system usage, duration of backups, etc.

! Any data on the disk used for the snapshot will be overwritten and lost.

A snapshot file system ceases to exist when unmounted. Snapshot file systems must be unmounted before the snapped file system can be unmounted. If remounted it will be a fresh snapshot of the snapped file system.

Neither *fuser* nor *mount* will indicate that a snapped file system is unmountable because a snapshot of it exists.

Examples of making a backup

Here are some typical examples of making a backup of a file system */home* which exists on disk */dev/ios0/sdisk000s6* using a snapshot file system on */dev/ios0/sdisk001s6* .

- Backup files changed within the last week using *cpio*:

  ```
  # mount -F vxfs -o snapof=/dev/ios0/sdisk000s6, \
  □□□□□□□snapsize=100000 /dev/ios0/sdisk001s6 /bckup/home
  # cd /bckup
  # find home -ctime -7 -depth -print | cpio -oc > \
  □□□□□□□/dev/ios0/rstape003
  # umount /bckup/home
  ```

- Backup entire file system using *volcopy*:

  ```
  # mount -F vxfs -o snapof=/dev/ios0/sdisk000s6, \
  □□□□□□□snapsize=100000 /dev/ios0/sdisk001s6 /bckup/home
  # cd /bckup
  # volcopy -F vxfs home /dev/ios0/sdisk001s6 001s6 \
  □□□□□□□/dev/ios0/rstape003 tape77
  # umount /bckup/home
  ```

- To do a full backup of */dev/ios0/sdisk001s6* and use *dd* to control blocking of output onto tape device using *vxdump*:

  ```
  # vxdump f - /dev/ios0/sdisk000s6 | dd bs=128k > \
  □□□□□□□/dev/ios0/rstape003
  ```

- To do a level 3 backup of */dev/ios0/sdisk001s6* and collect those files that have changed in the current directory:

  ```
  # vxdump 3f - /dev/ios0/sdisk000s6 | vxrestore -xf
  ```

- To do a full backup of a snapshot file system:

  ```
  # mount -o snapof=/dev/ios0/sdisk000s6,snapsize=100000 \
  □□□□□□□/dev/ios0/sdisk001s6 /bckup
  # vxdump f - /dev/ios0/sdisk001s6 | dd bs=128k > \
  □□□□□□□/dev/ios0/rstape003
  ```

The *vxdump* program will ascertain that */dev/ios0/sdisk001s6* is a snapshot mounted as */bckup*.

5.5 Performance of snapshot file systems

Snapshot file systems maximize the performance of the snapshot at the expense of the snapped file system. Reads from a snapshot file system will typically perform at nearly the throughput of reads from a VxFS file system, allowing backups to proceed at the full speed of the VxFS file system.

Writes to the snapped file system, however, typically average two to three times as long as without a snapshot, since the initial write to a data block now requires a read of the old data, a write of the data to the snapshot, and finally the write of the new data to the snapped file system. If multiple snapshots of the same snapped file system exist, writes will be even slower. Only the initial write to a block suffers this penalty, however, so operations like writes to the intent log or inode updates proceed at normal speed after the initial write.

Reads from the snapshot file system are impacted if the snapped file system is busy, since the snapshot reads are slowed by all the disk I/O associated with the snapped file system.

6 Performance and tuning

For any file system, the ability to provide peak performance is important. Adjusting the available file system options provides a way to accomplish optimal system performance. The file system options that provide you with the tools to enhance performance include:

- modifying the block size for the file system
- modifying the size of the intent log
- modifying the file system *mount* options
- tuning kernel variables

Each of the following sections addresses one of these options.

6.1 Choosing a block size

The standard *ufs* file system has an 8 Kbyte block size with a 1 Kbyte fragment size (older systems) or a 16 Kbyte block size with a 2 Kbyte fragment size (RM systems). The following description is based on an *ufs* file system with 8 Kbyte block size and 1 Kbyte fragment size.

For small files (for example up to 8 Kbyte on an older system) space is allocated in 1 Kbyte increments. Since the majority of files are small, the fragment facility saves a large amount of space when compared to allocating space 8 Kbyte at a time.

The allocation unit in VxFS is a block. There are no fragments, since storage is allocated in extents that consist of one or more blocks. For the most efficient space utilization, the smallest block size available on the system should be used. The smallest block size available is 1 Kbyte, and is the default block size for VxFS file systems created on the system. Unless there are special concerns, there should never be a need to specify a block size when creating file systems.

For large file systems (over a Gbyte), with relatively few files, the system administrator may wish to experiment with larger block sizes. The tradeoff for specifying larger block sizes is the amount of space used to hold the free extent bitmaps versus units of allocation and maximum extent size. The largest extent size available is 256 Kbyte blocks, whether the block size is 1 Kbyte or 8 Kbyte. To create a contiguous file of 1 Gbyte with just one extent on a VxFS file system, a block size of at least 4 Kbyte must be used.

Note that the throughput depends on the extent size not on the block size. High performance of a VxFS file system needs large extents when a file grows (refer to Section "Extent reorganization" for extent reorganization with the *fsadm*(1M) utility).

Overall file system performance may be improved or degraded by changing block size. It is recommended that the default values for the system are used. However, the easiest way to judge which block sizes will provide the greatest system efficiency is to try representative system loads against various sizes and pick the fastest.

6.2 Choosing an intent log size

The *mkfs*(1M) utility uses a default intent log size of 256 blocks. This size is sufficient for most workloads. If the system is used as an *nfs* server or for intensive synchronous write workloads, such as a database, performance may be improved using a larger log size.

There are several system performance benchmark suites for which VxFS performs better with larger log sizes. An intent log size of 512 or 1024 blocks may be chosen, based upon expected load. The best way to pick the log size is to try representative system loads against various sizes and pick the fastest.

When choosing a larger intent log size, recovery time will be proportionately longer, and the file system may consume more system resources, such as memory.

6.3 Choosing mount options

In addition to the standard mount mode, the VxFS file system provides *blkclear*, *nolog*, and *nodatainlog* modes of operation. Also, the caching behavior can be altered with the *mincache* option and the behavior of *O_SYNC* reads and writes can be altered with the *convosync* option.

blkclear mode

The *blkclear* mode is used in increased data security environments. The *blkclear* mode guarantees that uninitialized storage never appears in files. The increased integrity is provided by clearing extents on disk when they are allocated within a file. Extending writes are not effected by this mode. A *blkclear* mode file system should be approximately 15% slower than a standard mode VxFS file system depending on the workload.

nolog mode

The *nolog* mode is used for temporary file systems (such as */tmp*). As the name implies, the intent log is not used. All I/O requests, including synchronous I/O, are performed asynchronously. Data loss on *nolog* file systems is likely in the event of a system crash. It is recommended that *nolog* mode file systems are remade using *mkfs*(1M) during system startup, as opposed to attempting to clean them using *fsck*(1M). Depending on the workload, a *nolog* mode file system should be significantly faster than a standard mode VxFS file system. Performance improvement varies depending on the operations being performed.

nodatainlog mode

The *nodatainlog* mode should be used on systems with disks that don't support bad block revectoring. Normally, a VxFS file system uses the intent log for synchronous writes. The inode update and the data are both logged in the transaction, so a synchronous write only requires one disk write instead of two. When the synchronous write returns to the application, the file system has told the application that the data is already written. If a disk error causes the data update to fail, then the file must be marked bad and causes the entire file to be lost.

If a disk supports bad block revectoring, then a failure on the data update is unlikely, so logging synchronous writes should be allowed. If the disk doesn't support bad block revectoring, then a failure is more likely, so the *nodatainlog* mode should be used.

A *nodatainlog* mode file system should be approximately 50% slower than a standard mode VxFS file system for synchronous writes. Other operations are not affected.

mincache mode

The *mincache* mode has three suboptions. The *mincache=closesync* mode is used in desktop environments where users are likely to shut off the power on the machine without halting it first. In this mode, any unlogged changes to the file are flushed to disk when the file is closed by all users.

To improve performance, most file systems don't synchronously update data and inode changes to disk. If the system crashes, files that have been updated within the past minute are in danger of losing data. With the *mincache=closesync* mode, if the system crashes or is switched off, only files that are currently being written can lose data. A *mincache=closesync* mode file system should be approximately 15% slower than a standard mode VxFS file system depending on the workload.

The *mincache=direct* and *mincache=dsync* suboptions are used in environments where applications are experiencing reliability problems caused by asynchronous I/O. The direct suboption guarantees that all asynchronous I/O requests to files will be handled as if the *VX_DIRECT* caching advisory had been specified. The *dsync* suboption guarantees that all asynchronous I/O requests to files will be handled as if the *VX_DSYNC* caching advisory had been specified. Both the *mincache=direct* and *mincache=dsync* modes also flsuh file data on close like *mincache=closesync*.

Since the *mincache=direct* and *mincache=dsync* modes change asynchronous I/O to synchronous I/O, there is a substantial performance degradation for most applications. Since the *VX_DIRECT* advisory doesn't allow any caching of data, applications that would normally benefit from caching will usually experience less degradation with the *mincache=dsync* mode.

convosync mode

The *convosync* mode has three suboptions. The *convosync=closesync* converts synchronous writes to asynchronous writes. The *convosync=direct* converts synchronous reads and writes to direct reads and writes (ref *VX_DIRECT*). The *convosync=dsync* converts synchronous writes to data synchronous writes. All three modes flush changes to the file when it is closed.

These modes can be used to speed up applications that use synchronous I/O. Normally, a synchronous I/O updates both the data and the inode access or modification times to disk. With the *convosync=dsync* and *convosync=direct* modes, the inode time will not be updated to disk immediately. If the system crashes, the time update can be lost. Applications that use synchronous I/O because they depend on reliable inode time updates should not be run on a file system mounted with the *convosync* option.

⚠ The *convosync=closesync* option actually changes the synchronous I/O into asynchronous I/O. This causes applications that use synchronous I/O for data reliability to fail if the system crashes and data is lost. Extreme care should be taken when using this option.

⚠ When the *convosync* mount option is used, the VxFS file system is no longer POSIX compliant.

Except for the *nolog* mode all the modes can be used in combination. For example, a secure desktop machine might want to use both the *blkclear* and *mincache=closesync* modes.

6.4 Kernel tunables

There are several kernel tunables that pertain to the VxFS file system. These tunables are discussed in the following sections.

6.4.1 Buffer cache space

The VxFS file system uses the kernel buffered I/O facility to read and write file system structure. It is important that sufficient resources be available for use by the buffer cache. There is one buffer cache tunable:

BUFHWM BUFferHigh Water Mark (*BUFHWM*)the maximum amount of space available for use by block I/O buffers on RM systems in units of 16□Kbyte

This and other tunables are found in */etc/conf/cf.d/mtune* (or */etc/conf/cf.d/stune*, if changed with *idtune*). The *mtune* file contains the system default, minimum, and maximum values for each tunable. The *stune* file contains entries for tunables that cannot use the default value. The number of buffers and space available for buffers varies according to system size. A typical value for a RM 600 system is 256 (units of 16 Kbyte). The value is tunable up to 1024.

If insufficient buffer cache space is available–which can occur on a heavily loaded system–the file system can thrash on the buffer cache and adversely effect file system performance. However, space reserved for the buffer cache is unavailable for other purposes. *sar*(1M) can be used to determine the buffer cache hit rate for reads. The hit rate should be better than 96%, depending on the type of load. If the hit rate is not 96%, increasing the *BUFHWM* should help.

6.4.2 Internal inode table size

In the *ufs*, and VxFS file system types, inodes are cached in a "per file system table", known as the inode table. Each file system type has a tunable to determine the number of entries in its inode table. For *ufs*, the tunable is *UFSNINODE*.

The VxFS file system type (FSType) uses the define *VX_NINODE* in */etc/conf/pack.d/vxfs/space.c* as the number of entries in the VxFS inode table. The current default inode table size for VxFS is being calculated from the kernel parameters *NPROC* and *MAXUSERS*. This value (*ninode*) is defined in *mtune* and corresponds with *ufsinode*. Guidelines for optimal inode table size are the same as for *ufs*.

6.4.3 Monitoring free space

The current release of VxFS works best if the percentage of free space in the file system does not get below 10%. Regular use of the *df*(1M) command to monitor free space is desirable. Full file systems may have an adverse effect on file system performance. Full file systems should have files removed, or should be expanded (see *fsadm* from fsadm(1M) for a description of online file system expansion).

6.4.4 Monitoring fragmentation

Fragmentation reduces performance and availability. Regular use of the fragmentation reporting and reorganization facilities of *fsadm*(1M) is encouraged. The easiest way to ensure that fragmentation does not become a problem is to schedule regular defragmentation runs from *cron*(1M).

Defragmentation scheduling should range from daily (for heavily-used file systems), to weekly or monthly for more lightly-used file systems. Extent fragmentation should be monitored with *fsadm*(1M) or the *-o s* option of *df*(1M). There are three factors which can be used to determine the degree of fragmentation:

- percentage of free space in "small" extents
  - "small" extents are extents with less than (*IADDREXTSIZE/fs_bsize*) blocks in length
  - *IADDREXTSIZE* is the indirect address extent size in */usr/include/sys/fs/vx_param.h*.
  - *fs_bsize* is the block size as used with *mkfs*(1M).
  - For example in the case of a block size of 1 Kbyte and an indirect address extent size of 8 Kbyte the value is 8.
- percentage of free space in extents of less than 64 blocks in length
- percentage of space available in extents of length 64 blocks or greater

An unfragmented file system will have the following characteristics:

- less than 1% of free space in "small" extents
- less than 5% of free space in extents of less than 64 blocks in length
- more than 5% of the total file system size is available as free extents in lengths of 64 or more blocks

A badly fragmented file system will have one or more of the following characteristics:

- more than 5% of free space in "small" extents
- more than 50% of free space in extents of less than 64 blocks in length
- less than 5% of the total file system size is available as free extents in lengths of 64 or more blocks

The optimal period for scheduling of extent reorganization runs can be determined by choosing a best guess interval, scheduling *fsadm* runs at the initial interval, and running the extent fragmentation report feature of *fsadm* before and after the reorganization. The "before" result is the degree of fragmentation prior to the reorganization. If the degree of fragmentation is approaching the figures for bad fragmentation, then the interval between fragmentation runs should be reduced.

The "after" result is an indication of how well the reorganizer is performing. In case of a too high degree of fragmentation *fsadm* isn't able to reorganize the file system. The file system may be a candidate for expansion. (Full file systems tend to fragment and are difficult to defragment.) It is also possible that the reorganization is not being performed at a time during which the file system in question is relatively idle.

Directory reorganization is not nearly as critical as extent reorganization, but regular directory reorganization will improve performance. It is advisable to schedule directory reorganization for file systems when the extent reorganization is scheduled. The following is a sample script that is run daily at 5:00 A.M. from *cron*(1M) for a number of file systems.

Example: Sample script for directory reorganization

```
outfile=/usr/spool/fsadm/out.`/bin/date +%m%d'
for i in /home /home/user1 /home/user2 /home/user3 /home/user4
do
□□□□□□/bin/echo "Reorganizing $i"
□□□□□□/bin/timex /etc/fs/vxfs/fsadm -e -E -s $i
□□□□□□/bin/timex /etc/fs/vxfs/fsadm -s -d -D $i
done > $outfile 2>&1
```

When directory reorganization changes a directory, it updates the modification time on the directory inode.

7 Application interface

7.1 Cache advisories

The VxFS file system allows an application to set cache advisories for use when accessing files. These advisories are memory only and they do not persist across reboots. In the current version, advisories are maintained on a per file basis, not per file descriptor. This means only one set of advisories can be in effect for all accesses to the file. If two conflicting applications set different advisories, both use the last advisories that were set.

All advisories are set using the *VX_SETCACHE ioctl* command. The current set of advisories can be obtained with the *VX_GETCACHE ioctl* command. For details on the use of these *ioctl* commands see the *vxfsio*(7) man page.

7.1.1 Direct I/O

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.

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 transfer size must be a sector size multiple. Also, the user buffer needs to be aligned on a sector boundary and subpage requests shouldn't cross page boundaries.

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

When direct I/O is enabled, the assumption is made that the direct I/O is an important application. No cached data is maintained for the file, so all nondirect operations are uncached, and they invalidate any data they have put in the cache before they return to their caller. The direct I/O feature can provide significant performance gains for some applications.

Since 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 the direct I/O doesn't allocate storage or extend the file, the inode isn't immediately written. For database applications that are rewriting existing files, using direct I/O can reduce by half the time to do a disk write by delaying the inode time updates.

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 asynchronous I/O with much less CPU overhead. If an application requests direct I/O and doesn't meet the alignment constraints, any cached data for the file is invalidated after each I/O.

If partial block updates are being done, the block has to be written using a read-modify-write cycle. In this case, using the data synchronous I/O advisory instead of the direct I/O advisory could work better since the block might be found in the cache, thus avoiding a read. If the file is being extended or storage is allocated, then direct I/O must write the inode change before returning to the application. This eliminates the biggest performance advantage of direct I/O. However, if the direct I/O is 8 Kbyte or smaller, then the write may be performed as a logged write. A logged write stores both the inode changes and the user data in the intent log. With logged writes, the file system can return to the application after the intent log is written. Since only one disk I/O is needed instead of two, this can restore extending direct I/O requests to their original speed.

7.1.2 Data synchronous I/O

If the *VX_DSYNC* advisory is set, all writes are done as data synchronous I/O. In synchronous I/O mode, the data is written, the inode is only written once with updated times; if necessary, an increased file size is written before the write returns to the user. In data synchronous I/O, the data is transferred to disk synchronously before the write returns to the user. If the file is not extended by the write, the times are updated in memory, and the call returns to the user. If the file is extended by the operation, the inode is written before the write returns.

Like direct I/O, the data synchronous I/O feature can provide significant application performance gains. Since data synchronous 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 the data synchronous I/O doesn't allocate storage or extend the file, the inode isn't immediately written. For database applications that are rewriting existing files, using data synchronous I/O can reduce by half the time to do a disk write, by delaying the inode time updates.

The data synchronous I/O doesn't have any alignment constraints, so applications that find it difficult to meet the alignment constraints of direct I/O should use data synchronous I/O.

If the file is being extended or storage is allocated, then data synchronous I/O must write the inode change before returning to the application. This eliminates the performance advantage of data synchronous I/O. If the data synchronous I/O is 8K or smaller, then the write may be performed as a logged write. A logged write stores both the inode changes and the user data in the intent log. With logged writes, the file system can return to the application after the intent log is written. Since only one disk I/O is needed instead of two, this can restore extending data synchronous I/O requests to their original speed.

7.1.3 Other advisories

VX_SEQ

The *VX_SEQ* advisory indicates the file is being accessed sequentially. When the file is being read, the maximum read-ahead is always performed. When the file is written, instead of trying to determine whether the I/O is sequential or random based on the write offset, sequential I/O is assumed. The pages for the write are not immediately flushed. Instead, pages are flushed behind the current write point.

VX_RANDOM

The *VX_RANDOM* advisory indicates the file is being accessed randomly. For reads, this disables read-ahead. For writes, this disables the flush behind. The data is flushed when it is invalidated out of the page cache.

VX_NOREUSE

The *VX_NOREUSE* advisory is used as a modifier. If both *VX_RANDOM* and *VX_NOREUSE* are set, pages are immediately freed and put on the quick reuse free list as soon as the data has been used. If *VX_NOREUSE* is set when doing sequential I/O, pages are also put on the quick reuse free list when they are flushed. The *VX_NOREUSE* slows down access to the file, but it can reduce the cached data held by the system. This can allow more data to be cached for other files and may speed up those accesses.

7.2 Extent information

The *VX_SETEXT ioctl* command allows an application to reserve space for a file, and set fixed extent sizes and file allocation flags. The current state of this information can be obtained by applications using the *VX_GETEXT ioctl* (the *getext*(1) command provides access to this functionality). For details, see the *getext*(1) and *setext*(1) section in Chapter "Utilities and commands", for details to *vxfsio*(7) and *ioctl*(2) system calls see the man pages.

Each invocation of the *VX_SETEXT ioctl* sets all the elements in the *vx_ext* structure. When using the *ioctl*, always use a *VX_GETEXT* to read the current settings, modify the values you want to change, and call *VX_SETEXT* to set the values.

Follow this procedure carefully. Otherwise, a fixed extent size could be cleared when the reservation is changed.

7.2.1 Space reservation

Storage can be reserved for a file at any time. When a *VX_SETEXT ioctl* is issued, the reservation value is set in the inode on disk. If the file size is less than the reservation amount, the kernel allocates space to the file from the current file size up to the reservation amount. When the file is truncated, space below the reservation amount is not freed. The *VX_TRIM*, *VX_NOEXTEND*, *VX_ALIGN*, *VX_CHGSIZE*, *VX_NORESERVE* and

VX_CONTIGUOUS flags can be used to modify reservation requests.

VX_TRIM

If the *VX_TRIM* flag is set, when the inode is inactivated the reservation is trimmed to match the file size. Any unused space 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, any extra space will be released.

VX_NOEXTEND

If the *VX_NOEXTEND* flag is set, an attempt to write beyond the current reservation does not allocate new space for the file. To allocate new space to the file, the space reservation must be increased. This can be used like *ulimit*(2) to prevent a file from using too much space.

VX_ALIGN

If the *VX_ALIGN* flag is set, all new extents are aligned on a *ext_size* boundary relative to the starting block of an allocation unit (ref to the *vx_ext* struct in */sys/fs/vx_ioctl.h*. If *VX_CONTIGUOUS* is set, the single extent allocated during this invocation is not subject to the alignment restriction.

VX_CONTIGUOUS

If the *VX_CONTIGUOUS* flag is set, any space allocated to satisfy the current reservation request is allocated in *one* extent. If there isn't one extent large enough to satisfy the request, the request fails. For example, if a file is created and a 1 Mbyte contiguous reservation is requested, the file size is set to 0 and the reservation to 1024 (assuming a 1 Kbyte block size). The file will have one extent of 1024 blocks. If another reservation request is made for a 3 Mbyte contiguous reservation, the new request will find the first 1 Mbyte is already allocated and allocate a 2 Mbyte extent to satisfy the request. If there are no □□ 2 Mbyte extents available, the request fails.

VX_NORESERVE

If the *VX_NORESERVE* flag is set, the reservation value in the inode is only changed in memory, not on the disk. This flag is used by applications to do temporary reservation. Any space past the end of the file is given up when the file is closed. For example, if the *cp*(1) command is copying a file that is 1 MbyteMbyte long, it can request a 1 Mb reservation with the *VX_NORESERVE* flag set. The space will be 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 will be released. When the program finishes, there is no cleanup because the reservation was never recorded on disk.

VX_CHGSIZE

If the *VX_CHGSIZE* flag is set, the file size will be increased to match the reservation amount. This flag can be used to create files with uninitialized data. Because this allows uninitialized data in files, it is restricted to users with root privileges.

It is possible to use these flags in combination. For example, using *VX_CHGSIZE* and *VX_NORESERVE* will change the file size but not set any reservation. When the file is truncated, the space will be freed. If the *VX_NORESERVE* flag had not been used, the reservation would have been set on disk as well as the file size. Space reservation is used to make sure applications don't 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, space reservation can avoid using indirect extents. It can also improve performance and reduce fragmentation by guaranteeing that the file consists of large contiguous extents. Sometimes when critical file systems run out of space, *cron* jobs, mail, or printer requests fail. These failures are harder to track if the logs kept by the application can not be written due to a lack of space on the file system. By reserving space for key log files, the logs won't fail when the system run out of space. Process accounting files can also have space reserved so accounting records won't be lost if the file system runs out of space. In addition, by using the *VX_NOEXTEND* flag for log files, the maximum size of these files can be limited. This can prevent a runaway failure in one component of the system from filling the file system with error messages and causing other failures. If the *VX_NOEXTEND* flag is used for log files, to avoid losing information the logs should be cleaned up before they reach the size limit.

7.2.2 Fixed extent sizes

The VxFS file system uses the I/O size of write requests, and a default policy, when allocating space to a file. For some applications this may not work out well. These applications can set a fixed extent size. All new extents allocated to the file are the fixed extent size.

By using a fixed extent size, an application can reduce allocations and guarantee sufficient extent sizes for a file. An application can reserve most of the space a file needs, and then set a relatively large fixed extent size. If the file grows beyond the reservation, any new extents are allocated in the fixed extent size.

Another use of a fixed extent size occurs with sparse files. The file system usually does I/O in page size multiples. When allocating to a sparse file, the file system allocates pages as the smallest default unit. If the application always does subpage I/O, it can request a fixed extent size to match its I/O size and avoid wasting extra space.

When setting a fixed extent size, an application should not select too large a size. When all extents of the required size have been used, attempts to allocate new extents fail; this failure can happen even though there are blocks free in smaller extents.

The *mkfs*(1M) command is used to make a file system with all the allocation units aligned to physical disk boundaries (see Alignment).

8 Utilities and commands

This chapter contains descriptions for the utilities and commands that are an integral part of the VERITAS File System (VxFS). Each description contains general information about the command, the synopsis and the command options. For additional information about the VxFS-specific commands, refer to the manual pages. You can find a complete list of all error messages in Chapter "Messages".

VxFS exists within a **virtual file system** architecture that allows multiple file system types (FSTypes) to coexist within the UNIX system kernel. Although each file system has characteristic features that are not shared with any other FSTypes, they also have some shared features. The VxFS administrative commands provide a common interface that allows the administrator to maintain file systems of different types.

The file system commands discussed in this chapter are:

Universal commands:

getext(1)
: displays extent attribute information (...)

setext(1)
: changes extent attribute information (...)

VxFS specific user commands:

df(1)
: shows free resources of the file system (...)

ff(1M)
: find files (see Section "ff(1M)")

fsadm(1M)
: file system on-line administration (...)

fscat(1M)
: shows a dump of the file system (...)

fsck(1M)
: checks the file system for consistency (...)

fsdb(1M)
: examines and repairs damaged file systems (...)

fstyp(1M)
: checks the type of a file system (...)

labelit(1M)
: labels file systems and tapes (...)

mkfs(1M)
: creates a VxFS file system (...)

mount(1M)
: mounts VxFS file systems (...)

ncheck(1M)
: checks the attach of inodes and pathnames (...)

volcopy(1M)
: makes a copy of a file system (...)

vxdump(1M)
: backs up a file system (...)

vxrestore(1M)
: restores archives written by *vxdump* (...)

Generally used command options

-F *FSType*

Specifies the file system typeonly necessary, if the type can't be detected automatically via */etc/vfstab*

-r *raw_device*

Pathname of the raw device to useShould be used when the mountpoint or *fsck* device for the mountpoint is not specified in */etc/vfstab*.

The generic command options are not described in this manual. Refer to the man pages for information about them.

8.1 getext(1)

getext displays extent attribute information for given files on a VxFS file system.

Synopsis

getext [**-f**] [**-s**]file ...

Command Options

-f Suppresses the printing of the file name at the beginning of the list of extent attributes. This makes parsing of the output easier.

-s Suppresses the printing of output for files which do not have any extent attribute information associated with them. When the command is issued against a group of files, some of which have extent information and some of which do not, just those files with extent information will produce output.

Output

The output format is

filename: Bsize num□□□Reserve num□□□Extent Size num□□□flags

with the following meaning

Bsize

is the block size of the file system on which *filename* is located

Reserve

is the amount of space reserved for the file in numbers of blocks

Extent Size

is the number of blocks in extents that will be allocated to the file if it is a fixed extent file.

flags

flags set with *setext*(1).

8.2 setext(1)

setext provides a command line interface for manipulating extent attribute information for files in a VxFS file system. The file to be operated on must exist before issuing the *setext* command – *setext* does not create files.

Using the *setext* command, users can do preallocation of space for files, including persistent and nonpersistent allocation, set a fixed extent size for allocations, request aligned or contiguous allocation, or quickly create large temporary files.

Synopsis

setext [**-e** extent_size][**-r** reservation][**-f** flag] file

Command Options

-e *extent_size*

Sets a fixed extent size of *extent_size* file system blocks. Each subsequent allocation to the file will be made in increments of *extent_size* blocks unless the file has already gone into indirect extents, in which case the extent size is already fixed.

threshold values: 0<=*extent_size*<=155648

-r *reservation*

Preallocates space to the file of *reservation* file system blocks.

threshold values: 0<=*reservation* <=911784

-f *flag*

Specifies characteristics of allocation, where *flag* is one of the following:

align

each extent must be aligned to *extent_size* boundaries.

contig

reservation must be allocated contiguously.

noextend

indicates no allocation should be made to the file once the current *reservation* blocks have been exhausted.

trim

specifies that the file is to be trimmed to the current file size upon last close by all processes that have the file opened.

chgsize

specifies that the reservation is to be immendiately incorporated into the file. *chgsize* is restricted to the superuser.

noreserve

The reservation is to be made as a nonpersistent allocation to the file. The on-disk inode will not updated with the reservation information so that the reservation will not survive a system crash. The reservation is associated with the file until the close of the file. The reservation is trimmed to the current file size on close.

Both the *-e* and *-f* options take option arguments that are expressed in units of file system blocks. If the the file system block size is unknown, it can be determined by first doing a *getext* on the file. (See the *getext* manual page, or documentation in Section "getext(1)" for further information.)

Multiple *flag* options can be specified by using more than one *-f* option on the command line. As noted all of the *flag* options are tied to either the *-e* or *-r* option argument. The command will fail if, for example, the *-f align* flag is selected and no *-e* is specified.

8.3 df(1)

The *df* command displays the amount of free space and the number of free files in a file system. *df*(1) is applicable to mounted and unmounted file systems and also to file systems which are mounted via NFS. An option is provided to display the total number of blocks and files in the file system. Options are provided for several different print formats.

Synopsis

df [**-F vxfs**] [**generic_options**] [**-o s**][directory | special]

Command options

For mounted file systems, either the *directory* or *special* device may be specified; for unmounted file systems, use the *special*.The generic version of *df* handles all mounted file systems unless the *-o* option is specified. For mounted file systems, the generic *df* issues a *statvfs*(2) call against the mountpoint in order to gather resource statistics.

-b Prints the free disk space, in Kbyte

-e Prints the number of free inodes

-k Prints the following information in columns for all mounted file systems or for the specified file systems:

filesystem
: Special file associated with the file system

kbyte
: Total disk space, in Kbyte

used
: Used disk space, in Kbyte

avail
: Available disk space, in Kbyte

capacity
: Percentage of disk space used

mounted on
: Name of the directory on which the file system is mounted

-l *df* prints the number of free blocks and inodes for all locally mounted file systems or for the specified file systems. The *-l* option is set by default.

-n *df* prints the file system type name.

-P Generates a portable format that can be incorporated into automatic procedures ((e.g. shell scripts). The following information is printed in columns for all mounted file systems or for the specified file systems

-t *df* prints the number of free blocks and files as well as the total number of available blocks and files for each mounted or specified file system.

-o s This option displays the count of available extents by size. The list will be displayed as four extents per line, with a label that indicates the size of extent, followed by the number of available extents of that size. If there are a number of small extents (less than size 64), but few large extents, the system is a candidate for extent reorganization. For details on extent reorganization, see "fsadm(1M)".

Because of different print formats not all of the options can be combined. The following options can be usefully combined:

- *-b*, *-e*, *-n*
- *-k* , *-P*

All the other options should not be used in any combination.

Example

The command *df -t /lhome* produces the following output:

```
/lhome□□□□(/dev/ios0/sdisk011s5 ):  64110 blocks□□□□□30568 files
□□□□□□□□□□□□□□□□□□□□□□□□□□□□total: 311472 blocks□□□□□38880 files
```

There is an example for *df -o s* in section Determining fragmentation.

8.4 ff(1M)

The *ff*(1M) command generates a list of pathnames for files that match specified criteria. The *ff* utility can be used for inode versus pathname cross-reference checks in a file system. Used without selection criteria, the command will print an exhaustive list of inodes with their associated file names. Selection criteria can be chosen which will find the inode/pathname pairs by:

- inode
- the times that the inode has been accessed, changed, or modified

- whether the inode represents a block/character special file or has its set user id or set group id mode bits on

Synopsis

ff [**-F vxfs**] [**generic_options**]] [**-o** s] special ...

Command options

-o *s* Only special files and files with set-user-ID mode bit on will be selected.

-I The *-I* option suppresses the printing of the inode number after each pathname.

-l Default behavior of *ff* is only to show one pathname for a given linked file. With the *-l* option specified, all pathnames associated with a given inode are printed.

-p *prefix*
The specified prefix will be added to each generated path name (the default is ".").

-s Print the file size in bytes after each pathname.

-u Print the owner's login name after each pathname.

-a *n* Select if the inode has been accessed in *n* days.

-m *n* Select if the inode has been modified in *n* days.□
(i. e. a modification of the **file content**, refer to *-c*)

-c *n* Select if the inode status has been changed in *n* days.□
(i. e. a modification of the **inode itself**, refer to *-m*)

-n *file*
Select if the inode has been modified more recently than the argument file.

-i *i_node_list*
Generate names for only those inodes specified in *i_node_list*. The option *i_node_list* is a list of numbers separated by commas and without spaces.

ff is useful for tracking down file system problems and potential security violations. For example, if a structural *fsck* of a file system uncovers problems with a specific inode, the pathname can be discovered through the use of *ff*.

Printing is done one inode/pathname pair to a line, in inode order. Each inode/pathname pair is tab-separated on the line with the pathname preceding the inode. When not supplied with a specific list of special devices to check, the generic utility will attempt to generate the cross-reference list for each special device in the */etc/vfstab* for which the *fsckpass* is numeric.

When provided with a list of special files, the generic *ff* will use the *FSType* provided as the option argument to *-F* as the indicator of which file system specific command to invoke. If no *-F* is specified, the generic command will attempt to determine the *FSType* of the particular device from the */etc/vfstab*. Once the appropriate type-specific version of *ff* is determined, *ff* is run against the named device.

The VxFS-specific *ff* command will assure that the specified device file is truly a VxFS file system by checking its *VX_MAGIC*, *VX_CHECKSUM* and *VX_VERSION* in the superblock. The format of the superblock is specified in the manual page for *fs*(4)

The two utilities *ff* and *ncheck* overlap in function; however, there are specific capabilities that distinguish each. For comparison, see the manual page for *ncheck*(1M) for finding pathname based on block number.

8.5 fsadm(1M)

The *fsadm*(1M) command provides on-line administration facilities.

Synopsis

The basic forms of the command line are

- for resizing the file system (...):

  ```
  fsadm [ -F vxfs ][ -b new_size ][ -r raw_dev ]mount_point
  ```

- for a report about extent fragmentation (...):

  ```
  fsadm [ -F vxfs ][ -E ] [ -l large_size ]mount_point
  ```

- for a report on directory fragmentation (...):

  ```
  fsadm [ -F vxfs ] [ -D  ]mount_point
  ```

- for **reorganizing extents** (...):

  ```
  fsadm [ -F vxfs ] [ -e ] [ -E ][ -s ] [ -v ] [ -p passes ]] [ -t time ] [ -a days ]\
  [ -l large_size ] [ -r raw_dev ]mount_point
  ```

- for reorganizing directories (...):

  ```
  fsadm  [ -F vxfs ] [ -d ] [ -D ][ -s ][  -v ] [ -p passes ]] [ -t time ] [ -a days ]\
  [ -r raw_dev ]mount_point
  ```

The *fsadm*(1M) command can only be run on mounted file systems. Many of the functions are implemented using VxFS-specific *ioctl* functions that are applied to the root inode of the file system. However, *fsadm* also accesses the disk directly through the raw disk device. If the mountpoint has an "fsck device", then *fsadm* uses the *fsck* device specification as the raw device to use for the mountpoint. If an entry for the file system is not in */etc/vfstab*, then *fsadm* attempts to determine the raw device corresponding to the block device by using certain heuristics. If this also fails then the raw device must be specified on the command line using the *-r raw_device* option.

8.5.1 Resizing a file system

fsadm provides for either increasing or decreasing the size of a file system while it is mounted. If a file system is full, it can be expanded while being accessed, provided that there is unused disk space after the end of the file system. If the file system is mounted on a virtual disk, the disk itself can be expanded to provide space. Refer to the manual "Virtual Disks, System Administrator Guide".

Expanding a file system is easy, but the system administrator is responsible for ensuring that the disk space is truly free. The *fsadm* utility has no way of telling if space is being used elsewhere.

Example: Expanding of a virtual disk

With the following commands a virtual disk can be expanded:

```
# umount /mnt
# dkconfig -uv /dev/vd/vdisk1
□□□□□□□#add a new partition to vdisk1 in /etc/dktab
# dkconfig -cv /dev/vd/vdisk1
# mount /mnt
# fsadm -b newsize /mnt
```

If a file system has been allocated more disk space than is used on a regular basis, it can be shrunk so that the space can be reclaimed for use elsewhere. The only caveat for shrinking a file system is that the space to be taken out of the file system must not be currently in use. If the space is used *fsadm* will print an indication of this fact and the resize operation will be failed. File system reorganization may be needed in this case to free up the resources that prevent shrinking.

Synopsis for resizing a file system

fsadm [**-F** vxfs] [**-b** new_size][**-r** raw_dev]mount_point

Command options

-b *new_size*

Resize the file system to *new_size* sectors. *new_size* is specified in sectors, as when building a file system using *mkfs*(1M).

A VxFS file system consists of one or more allocation units. Each allocation unit has the same capacity, but the last allocation unit on a file system may have a partial set of blocks. To "grow" a filesystem, the last allocation unit may need to be expanded or new allocation units may need to be added, depending on the size of the last allocation unit and the number of additional blocks requested. Likewise to "shrink" a filesystem, the last allocation unit may need to be reduced or allocation units may need to be taken out of the file system.

The following steps are taken to resize a file system:

1. The following superblock fields in the main superblock, and the copy of the superblock in each allocation unit are changed to reflect:
   - *fs_size* – the new number of sectors in the file system
   - *fs_dsize* – the new number of data blocks in the file system
   - *fs_nau* – the number of allocation units in the file system
2. Any new allocation units are built.
3. For file system expansion, if the last allocation unit in the file system had a partial set of data blocks the *emap* (extent map) for the allocation unit is updated to reflect that the newly expanded blocks are available to be allocated.
4. If the last allocation unit created by shrinking the file system has a partial set of data blocks the *emap* (extent map) for the allocation unit is updated to reflect the fact that the blocks beyond the new last block are unavailable for allocation.
5. The resource counts in the main superblock are updated as follows:
   - *fs_free* – the number of free data blocks in the file system
   - *fs_ifree* – the count of free inodes
   - *fs_efree* – the number of free extents by size

8.5.2 Report on extent fragmentation

Over time as files are created and removed, the free extent list for an allocation unit will change from having one large free area to having many smaller free areas. This process is known as **fragmentation**. Also, when files are grown – particularly when growth occurs in small increments – a small file could be allocated in multiple extents. In the ideal case, each file that is not sparse would have exactly one extent (containing the entire file), and the free extent list would be one continuous range of free blocks.

Conversely, in a case of extreme fragmentation, there can be free space in the file system, none of which can be allocated. For example, for indirect extents, the indirect address extent is always 8 Kbyte. This means that to allocate an indirect address extent to a file, an 8 Kbyte extent must be available. Is no 8 Kbyte extent available but many extents with less than 8 Kbyte, it is not possible to add a file larger than 8 Kbyte.

Determining fragmentation

The first step in determining fragmentation is to run the *df* utility against the file system to get a list of free extents by size. For example:

```
# df -os /lhome
```

This command produces output similar to the following:

```
(/dev/ios0/sdisk011s5 ):□□□122732 blocks□□□□37154 files
Free Extents by Size
```

```
□□□1:□□□□8□□□□□□□□□□2:□□□11□□□□□□□□□□4:□□□10□□□□□□□□□□8:□□□□6
□□16:□□□10□□□□□□□□□32:□□□□9□□□□□□□□□64:□□□10□□□□□□□□128:□□□□4
□256:□□□□9□□□□□□□□512:□□□□4□□□□□□□1024:□□□□8□□□□□□□2048:□□□□5
4096:□□□□3□□□□□□□8192:□□□□1□□□□□□16384:□□□□1
```

Although the term in the output is "blocks", in this case it is used for "sectors".

If there is a large number of small extents free, then there is fragmentation. If more than one-half of the amount of free space is taken up by small extents (smaller than the indirect data extent size), or there is less than five percent of free space available in large extents, then there is serious fragmentation.

In the preceding example, there are eight free extents of size 1, 11 of size 2, 10 of size 4, six of size 8, 10 of size 16, and nine of size 32. This totals 1132 sectors, which is only a small fraction of the total free count of 122732 sectors. This file system has an extent reorganization performed on it daily, which keeps fragmentation minimal.

Running the extent fragmentation report

Once it is determined that a file system has a degree of fragmentation, the extent fragmentation report can be run to acquire more detailed information.

Synopsis for an extent fragmentation report

fsadm [**-F** vxfs] [**-E**] [**-l** large_size]mount_point

Command options

-E Report on extent fragmentation

-l *large_size*
Large file size in file system blocks

The extent reorganizer has a concept of an immovable extent. This means that if the file already contains large extents, reallocating and consolidating these extents will not improve performance. By default, the *large_size* is 64 blocks, meaning that any extent larger than 64 blocks is considered to be immovable.

For the purposes of the extent fragmentation report, the *large_size* chosen will affect which extents are reported as being immovable extents.

Example of a report on extent fragmentation

The following is the output of the command *fsadm -E /lhome*

```
Extent Fragmentation Report □
Files with          Total        Total □
Extents    Extents    Blocks    Distance □
au   0     3335       3633      58344      3626205 □
au   1     2611       2786      24390      1662696 □
au   2      535        570       7020       894501 □
au   3     1211       1332      22107       824842 □
au   4     1491       1622      18134       745327 □
total      9183       9943     129995            0□
□
Consolidatable      Immovable □
Extents    Blocks    Extents    Blocks □
au   0        0         0         35        6401 □
au   1        0         0         55       12966 □
au   2       20        90         24        6184 □
au   3        5        42         42        6307 □
au   4      111      1379         47        5384 □
total        36      1511        203       37242 □
□
Free Extents By Size □
au   0    Free Blocks 217,   Smaller Than 8 - 48%,    Smaller Than 64 - 100% □
1:      15     2:        15     4:         15     8:      14 □
16:      0     32:        0     64:         0    128:      0 □
256:      0   512:        0   1024:         0   2048:      0 □
4096:      0  8192:        0  16384:         0 □
au   1    Free Blocks 286,   Smaller Than 8 - 41%,    Smaller Than 64 - 100% □
1:      16     2:        21     4:         15     8:      13 □
16:      4     32:        0     64:         0    128:      0 □
256:      0   512:        0   1024:         0   2048:      0 □
4096:      0  8192:        0  16384:         0 □
au   2    Free Blocks 510,   Smaller Than 8 - 15%,    Smaller Than 64 - 100% □
1:      10     2:        14     4:         10     8:      14 □
16:      8    32:         6     64:         0    128:      0 □
256:      0   512:        0   1024:         0   2048:      0 □
4096:      0  8192:        0  16384:         0 □
au   3    Free Blocks 6235,  Smaller Than 8 - 3%,     Smaller Than 64 - 15% □
1:      29     2:        33     4:         27     8:      30 □
16:     18    32:         8     64:         4    128:      3 □
256:      2   512:        2   1024:         1   2048:      1 □
4096:      0  8192:        0  16384:         0 □
au   4    Free Blocks 8551,  Smaller Than 8 - 2%,     Smaller Than 64 - 22% □
1:      29     2:        33     4:         30     8:      38 □
16:     28    32:        29     64:        26    128:     11 □
256:      8   512:        3   1024:         0   2048:      0 □
4096:     0   8192:       0  16384:         0 □
total     Free Blocks 15799, Smaller Than 8 - 4%,     Smaller Than 64 - 24% □
1:      99     2:       116     4:         97     8:     109 □
16:     58    32:        43     64:        30    128:     14 □
256:     10   512:        5   1024:         1   2048:      1 □
4096:      0  8192:        0  16384:         0 □
```

The numbers in the column labeled *Files with Extents* contains the total number of files which have data extents. A file is considered to be in the allocation unit where its inode is. As there are *fs_inopau* inodes per allocation unit, you can easily tell from the inode number to which allocation unit a file belongs.

The column labeled *Total Extents* contains the total number of extents belonging to files in the allocation unit. The extents themselves are not necessarily in the same allocation unit.

The column labeled *Total Blocks* contains the total number of blocks used by files in the allocation unit. If the total number of blocks is divided by the total number of extents, the resulting figure is the average extent size.

The column labeled *Total Distance* contains the total distance between extents in the allocation unit. For example, if a file has two extents, the first containing blocks 100 through 107 and the second containing blocks 110 through 120, the distance between the extents is 110 **minus** 107 , or three. In general, a lower number means that files are more contiguous. If an extent reorganization is run on a fragmented file system, the value for Total Distance should be reduced.

The column labeled *Consolidatable Extents* contains the number of extents which are candidates to be consolidated. *Consolidation* means merging two or more extents into one combined extent. For files that are entirely in direct extents, the extent reorganizer will attempt to consolidate extents into extents up to size *large_size*. Since many files are small, in general all files of size *large_size* or less will be contiguous in one extent after reorganization.

The column labeled *Consolidatable Blocks* contains the total number of blocks in Consolidatable Extents.

The column labeled *Immovable Extents* contains the total number of extents which are considered to be immovable. In the report, an immovable extent appears in the allocation unit of the extent itself, as opposed to in the allocation unit of its inode. This is because the extent is considered to be immovable, and thus permanently fixed in the associated allocation unit.

The column labeled *Immovable Blocks* contains the total number of blocks in immovable extents.

The figures under the heading *Free Extents By Size* indicate per allocation unit totals for free extents of each size. The totals are for free extents of size 1, 2, 4□... up to a maximum of the number of data blocks in an allocation unit. The totals should match the output of *df -os* unless there has been recent allocation or deallocation activity (as this utility acts on mounted file systems). These figures give an indication of fragmentation and extent availability on a per allocation unit basis.

For each allocation unit, and for the complete file system, the total free blocks and total free blocks by category is shown.

The figure labeled *Free Blocks* indicates the total number of free blocks.

The figure labeled *Smaller Than 8* indicates the percentage of free blocks that are in holes of less than 8 blocks in length.

The figure labeled *Smaller Than 64* indicates the percentage of free blocks that are in holes of less than 64 blocks in length.

In the preceding example, 4% of free space is in holes less than 8 blocks in length, and 24% of the free space is in holes less than 64 blocks in length. This represents a typical value for a mature file system that is regularly reorganized.

Visible results of reorganization

If the extent reorganizer is run on a fragmented file system with a fragmentation report taken before and after the reorganization, the following changes should be noticeable:

- Total Extents should be reduced.
- Total Distance should be reduced.
- Consolidatable Extents should be reduced.
- The total number of free extents should be reduced by small free extents getting consolidated into larger extents.

8.5.3 Report on directory fragmentation

As files are allocated and freed, directories tend to grow and become sparse. In general, a directory is as large as the largest number of files it ever contained, even if some files have been subsequently removed.

Synopsis for a report on directory fragmentation

fsadm [**-F** vxfs] **-D** mount_point

Command options

-D Report on directory fragmentation

Example of a report on directory fragmentation

The following is the output from the command *fsadm -D /lhome*

```
Directory Fragmentation Report □
Dirs       Total    Immed   Immeds    Dirs to    Blocks to □
Searched   Blocks    Dirs   to Add    Reduce      Reduce □
au    0      743     168     584        0      0        0 □
au    1      396     136     273        0      0        0 □
au    2      111      39      72        0      0        0 □
au    3      174      84      91        0      0        0 □
au    4      153      67      95        0      0        0 □
total       1577     494    1115        0      0        0 □
```

The column labeled *Dirs Searched* contains the total number of directories. A directory is associated with an allocation unit in which its inode is located.

The column labeled *Total Blocks* contains the total number of blocks used by directory extents.

The column labeled *Immed Dirs* contains the number of directories that are immediate, meaning that the directory data is in the inode itself, as opposed to being in an extent. Immediate directories save space and speed up pathname resolution.

The column labeled *Immeds to Add* contains the number of directories which currently have a data extent, but which could be reduced in size and contained entirely in the inode.

The column labeled *Dirs to Reduce* contains the number of directories for which one or more blocks could be freed if the entries in the directory are compressed. Since directory entries are variable in length, it is possible that some large directories may contain a block or more of total free space, but the entries are arranged in such a way that the space cannot be freed. As a result, it is possible to have a non-zero *Dirs to Reduce* calculation immediately after running a directory reorganization. The *-v* (verbose) option of directory reorganization reports occurrences of failure to compress free space.

The column labeled *Blocks to Reduce* contains the number of blocks which could be freed if the entries in the directory are compressed.

Measuring directory fragmentation

If the totals in the columns labeled *Dirs to Reduce* or *Blocks to Reduce* are substantial, a directory reorganization should improve performance of pathname resolution. The directories that fragment tend to be the directories with the most activity. A small number of fragmented directories may account for a large percentage of name lookups in the file system.

Periodic reorganization is recommended. If there is a daily or weekly time of reduced activity, run directory reorganization during the period of reduced activity.

8.5.4 Extent reorganization

The extent reorganizer attempts to arrange the extents for files on an allocation by allocation unit basis as follows:

Inode List
Extents for directories and symbolic linksSize 1□ Size 2□ Size 3□ .□ .□ Size *large_size*
Extents for recently accessed filesSize 1□ Size 2□ Size 3□ .□ .□ Size *large_size*
Extents for aged and large filesSize 1□ Size 2□ Size 3□ .□ .□ Size *large_size*
Free Extent area
Extents for files from other allocation units
End of allocation unit

Directories and symbolic links are placed close to the inode list. Next are recently accessed files. Aged and large files are at the end of the allocation unit. The free area follows allocated units. Extents allocated to files from other allocation units are placed at the end of the allocation unit.

The algorithm for reorganizing extents is simple in theory. Take extents that are in the wrong place, consolidate them into extents of up to size *large_size*, and move them to the proper location. There are a number of complications which make reorganizing mounted file systems difficult. First, allocation and deallocation of files and extents can take place during the reorganization. Second, in order to move an extent to the proper location, any extents currently occupying the desired location need to be moved.

The extent reorganizer reorganizes one allocation unit at a time starting with the first allocation unit. The process is as follows:

1. Run through the file system deciding the proper location for each file and extent.
2. Run through the allocation unit being reorganized. If an extent can be moved to the proper location, move it there. If an extent cannot be moved to the proper location, but is in the proper location for another extent in the allocation unit being reorganized, move the extent to any convenient free location.
3. Run through all subsequent allocation units. If an extent can be moved into the proper location, move it there. If an extent is in the area designated for the allocation unit being reorganized but cannot be moved to its proper location, move the extent out this area if free space can be found.
4. Once the allocation unit being reorganized has all extents in the proper location, proceed with the next allocation unit

One iteration through the preceding process is considered to be one pass of the reorganizer.

The preceding algorithm breaks down when file systems are more than 90 percent full, since the amount of free space is too little to allow extents to be moved around. For this reason, it is suggested that VxFS file systems should never exceed 90 percent of capacity.

In the preceding algorithm, free space is used to facilitate moving extents to desired locations. It is possible that there is insufficient free space to make the necessary moves. When a file system reaches this degree of fullness, the extent reorganizer switches focus from attempting to provide contiguity and optimal locality, to attempting to minimize fragmentation of free space. To minimize fragmentation, the final pass of the extent reorganizer utilizes a different algorithm, if required. The secondary algorithm pushes nonadjacent extents together, until a sufficiently large hole has been built up.

Synopsis for extent reorganization

fsadm [**-F** vxfs] [**-e**] [**-E**] [**-s**] [**-v**][**-p** passes] [**-t** time] [**-a** days] \

[**-l** large_size] [**-r** raw_dev]mount_point

Command Options

-e extent reorganisation

-s Prints a summary of activity for each pass. The example in this section illustrates the summary output.

-v Provides a detailed record of activity during each pass. The output can be lengthy.

-p *passes*

Specifies the number of passes to run. The default is currently five. If the reorganization takes less than the maximum number of passes, it will exit when complete.

-t *time*

Specifies the maximum time to run (in seconds). If your system has a scheduled period of time for administrative activity, or general periods of inactivity, an extent reorganization can be started at a designated time and with a specific time limit. This feature is particularly useful when running reorganization from *cron*(1M). The default is zero (no time limit).

-a *days*

Specifies the number of days since last access after which a file is considered to be aged. The default is 14. The theory of aging files is that aged files are moved to less desirable locations, since if a file has not been recently accessed, it is not likely to be accessed in the near future.

-l *largesize*

Specifies the size at which files and extents are considered to be large. The default is 64 blocks. This size should not need to be changed.

Example of the output of extent reorganization

The following is the output from the command

```
# /etc/fs/vxfs/fsadm -e -s /lhome
Allocation Unit 0, Pass 1 Statistics □
Extents     Consolidations Performed       Total Errors □
Searched    Number    Extents    Blocks  File Busy   Not Free □
au    0        2467       11        30        310          0          0 □
au    1           0        0         0          0          0          0 □
au    2           0        0         0          0          0          0 □
au    3           0        0         0          0          0          0 □
au    4           0        0         0          0          0          0 □
total          2467       11        30        310          0          0□
□
In Proper Location   Moved to Proper Location □
Extents    Blocks   Extents     Blocks □
au    0       1379      8484       794       10925 □
au    1          0         0         0           0 □
au    2          0         0         0           0 □
au    3          0         0         0           0 □
au    4          0         0         0           0 □
total         1379      8484       794       10925□
□
Moved to Free Area       In Free Area       Could not be Moved □
Extents     Blocks    Extents    Blocks     Extents     Blocks □
au    0        231      4851          4       133          0          0 □
au    1          0         0          0         0          0          0 □
au    2          0         0          0         0          0          0 □
au    3          0         0          0         0          0          0 □
au    4          0         0          0         0          0          0 □
total          231      4851          4       133          0          0 □
□
Allocation Unit 0, Pass 2 Statistics □
Extents     Consolidations Performed       Total Errors □
Searched   Number    Extents     Blocks  File Busy   Not Free □
au    0        2467        0         0          0          0          0 □
au    1           0        0         0          0          0          0 □
au    2           0        0         0          0          0          0 □
au    3           0        0         0          0          0          0 □
au    4           0        0         0          0          0          0 □
total          2467        0         0          0          0          0□
□
□
In Proper Location    Moved to Proper Location □
Extents     Blocks    Extents     Blocks □
au    0       2173     19409       235        4984 □
au    1          0         0         0           0 □
au    2          0         0         0           0 □
au    3          0         0         0           0 □
au    4          0         0         0           0 □
total         2173     19409       235        4984 □
□
Moved to Free Area       In Free Area        Could not be Moved □
Extents    Blocks    Extents     Blocks     Extents     Blocks □
au    0          0         0          0         0          0          0 □
au    1          0         0          0         0          0          0 □
au    2          0         0          0         0          0          0 □
au    3          0         0          0         0          0          0 □
au    4          0         0          0         0          0          0 □
total            0         0          0         0          0          0 □
```

Note that the default five passes were scheduled, but the reorganization finished in two passes.

The file system had not had much activity since the last reorganization, with the result that little reorganization was required. The time it takes to complete extent reorganization is highly variable depending on fragmentation and disk speeds. However, in general, extent reorganization may be expected to take approximately one minute for every 10 Mbyte of disk space used.

In the preceding example, the column labeled *Extents Searched* contains the total number of extents searched.

The column labeled *Number* located under the heading *Consolidations Performed* contains the total number of consolidations or merging of extents performed.

The column labeled *Extents* under heading *Consolidations Performed* contains the total number of extents that were consolidated. The column labeled *Blocks* located under the heading *Consolidations Performed* contains the total number of blocks that were consolidated.

The column labeled *File Busy* located under the heading *Total Errors* contains the total number of reorganization requests that failed because the file was active during reorganization.

The column labeled *Not Free* located under the heading *Total Errors* contains the total number of reorganization requests that failed because an extent which the reorganizer expected to be free was allocated at some time during the reorganization.

The column labeled *In Proper Location* contains the total extents and blocks that were already in the proper location at the start of the pass. The column labeled *Moved to Proper Location* contains the total extents and blocks that were moved to the proper location during the pass.

The column labeled *Moved to Free Area* contains the total number of extents and blocks that were moved into a convenient free area, in order to free up space which has been designated as the proper location for an extent in the allocation unit being reorganized. The column labeled *In Free Area* contains the total number of extents and blocks that were in areas designated as free areas at the beginning of the pass.

The column labeled *Could not be Moved* contains the total number of extents and blocks that were in an undesirable location and could not be moved. This occurs when there is not enough free space to allow sufficient extent movement to take place. This often occurs on the first few passes for an allocation unit if a large amount of reorganization needs to be performed.

If the next to last pass of the reorganization run indicates extents which could not be moved, then the reorganization failed. A failed reorganization may leave the file system badly fragmented, since free areas are used when trying to free up reserved locations. To lessen this fragmentation, extents are not moved into the free areas on the final two passes of the extent reorganizer, and the last pass of the extent reorganizer only consolidates free space. The system administrator should monitor free space counts or add space using *fsadm -b* to ensure that sufficient space is preserved (...).

8.5.5 Directory reorganization

The directory reorganizer removes unused space from directories and reorganizes the entries in directories in order to link subdirectories and symbolic first, followed by files arranged by time of last access. The command line to reorganize directories of a file system is:

Synopsis for directory reorganization

fsadm [**-F** vxfs] [**-d**] [**-D**][**-s**][**-v**] [**-p** passes]] [**-t** time] [**-a** days]\

[**-r** raw_dev]mount_point

Command options

-d Reorganize directories

-D Reports on directory fragmentation. If specified in conjunction with the □
-d option, the fragmentation report is produced both before and after the directory reorganization.

-s Prints a summary of the activity of each pass. The example in this section shows the summary output.

-v (verbose) Provides a detailed record of the activity during each pass; output can be lengthy.

-p *passes*
Maximum number of passes to run.

-t *time*

Specifies a maximum time (in seconds) for the reorganization to run. By default, the maximum time is zero, which means unlimited. Directory reorganization will likely take approximately one minute per pass for every 500 directories on the file system, if run on a quiet file system. The example in this section took 27 seconds to run.

-a *days*

Specifies the number of days since last access after which a file is considered to be aged. The default is 14. The theory of aging files is that aged files are moved to less desirable locations, since if a file has not been recently accessed, it is not likely to be accessed in the near future.

Example of the output of directory reorganization

The following is the output from the command

```
# /etc/fs/vxfs/fsadm -d -s /lhome
```

```
Pass 1 Statistics □
Dirs     Dirs      Total    Failed  Blocks   Blocks     Immeds □
Searched  Changed   Ioctls   Ioctls  Reduced  Changed    Added □
au   0      134      1       1       0       1       0      0□
au   1        0      0       0       0       0       0      0□
au   2        0      0       0       0       0       0      0 □
au   3        0      0       0       0       0       0      0 □
au   4        0      0       0       0       0       0      0 □
total       134      1       1       0       1       0      0 □
```

The column labeled *Dirs Searched* contains the number of directories searched. Only directories with data extents are reorganized. Immediate directories are skipped. The column labeled *Dirs Changed* contains the number of directories for which a change was made.

The column labeled *Total Ioctls* contains the total number of *VX_DIRSORT ioctls* performed (see the manual page for *vxfsio*(7) for details). Reorganization of directory extents is performed using this *ioctl*.

The column labeled *Failed Ioctls* contains the number of requests that failed for some reason. The reason for failure is usually that the directory being reorganized is active. A few failures should be no cause for alarm. If the *-v* option is used, all *ioctl* calls and status returns are recorded.

The column labeled *Blocks Reduced* contains the total number of directory blocks freed by compressing entries. The column labeled *Blocks Changed* contains the total number of directory blocks updated while sorting and compressing entries.

The column labeled *Immeds Added* contains the total number of directories with data extents that were able to be compressed into being immediate directories.

8.6 fscat(1M)

The *fscat* utility provides an interface to a VxFS file systems similar to that provided by the *cat*(1) or *dd*(1) utilities invoked on the special file of a VxFS file system. On most VxFS file systems, the block or character special file for the file system provides access to a raw image of the file system for purposes such as backing up the file system to tape; on a snapshot file system access to the corresponding block or character special provides little useful information.

When invoked on the block or character special file of a VxFS file system, *fscat* provides a stream of bytes representing the "raw image" of the file system. This stream of bytes can be processed in a pipeline, written to a tape, etc. If *fscat* is invoked on the block or character special file of a snapshot file system, it produces the raw image of the file system snapped at the moment the snapshot was made.

By default, the output of *fscat* is a stream of bytes that starts at the beginning of the file system and goes to the end. On most file systems, the data is read from the special file using *read*(2) but on a snapshot file system, data is read from the file system using the *VX_SNAPREAD ioctl*(2) on the mountpoint. Data is read in *VX_MAXIO* chunks. Unless otherwise specified, data is written in blocks of size *st_blksize* as returned by a *fstat*(2) on *stdout*.

Synopsis

fscat [**-F** vxfs] [**-o** offset] [**-l** length] [**-b** block_size] special

Command options

-o *offset*

Specify the starting *offset*, in bytes of the output. This is interpreted as a seek to the specified offset, at which point dumping starts. If multiple file systems are specified, this starting *offset* is used for all of them. If the *-o* option is not used, offset defaults to 0.

-l *length*

Specify the length, in bytes to output. Output starts at the offset specified (or 0, in none is specified by the *-o* option) and continues for *length* bytes. A *length* of 0 goes to the end of the file system, and is the default. If multiple file systems are specified, this starting offset is used for all of them.

-b *block_size*

Each write is *block_size* bytes. *block_size* must be less than or equal to 1 megabyte. The last write for a file system may be less than *block_size*. If more than one file system is specified, the last write for each file system may be less then *block_size*. The default *block_size* depends on the file being written to and is based on the value returned by *statvfs*(2) for *f_bsize*, generally 5 Kbyte when the output is a pipeline.

All numbers entered as options may have "0" prepended to indicate octal, or "0x" prepended to indicate hexadecimal. A "b" may be appended to indicate the value is in 512-byte blocks, a "k" to indicate the value is in kilobyte, or an "m" to indicate the value is in megabytes. The number must always be followed by the appended letter without any space.

8.7 fsck(1M)

The VxFS-specific *fsck*(1M) command checks file systems for consistency.

Synopsis

The basic forms of the command line are:

fsck [**-F** vxfs] **-m** special

fsck [**-F** vxfs] **-n** special

fsck [**-F** vxfs] [**-o nolog, full**][**-y**]special

Command options

-m Check for mountability. To determine if a file system can be mounted, the *-m* option is specified.

-n This option runs a full file system check. Specifying the *-n* option will generate a report on errors but will not make any repairs in conjunction with those errors. Intent log replay is not performed. This form of the command can be run safely on mounted file systems on which there may have been damage. If file system damage is suspected, a full *fsck* should be run to determine the extent of file system damage.

-N Synonym for -n.

-y Answer all questions yes. This option has two ramifications. First, after an intent log replay is run, if the file system is not in the *CLEAN* state, a full file system check will be initiated automatically. Second, yes will be automatically answered to all questions posed by the full file system check.

-Y Synonym for -y.

-o *nolog*

Inhibit log replay. An intent log replay is not performed. This option can be used to disable log replay, if the intent log is physically damaged.

-o *full*

Force full file system check. By default, only a log replay is run.

-l Synonym for *-o*

A full file system check is normally interactive, meaning that the system administrator is prompted before any corrective actions are taken. This is not true if the *-y* or *-n* options are used. When these options are used, *fsck* automatically answers "yes" or "no" to prompts, instead of waiting for input.

Invocation from system startup

When the *mountall*(1M) script runs to mount file systems, an *fsck* with the *-m* option is run on each file system found in */etc/vfstab*. This process is called sanity checking (see Section "Sanity check"). For all file systems that are reported as requiring a file system check, mountall runs an *fsck* with the *-y* option, before attempting to mount the file system.

Intent log replay versus full file system check

By default, the VxFS-specific *fsck* only runs an intent log replay, and does not perform a full file system check. The full file system check is provided to handle the case where a file system is damaged due to I/O failure.

The VERITAS V2.1 intent log format with Reliant UNIX 5.45 differs from the one of former versions (Section "Fast file system recovery"). The command *fsck* recognizes the version and accordingly reports the version of the intent log:

vxfs fsck: *device*: vxfs1 log format

resp.

vxfs fsck: *device*: vxfs2 log format

Sample command line

The following command line shows *fsck* being entered to check a file system that was active when the system failed. The system response is the messages generated by a successful intent log replay.

```
# fsck -F vxfs /dev/ios0/rsdisk001s2
log replay in progress
replay complete - marking superblock as CLEAN
```

The running of *fsck* can be separated into two different phases:

1. Initiation phase, containing
   - superblock verification

- sanity check
- intent log replay

2. Full file system check phase,
   consisting of four passes (see Section "Full file system check phase")

The messages pertinent to each phase of *fsck* you can find in Section "Messages of fsck(1M)".

8.7.1 Initiation phase

The initiation phase of *fsck* interprets the command line and decides which functions to perform. The operations of this phase are as follows:

1. Superblock verification is initiated to determine if the file system is a valid VxFS file system. If the file system is not a valid VxFS file system, *fsck* prints an explanatory message and then exits.
2. If the command line specifies Sanity Check (*-m* option), the Sanity Check is initiated. Sanity Check exits when complete.
3. The state of the file system is checked.
4. Intent log replay is initiated unless one of the following is true:
   - The file system state is CLEAN.
   - The *-n* option was specified.
   - The superblock indicates that the intent log is damaged.
   - The *-o nolog* option was specified.
   - The file system was not a logging file system.
5. Full file system check is initiated if one of the following is true:
   - The *-n* option was specified.
   - The file system state is not CLEAN and the *-y* option was specified.
   - The *-o full* option was specified.

8.7.1.1 Superblock verification

The superblock verification phase checks to see that the superblock specified a valid VxFS file system. It also ensures that all allocation unit (au) headers for the file system match the superblock. In normal operation, superblock verification succeeds, except for the following cases:

- The superblock was physically damaged.
- An allocation unit header was physically damaged.
- The system failed during a *resize* operation.

If the superblock is damaged, the superblock can be restored from an auxiliary copy found in the first allocation unit header.

If an allocation unit header is damaged, it can be restored from the superblock.

If a *resize* operation (ref *fsadm*(1M)) was in progress when the system failed, there is a possibility that the superblock and allocation unit headers do not match. The process of undoing the interrupted *resize* operation begins with the restoration of the superblock copies in the allocation unit headers. An interrupted *resize* operation can only be fully undone, however, through a full file system check.

8.7.1.2 Sanity check

The *-m* option of *fsck*(1M) is used by *mountall*(1M) to determine if file systems need to be checked before they can be mounted.

The VxFS-specific *fsck*(1M) checks for a valid superblock to see if the file system is in the CLEAN state. In all cases a message is displayed on *stderr*, and the program terminates with an exit code.

The following exit codes can be found:

0	The file system is clean.
32	The file system needs checking.
33	The file system is already mounted.
34	There was an error when attempting to access the file system.

8.7.1.3 Intent log replay

The VxFS file system records pending updates to the file system in an intent log. The intent log replay reads the intent log and completes or "rolls back" the pending updates.

Once the pending updates have been processed, any pending extended inode operations are performed. Extended inode operations are operations which may remain pending for an extended period of time, and thus are not suitable to be recorded in the intent log. The following are extended inode operations:

1. Deferred removal of files.

   It is possible to remove the last link to a file while the file is being accessed by a process. The file is no longer accessible by name, but the file must remain in existence until the accessing process exits. In the *s5* and *ufs* file systems, *fsck* places such files in *lost+found*. The VxFS file system specifically marks a deferred removal situation in the *i_eopflags* field of the inode, allowing the *fsck* to safely remove the file.

2. Truncation of large files

   A large file can contain a large number of extents. If a large file is to be truncated, the intention to remove a large number of extents cannot fit in the intent log. In order to ensure that the file does not remain as partially truncated, pending truncations are recorded in the *i_eopflags* field of the inode, and the truncation is completed by *fsck*.

3. Clearing of Large Extents

   The VxFS file system supports large extents that can be up to 256 Kbyte blocks in size. It is possible to have sparse files with large extents. In this case, if an extent is allocated in the "middle" of a file, the extent must be zeroed so that previous data on the disk is not retained. When this condition is detected by *fsck*, the extent is removed from the file.

Intent log replay and extended inode operation completion are normally very brief. An elapsed time of one or two seconds is common. However, if a large file needs to be truncated, the recovery takes considerably longer.

8.7.2 Full file system check phase

Four passes through the file system are performed as part of full file system check:

Pass 1: Check inode sanityCheck for bad or duplicate extent references.

Pass 2: Check directory entriesCheck and verify directory entries.

Pass 3: Check link counts of filesCheck that no files in the file system are "orphaned".

Pass 4: Check free inode maps, free extent maps, and superblock counts.

8.7.2.1 Full file system check: pass 1, inode sanity check

This pass performs the following functions:

1. Basic inode sanity checks are performed on the files to ensure that mode, link count, file size, extent allocation information, and extended inode operation information is valid. If the basic information is invalid, the file is considered to be bad.
2. If the file passes basic checking, any pending extended inode operations are performed (see Section "Intent log replay" for details).
3. All extents allocated to the file are checked to ensure that all references are to valid block numbers, and that blocks are not allocated to multiple files.
4. The basic file size information is rechecked to ensure that the file size and extents allocated to the file are in agreement

If an inode fails sanity checking, it should be removed. Unless *fsck* is being run with the *-n* option, answering *no* to any of the messages is **extremely dangerous to your file system integrity**.

If it is essential to save an inode, for example, an irreplaceable file otherwise would be lost, select the *q* (quit) option, use *fsdb* to attempt to fix the problem, and recheck the file system with *fsck*.

Unless you are extremely experienced in file system debugging, these procedures are NOT recommended.

8.7.2.2 Full file system check: pass 2, directory entry check

Pass 2 checks directory blocks and directory entries. During Pass 2 the following checks are performed:

1. Directory entries are checked to ensure that they have valid inode numbers and valid file names.
2. If a directory has subdirectories, the *i_dotdot* field of each subdirectory is checked to ensure that the subdirectory acknowledges the referencing directory as its parent.
3. Free space counts and hash chains in the directory block are validated.

Some inconsistencies (for example, if data in the directory is invalid or has become corrupted) cause entire directory blocks to be considered invalid. Other inconsistencies are treated as minor and cause directory blocks to be updated but retained.

If a directory or directory block is considered to be invalid and gets cleared, any files originally referenced by the directory or directory block will no longer be referenced. In this situation, Pass 3 will report unreferenced files, or files with invalid link counts.

8.7.2.3 Full file system check: pass 3, link count check

Pass 3 checks

1. that all files are referenced by directories,
2. that all "dot-dot" (..) paths from directories lead back to the root inode of the file system,
3. that the link counts of files are correct.

If a file is not referenced, or the ".." path does not lead back to the root inode, the file must be attached to the *lost+found* directory, or removed.

8.7.2.4 Full file system check: pass 4

Free inode, free extent, superblock count check

Pass 4 performs the following checks:

1. The free inode maps on disk agree with the file system as it was found or reconstructed.
2. The free extent maps agree with extents allocated to files.
3. The allocation unit summaries match the counts of free inodes and extents.
4. Clears the intent log to ensure that the intent log is in a known state and accessible.
5. Checks that superblock free counts agree with actual free inodes and extents.
6. Marks the file system as CLEAN so that it can be mounted.

8.7.3 Example

The superblock of the filesystem has been lost (probably because of a disk error). The superblock can only be restored by a full file system check using the *fsck*(1M) command:

```
# fsck -F vxfs -l /dev/ios0/sdisk110s7
invalid superblock magic 00000000
rebuild from auxiliary? (ynq)y
found au header at offset 526336
creation time Fri Nov 13 11:21:04 1998
blocks 2000000  inodes 524288

rebuild superblock from au header? (ynq)y
log replay in progress
pass1 - checking inode sanity and blocks
pass2 - checking directory linkage
pass3 - checking reference counts

pass4 - checking resource maps

OK to clear log? (ynq)y

free inode count incorrect found 0 expected 524278 fix? (ynq)y
free block count incorrect found 0 expected 1933863 fix? (ynq)y
free extent vector incorrect fix? (ynq)y

set state to CLEAN? (ynq)y
```

8.8 fsdb(1M)

The *fsdb* utility is used to examine or repair damaged file systems. *fsdb* is most likely to be used on a file system that has had previous I/O errors which have rendered some data unreachable, or has caused the file system to be unaffected by the *fsck* utility. *fsdb* also has a non-interactive mode that allows a particular inode to be zeroed.

! *fsdb* is intended for use by an **experienced** user. Mistakes can be devastating to file system integrity.

The *fsdb*(1M) command can be used to examine and alter disk blocks, inodes, and directories. It also has commands to examine other file system structures such as maps, the superblock, allocation unit headers, and log entries. This utility reads and writes data a block at a time, so that the *fsdb* can be used with either a block or character special devices. Any changes are immediately written to disk.

ℹ Changes should only be made to an unmounted file system, and a full *fsck* should be done before attempting to mount the file system.

fsdb(1M) can be run on a snapshot file system and will show the actual data on the disk, rather than an image of the snapped file system. The superblock is the only structure that can sensibly be printed by *fsdb*(1M) on a snapshot file system.

Synopsis

fsdb [**-F** vxfs] [**generic_options**][**-z** i-number] special

Command options

-z *i-number*

Causes *fsdb* to be invoked in a non-interactive mode, where the inode specified is zeroed. Multiple inodes may be specified by using multiple □ *-z* options. The following conditions should apply when using the *-z* option:

1. The file system should be unmounted before using the *fsdb -z* command.
2. The file system should be subjected to a full *fsck* after the *fsdb -z* command, and only then remounted.

If the *-z* option is not specified, *fsdb* reads the file system superblock to determine the file system block size, and then enters interactive mode. To quit the interactive mode of *fsdb*, enter the *q* or *Ctrl-D* commands.

8.8.1 Interactice mode

Commands are generally one, two, or three letters. Numbers are specified in hex (by prepending 0x to the number), in octal (by prepending 0 to the number) or in decimal (the default). Spaces are generally optional, and are required only to reduce ambiguity. Hex numbers should always be followed by white space to avoid confusing commands with part of the number.

There are three basic types of commands; those which:

- set position,
- print information,
- alter data on the file system.

Positioning commands change the current position, and generally print the contents of the new position if they are the last command on a line. Print commands print information based on the current position. Alteration commands alter data at the specified position, and change the current position to that position.

The basic data types are

- strings and integers (bytes, shorts, and words),
- inodes,
- directories,
- directory entries,
- miscellaneous (log entries, the superblock, and allocation unit headers).

The commands are grouped and described as they relate to these data types.

There are several notions of position within a file system:

- the current position, which is the offset within the file system,
- the current inode,
- the current directory block,
- the current directory entry (within the current directory block),
- the current allocation unit.

Setting the current inode, directory block, directory entry, and au also set the current position to the same value.

Help screen

An abbreviated help screen is available while using the *fsdb* command. This screen contains the abbreviated form for all interactive *fsdb* commands. To access this command, type help or a question mark (?).

```
Legend: [] = optional argument, | = select one of the list, # = number□
# - use 0x prefix for hex, 0 for octal, decimal is default□
B = bytes, H = short, W = word, D = double, i = inode, b = block□
```

```
c = char, x = hex, o = octal, e = decimal, S = super, L = intent log□
A = au header, I = inode, db/dh/dent = dir block/header/entries, □
! for shell.   use . or ; to put multiple commands on one line□
[+|-]  #  B|H|W|D|b                  -  seek to relative or absolute position□
[+|-]  #  i|au|d           - seek to relative or absolute inode/au/dir entry□
i|cdb|au|d              - seek to current inode/directory block/au/dir entry□
a # | im        - seek to address block # or immediate area of current inode□
d #           -  seek to #'th directory entry in the current directory block□
[+|-]  #  B|H|W|D  =  #                   -  change number at given position□
[+|-]  #  B|H|W|D  =  "string"            -  change string at given position□
p  [#]  c|x|e|o|xB|eB|oB|xH|eH|oH|xW|eW|oW                    - numeric print□
p  [#]  L|I|db|dh|dent  or  p  S|A                        - structure print□
md|ln|uid|gid|sz|at|mt|ct|maj|min|org|serhi|serlo  =  # - change inode field□
gen|af|pd|res|fe|bl|de #|des #|ie #|ies|eopflg|eopdat = # change inode field□
tfree|hash #|nhash  =  #               - change directory block header field□
ino|nmlen|reclen|hnext = # | nm = "string"    - change directory entry field□
calc  #  [+|-|*|/ #]                  - simple calculater and base converter□
find  # B|H|W|D | "string"  [#]     - find matching pattern in next # blocks□
fmtlog                                          - print the entire log area□
q|<cntl D>                                                - exit from fsdb□
```

8.8.1.1 Integers and strings

The most basic set of commands deal with integers and strings.

Positioning commands

+|- *n* **B**
: set current position to byte *n*

+|- *n* **H**
: set current position to half-word *n*

+|- *n* **W**
: set current position to word *n*

+|- *n* **D**
: set current position to double-word *n*

+|- *n* **b**
: set current position to block *n*

n is a number. If preceded by a "*+"* or "*-"*, it is relative to the current position. Otherwise it is absolute (from the beginning of the file system).

Printing commands

*p n***x**
: print *n* words in hex

*p n***e**
: print *n* words in decimal

*p n***o**
: print *n* words in octal

*p n***c**
: print *n* bytes as characters

*p n***xB**
: print *n* bytes in hex

*p n***eB**
: print *n* bytes in decimal

*p n***oB**
: print *n* bytes in octal

p n **xH**

print *n* half-words in hex

p n **eH**
print *n* half-words in decimal

p n **oH**
print *n* half-words in octal

*p n***xW**
print *n* words in hex (same as *x*)

p n **eW**
print *n* words in decimal (same as *e*)

p n **oW**
print *n* words in octal (same as *o*)

n is an optional number which specifies the number of items to print. If *n* is omitted, it defaults to one. Printing always begins at the current position.

Alteration commands

Alteration commands are used to alter the contents of a byte, half-word, or word. These commands are essentially the positioning commands with an equal sign (=) appended, though they are syntactically separate from the positioning commands. The commands require a position which may be "+0" to indicate a current position.

n **B = x**
set byte at *n* to *x*

n **H = x**
set half-word at *n* to *x*

n **W = n**
set word at *n* to *x*

n **D = x y**
set double-word at *n* to *x y*(two values are specified, one for each word)

n is a number. If preceded by a *"+"* or "-*"*, it is relative to the current position. Otherwise it is absolute (from the beginning of the file system).

8.8.1.2 Inode commands

There are several commands that are used to examine inodes, and a host of commands that are used to alter particular inode fields. Commands are also provided to move around in the file defined by the inode.

Positioning commands

[[+|-]n]i

set the current inode to *n*. The *i* command has also the effect of setting the current position to the specified inode, and will also print the inode if it is not the last command on the line.

If plus (+) or minus (-) are specified, the inode is set relative to the current inode. If no number is specified, the current position is set to the current inode.

There are also commands available to position within the current inode:

im set the current position to the immediate data area of the current inode

a *n* set the current position to the start of extent *n*; only works for direct extents

Printing commands

p *n***I**
print *n* inodes at the current positionFormat the data at the current position as inode data.

Alteration commands

There are a number of commands that act on the current inode to change particular fields. These commands are:

md = x	set the file mode to *x*
ln = x	set the link count to *x*
uid = x	set the owner id to *x*
gid = x	set the group id to *x*
sz = x	set the file size to *x*
at = x	set the file access time to *x* (in seconds)
mt = x	set the file modification time to *x* (in seconds)
ct = x	set the inode modification time to *x* (in seconds)
maj = x	set the special major number to *x*
min = x	set the special minor number to *x*
serhi = x	set the high word of the serial number to *x*
serlo = x	set the low word of the serial number to *x*
gen = x	set the generation count to *x* (used by NFS)
af = x	set the allocation flags to *x*
pd = x	set the parent directory inode number to *x*
res = x	set the reservation amount to *x*
fe = x	set the fixed extent size to *x* blocks
bl = x	set the blocks in file to *x*
de n = x	set direct extent *n* to *x* blocks
des n = x	set the size of direct extent *n* to *x* blocks
ie n = x	set indirect extent *n* to *x*
ies = x	set the size of indirect extents to *x*
eopflg = x	set the inode extended operation flag to *x*
eopdat = x	set the inode extended operation data to *x*
org = x	set the mapping type auf *x*

Care should be taken when manipulating inode fields. Inconsistencies, such as an invalid mode, invalid extent size or number, or a mismatch between the count of blocks in the inode and the number of blocks in allocated extents, will cause *fsck* to clear the inode.

8.8.1.3 Directory commands

Directory commands fall into two groups: those dealing with directory block headers, and those dealing with directory entries.

Positioning commands

[+|-] n **b**	set current position **and** current directory block to block *n*; clear current directory entry
cdb	set current position to current directory block
d n	set current directory entry to *n*th entry

The *im* command also has the effect of clearing the current directory entry and setting the current directory block pointer to point at the immediate area of the inode.

Printing commands

p [n]**db**	print **one** directory block at current position
p [n]**dh**	print **one** directory header at current position
p [n]**dent**	print *n* directory entries at current position

The *db* and *dh* formats print only **one** directory block or header, regardless of the value of *n*. *db* and *d n* use the contents of the directory header to decide where the directory entries start and how many of them there are.

Alteration commands for directory header fields

tfree = x	change total free bytes to *x*
nhash = x	change number of hash buckets to *x*
hash n = x	change hash bucket *n* to *x*

Alteration commands for directory entry fields

ino = x	change inode number to *x*
nm = filename	change name to string *filename*
nmlen = x	change name length to *x*
reclen = x	change record length to *x*
hnext = x	change hash next offset to *x*

8.8.1.4 Allocation unit header commands

The only au header commands that exists set the current au, and print the au header at the current position. There are no commands to alter the au header since the header is monitored by *checksum*, and any changes are likely to result only in *fsck* errors.

Any changes to an au header must be changed in all au headers and the superblock. The *fs_checksum* (a simple additive checksum) must be correctly updated.

Positioning commands

[[+|-]n] **au**	set the current position and the current au to the *n*th au;with no argument sets the current position to the current au

Printing commands

p A print au header at the current position

8.8.1.5 Miscellaneous commands

p S print superblock at current position

p [n] L print *n* log entries at current position

calc x [+|-|*|/ y] print *x* (or x +|-|*|/ *y*) in hex, octal, decimal and as a (valid) character

find [x][**B|H|W|D**] n find byte, half-word or word *x* in the next *n* blocks

find x y [**D**] n find double-word *x y* in the next *n* blocks(two values are specified, one for each word)

find string [n] find *string* in the next *n* blocks

fmtlog print all current log entries

q quit

Ctrl-D quit

8.8.2 Examples

These examples show how to perform certain common operations on a file system. They are in the form of scripts that document what the user types and what *fsdb* responds with.

Example 1

Find the inode for the file *debug/src/cpuburn/cpuburn.c*. The leading "/" is not used because the filename is relative to the mountpoint of the file system. Inode numbers can be more easily determined by the *-i* option to *ls* when the file system is mounted.

```
# fsdb -F vxfs /dev/ios0/rsdisk000s2□
> 2i # file system root inode
inode structure  at 00013a00□
mode 40755  nlink 4  uid 0  gid 0  size 1024□
atime 666932963  mtime 666932853  ctime 666932853  aflags 0 orgtype 1□
fixextsize 0  gen 0  rdev/reserve/dotdot 2  blocks 1□
eopflags 0  eopdata 0 serial 0-5□
de:  2074    0    0    0    0    0    0    0    0    0□
des:    1    0    0    0    0    0    0    0    0    0□
ie:     0    0□
ies:    0    0□
> 2074b # examine the first data block
00206800:  00200398
> p db # print as a directory block
directory block at 206800 - total free (d_tfree) 920 nhash 32□
Hash    0- 7:  0000 0000 0000 0000 0000 0000 0000 0058□
Hash    8-15:  0000 0044 0000 0000 0000 0000 0000 0000□
Hash   16-23:  0000 0000 0000 0000 0000 0000 0000 0000□
Hash   24-31:  0000 0000 0000 0000 0000 0000 0000 0000□
00206844:  d 0    d_ino 3  d_reclen 20  d_namlen 10  d_hashnext 0000□
l o s t + f o u n d□
00206858:  d 1    d_ino 4  d_reclen 936  d_namlen 5  d_hashnext 0000□
d  e  b  u  g
> 4 i # examine "debug", inode 4
inode structure  at 00013c00□
mode 40777  nlink 11  uid 0  gid 1  size 1024□
atime 666933073  mtime 666933074  ctime 666933074  aflags 0 orgtype 1□
fixextsize 0  gen 0  rdev/reserve/dotdot 2  blocks 1□
eopflags 0  eopdata 0 serial 0-20□
de:  2425    0    0    0    0    0    0    0    0    0□
des:    1    0    0    0    0    0    0    0    0    0□
ie:     0    0□
ies:    0    0□
> 2425b. p db    # look at first directory block
directory block at 25e400 - total free (d_tfree) 812 nhash 32□
Hash    0- 7:  0000 0000 0000 0000 0000 0000 0000 00a8□
Hash    8-15:  0078 0044 0000 0000 0000 0098 0000 0000□
Hash   16-23:  00c8 0068 00b8 0000 0000 0000 0088 0000□
Hash   24-31:  0000 0058 0000 0000 0000 0000 0000 0000□
0025e444:  d 0    d_ino 8  d_reclen 20  d_namlen 10  d_hashnext 0000□
l o s t + f o u n d□
0025e458:  d 1    d_ino 9  d_reclen 16  d_namlen 3  d_hashnext 0000□
b  i  n□
0025e468:  d 2    d_ino 10  d_reclen 16  d_namlen 3  d_hashnext 0000□
t  m  p□
0025e478:  d 3    d_ino 11  d_reclen 16  d_namlen 3  d_hashnext 0000□
s  r  c□
0025e488:  d 4    d_ino 52  d_reclen 16  d_namlen 4  d_hashnext 0000□
d  u  m  p□
0025e498:  d 5    d_ino 53  d_reclen 16  d_namlen 5  d_hashnext 0000□
a  i  m  +  +□
```

```
0025e4a8:  d 6    d_ino 54  d_reclen 16  d_namlen 4  d_hashnext 0000□
v  x  f  s□
0025e4b8:  d 7    d_ino 55  d_reclen 16  d_namlen 4  d_hashnext 0000□
s  v  v  s□
0025e4c8:  d 8    d_ino 204  d_reclen 824  d_namlen 2  d_hashnext 0000□
m  c□
> 11i  # examine "src", inode 11
inode structure  at 00014300□
mode 40775  nlink 17  uid 0  gid 1  size 1024□
atime 666933079  mtime 666932905  ctime 666932905  aflags 0 orgtype 1□
fixextsize 0  gen 0  rdev/reserve/dotdot 4  blocks 1□
eopflags 0  eopdata 0 serial 0-25□
de:  31250     0     0     0     0     0     0     0     0     0□
des:     1     0     0     0     0     0     0     0     0     0□
ie:      0     0□
ies:     0     0□
> a 0.p db # look at first directory block
directory block at 1e84800 - total free (d_tfree) 704 nhash 32□
Hash    0- 7:  0068 0000 0000 00f4 0000 0000 0000 0000□
Hash    8-15:  00d4 00a4 0000 00e0 0000 0000 0000 0110□
Hash   16-23:  0000 007c 0000 00c4 0000 0000 0000 0130□
Hash   24-31:  0000 0104 0000 0000 0000 0000 0090 0054□
01e84844:  d 0    d_ino 16  d_reclen 16  d_namlen 6  d_hashnext 0000□
t  i  o  c  t  l□
01e84854:  d 1    d_ino 17  d_reclen 20  d_namlen 7  d_hashnext 0000□
c  p  u  b  u  r  n□
01e84868:  d 2    d_ino 18  d_reclen 20  d_namlen 7  d_hashnext 0000□
f  s  d  e  b  u  g□
01e8487c:  d 3    d_ino 19  d_reclen 20  d_namlen 7  d_hashnext 0000□
p  r  i  n  o  d  e□
01e84890:  d 4    d_ino 32  d_reclen 20  d_namlen 8  d_hashnext 0000□
g  e  t  d  e  n  t  s
...  □□□□□□□□# lines deleted for clarity□
> 17i □□□□□□□□□# look at ,,cpuburn", inode 17
inode structure  at 00014900□
mode 40775  nlink 2  uid 0  gid 1  size 96□
atime 666933096  mtime 666932868  ctime 666932868  aflags 0 orgtype 2□
fixextsize 0  gen 0  rdev/reserve/dotdot 11  blocks 0□
eopflags 0  eopdata 0 serial 0-7□
> im.p db # print immediate area as a directory block
immediate directory block at 14950 - total free (d_tfree) 52□
00014954:  d 0    d_ino 1016  d_reclen 20  d_namlen 9□
c  p  u  b  u  r  n  .  c□
00014968:  d 1    d_ino 1017  d_reclen 72  d_namlen 7□
c  p  u  b  u  r  n□
> 1016i # examine ,,cpuburn.c", our target inode
inode structure  at 00053000□
mode 100664  nlink 1  uid 0  gid 1  size 1805□
atime 666918286  mtime 643066719  ctime 666932898  aflags 0 orgtype 1□
fixextsize 0  gen 0  rdev/reserve/dotdot 0  blocks 2□
eopflags 0  eopdata 0 serial 0-6□
de:  2530    0    0    0    0    0    0    0    0    0□
des:    2    0    0    0    0    0    0    0    0    0□
ie:     0    0□
ies:    0    0□
> q            # quit
```

Example 2

A VxFS file system mounted on */mnt* has developed I/O errors when reading a very important file, */mnt/debug/dump/foo*. The file can't be restored from backups (perhaps because they weren't made in a timely manner), and the important application that reads the file immediately fails when it encounters an I/O error. However, if the bad data were replaced by zeros, the rest of the file could be recovered. The I/O errors occur in the sixth block of the file. The following example removes the sixth block of */mnt/debug/dump/foo*.

```
# ls -li /mnt/debug/dump/foo
1299 -rw-rw-r--  1 root   staff   40596 Feb 18 19:24 /mnt/debug/dump/foo
# umount /mnt □□# always unmount the file system
# fsdb -F vxfs /dev/ios0/rsdisk000s2
# 1299i # the file in question
inode structure  at 00064b00□
mode 100664  nlink 1  uid 0  gid 1  size 40596□
atime 666933618  mtime 666933859  ctime 666933860  aflags 0 orgtype 1□
fixextsize 0  gen 0  rdev/reserve/dotdot 0  blocks 40□
eopflags 0  eopdata 0 serial 0-559□
de:  7566 8098 8154    0    0    0    0    0    0    0□
des:    4   24   12    0    0    0    0    0    0    0□
ie:     0    0□
ies:    0    0□
□□□□□□□□□□□□□# bad news: the block in question is□
□□□□□□□□□□□□□# located in an extent of 24 blocks.□
□□□□□□□□□□□□□# we have to delete all of them□
> de 1 = 0 □□□# replace the extent with a hole
00064b68: 0
> 1299i
inode structure□□at 00064b00□
mode 100664□□nlink 1□□uid 0□□gid 1□□size 40596□
atime 666933618□□mtime 666933859□□ctime 666933860□□aflags 0 orgtype 1□
fixextsize 0□□gen 0□□rdev/reserve/dotdot 0□□blocks 40□
eopflags 0□□eopdata 0 serial 0-559□
de:□□7566□□□□□0 8154□□□□0□□□□0□□□□0□□□□0□□□□0□□□□0□□□□0□
des:□□□□4□□□□24   12□□□□0□□□□0□□□□0□□□□0□□□□0□□□□0□□□□0□
ie:□□□□□0□□□□0□
ies:□□□□0□□□□0
□
> calc 40 - 24 # we remove 24 blocks from the file
□□□□□□□16□□000000000020□□0x00000010
□
> bl = 16 # update the block count; otherwise fsck will clear the inode
00064b40: 16
> 1299i
inode structure□□at 00064b00□
mode 100664□□nlink 1□□uid 0□□gid 1□□size□
40596atime 666933618□□mtime 666933859□□ctime 666933860□□aflags 0 orgtype 1□
fixextsize 0□□gen 0□□rdev/reserve/dotdot 0□□blocks 16□
eopflags 0□□eopdata 0 serial 0-559□
de:□□7566□□□□0 8154□□□□0□□□□0□□□□0□□□□0□□□□0□□□□0□□□□0□
des:□□□□4□□□24□□□12□□□□0□□□□0□□□□0□□□□0□□□□0□□□□0□□□□0□
ie:□□□□□0□□□□0ies:□□□□0□□□□0

□
> q # we're done here
□
# fsck -F vxfs -ofull -y /dev/ios0/rsdisk000s2□
# complete file system check
file system is clean - log replay is not required□
pass1 - checking inode sanity and blocks□
pass2 - checking directory linkage□
pass3 - checking reference counts□
pass4 - checking resource maps□
```

```
au 0 emap incorrect - fix? (ynq)y
au 0 summary incorrect - fix (ynq)y
OK to clear log? (ynq)y
free block count incorrect found 50049 expected 50073 fix? (ynq)y
free extent vector incorrect fix? (ynq)y
set state to CLEAN? (ynq)y
□□□□□□□□□□□□□□□□□□□□# now mount the file system, copy the file, □
□□□□□□□□□□□□□□□□□□□□# and then fix the problem with the disk
```

8.9 fstyp(1M)

The *fstyp*(1M) utility is used to check the type of file system that is present on a disk partition. Each file system type provides a file system specific version of *fstyp*. The *fstyp* utility may be used to determine if there is a file system on a system on a given partition, and if so, which type of file system. When *fstyp* is run, the generic utility checks the */usr/lib/fs* directory to see which file systems are installed on the system. The generic utility runs the *fstyp* utility from each directory it finds.

The file system-specific versions of the utility check for identifying signs of a file system. If the appropriate file systems are present, the utility will display the type string for the file system on standard output, and return an exit code of zero. If the appropriate file system is not found, a non-zero exit code is returned.

In the case of the VxFS file system, the *fstyp* utility checks for a VxFS superblock. Specifically, the *fs_magic* field of the superblock is checked to see if it is *VX_MAGIC*. The format of the superblock is specified in the man page of *fs*(4). If a VxFS file system is found, *vxfs* is displayed on standard output. There is a verbose mode to the *fstyp*(1M) command, which is specified by using the *-v* option on the command line. When the verbose mode is used, the superblock fields, the number of free blocks and inodes, and the number of free extents by size are displayed.

Synopsis

fstyp [**-v**]device

Command options

-v The superblock fields are also displayed (verbose).

If *device* contains a VxFS file system, *vxfs* is displayed on standard output.

8.10 labelit(1M)

The *labelit*(1m) utility provides *volcopy*-format labels for VxFS file systems. The *labelit* utility has two basic functions: to read and to write labels. Labeling tapes before using them with *volcopy* is encouraged since it cuts down on *volcopy* errors that would have to be overridden and also acts as a check that the correct tapes have been loaded.

The generic *labelit* program executes the file system-specific *labelit* program according to the *-F* option. The VxFS-specific *labelit* determines if the device is a tape drive by virtue of its name containing either *rmt*, or *rtp*.

Labels are used on superblocks and tapes. File system labels consist of a one to six-character file system name and a one to six character volume name. The file system label should match the last component of the mountpoint; otherwise the *mount* command will issue a message concerning the mismatched mountpoint and volume name. The volume name is often used to indicate the device on which the file system resides. The *volcopy* command supports labeled tapes.

It is not possible to relabel mounted file systems. A mounted file system that has been relabeled will revert back to its original name.

Synopsis

labelit [**-F** vxfs][**generic_options**] [**-n**] special [fsname volume]

Command options

-n Provided to prevent *labelit* from trying to read the label on a tape before rewriting a new label. This allows new tapes to be labeled without the read failing because of an end-of-tape or end-of-file mark.

fsname
Represents the mounted name of the file system.

volume
May be used to equate an internal name to a volume name applied externally to the disk pack, diskette or tape.

8.11 mkfs(1M)

mkfs is a utility for creating VxFS file systems. *mkfs* creates a file system by writing into device file *special*, provides option *-o N* has been specified. The numeric variable *size* specifies the size of the file system in sectors. If *size* is specified as a "-" (hyphen) the complete device file is used. *mkfs* creates a file system with a root and a *lost+found* directory (see *fsck*(1M)). The number of inodes is calculated as a function of the file system size. No boot program is initialized by *mkfs*.

generic_options are options which are supported by generic command *mkfs*. *specific_options* are suboptions which are supported by the Veritas-specific module of the *mkfs* command. They can be specified in a list of comma-separated values and/or as pairs of keywords and attributes. These suboptions are specified using option *-o*.

Synopsis

mkfs [**-F** vxfs] [**generic_options**] [**-o** specific_options]special size

Command options

-o *specific_options*

Specifies the VxFS-specific suboptions in a comma-separated list. The suboptions are:

N
: The file system is not created. This option supplies all information which is necessary to create a file system but does not create the system.

ninode=*n*
: Specifies number of inodes in the file system (rounded up). The default value is the total number of blocks in the allocation units divided by 4 (see Choosing the number of inodes).

logsize=*n*
: Specifies intent log size in blocks in the range of 32 to 1024. The default is 256 blocks (see Choosing a log size).

ausize=*n*

Specifies an allocation unit (au) size in *bsize* blocks. This can be used as a way of defining the allocation units. This option should not be used in conjunction with the *nau* option. With this option it can happen that the last allocation unit of the file system is shorter than the others. When the last allocation unit of the file system is too small to accommodate the complete header of the allocation unit the file system only extends to the end of the last complete allocation unit.

bsize=*n*

Specifies the block size for files in the file system and the smallest hard disk to which a file will be allocated. *n* must be a power of 2 in the range 1024 to 8192. The default value is 1024. For file systems >32 Gbyte the block size depends on the maximum file size (cf. *maxfilesize*) and on the size of the device file (the block numbers must be able to be represented in "signed 32-bit long").

maxfilesize=*n*

Specifies the maximum size of a file in the file system to be created (in Gbyte). *n* must be a power of 2 in the range 32 to 16384. By default *maxfilesize* is at least as big as the size of the file system.

Note: If the maximum file size is smaller than the size of the file system, it is not possible for a file to be larger than *maxfilesize* even if there is enough disk space available (see Notes).

iaddrlen=*n*

Specifies the size of the indirect address extent for setting up the indirect stages (in blocks)). *n* must be a power of 2 in the range 1 to 64. *iaddrlen* * *bsize* must be >= 8 Kbyte and <= 64 Kbyte (see Notes).

indsize=*n*

Specifies the size of indirect data extents in the indirection stages. *n* must be a power of 2 in the range of 1 to 1024 (in blocks) . *indsize* * *bsize* must be >= 8 Kbyte (see Notes).

nau=*n*

Specifies the number of allocation units of the file system. The default is the number of available blocks after the superblock and the log area were allocated, divided by 32768. Each allocation unit is given the same number of blocks of size *bsize*. This can result in the generated file system being smaller than the size specified in *size*.

aufirst=*n*

Is the first block number in blocks of size *bsize*, of the first allocation unit. This option allows allocation units of a specific size to be allocated, such as a cylinder boundary for example.

aupad=*n*

The size in blocks which is to remain for each allocation unit between the end of the inode-list and the first data block. This option allows allocation units of a specific size to be allocated, such as a cylinder boundary for example.

special

should be the character (or *raw*) disk device

size

is specified in 512 byte sectors.With "-" (dash) the entire device size will be used.

Notes

The tables below show the relationship between *bsize*, *iaddrlen* and *indsize* in respect of *maxfilesize*. If *iaddrlen* and *indsize* are not specified, *mkfs* uses the following strategy to represent the maximum file size:

Initially *iaddrlen* is increased if it is not sufficient for *maxfilesize*, then *indsize* is increased and *iaddrlen* set to the smallest-possible . This procedure reduces the space requirements for administration of data in the indirect levels.

Maximum file sizes for different *iaddrlen* values and a *bsize* value of 1 Kbyte:

bsize 1 Kbyte	*indsize* (blocks)							
iaddrlen blocks	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
8	32 GB	64 GB	128 GB	256 GB	512 GB	1 TB	2 TB	2 TB
16	128 GB	256 GB	512 GB	1 TB	2 TB	2 TB	2 TB	2 TB
32	512 GB	1 TB	2 TB	2 TB	2 TB	2 TB	2 TB	2 TB
64	2 TB	2 TB	2 TB	2 TB	2 TB	2 TB	2 TB	2 TB

Maximum file sizes for different *iaddrlen* values and a *bsize* value of 2 Kbytes:

bsize 2 Kbyte	*indsize* (blocks)								
iaddrlen blocks	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
4	32 GB	64 GB	128GB	256GB	512GB	1 TB	2 TB	4 TB	4 TB
8	128GB	256GB	512GB	1 TB	2 TB	4 TB	4 TB	4 TB	4 TB
16	512GB	1 TB	2 TB	4 TB	4 TB	4 TB	4 TB	4 TB	4 TB
32	2 TB	4 TB	4 TB	4 TB	4 TB	4 TB	4 TB	4 TB	4 TB

Maximum file sizes for different *iaddrlen* values and a *bsize* value of 4 Kbytes:

bsize 4 Kbyte	*indsize* (blocks)							
iaddrlen blocks	**2**	**4**	**8**	**16**	**32**	...	**512**	**1024**
2	32 GB	64 GB	128 GB	256 GB	512 GB	...	8 TB	8 TB
4	128 GB	256 GB	512 GB	1TB	2TB	...	8 TB	8 TB
8	256 GB	1TB	2TB	4TB	8TB	...	8 TB	8 TB
16	2 TB	4TB	8TB	8TB	8TB	...	8 TB	8 TB

Maximum file sizes for different *iaddrlen* values and a *bsize* value of 8 Kbytes:

bsize	*indsize* (blocks)	

8□Kbyte *iaddrlen*□ blocks	**1**	**2**	**4**	**8**	**16**	...	**512**	**1024**
1	32 GB	64 GB	128 GB	256 GB	512 GB	...	16 TB	16 TB
2	128 GB	256 GB	512 GB	1 TB	2 TB	...	16 TB	16 TB
4	512 GB	1 TB	2 TB	4 TB	8 TB	...	16 TB	16 TB
8	2TB	4 TB	8 TB	16 TB	16 TB	...	16 TB	16 TB

Detailed examples of VxFS-specific *mkfs* commands can be found in the Alignment.

The -o N option

When you specify *-o N*, the geometry of the file system that would be created given the command line is displayed, but the file system is not created. When experimenting with layouts, use this option until you are satisfied with the layout.

Choosing the number of inodes

In general, the default options for your installation should be used. However, the number of inodes may need to be changed. By default, an inode is allocated for every four data blocks. For very large file systems, with an expectation of large files, this will be too many inodes.

It is a good idea to overestimate the number of inodes. Default overhead for inodes is only six percent, and if the file system runs out of inodes there will be unusable space. The number of inodes gets rounded down by *mkfs*.

For file systems which are larger than 4 Gbyte (up to 16 Tbyte) the number of inodes is restricted to 1 million by default. A larger number of inodes can be set using the option *-o inodes*, but consequently a full file system check would take a lot of time (more than 4 hours with a size of 1 TByte).

Choosing a log size

The default is 256 blocks. This should be sufficient unless there is a great degree of file creation and deletion, in which case, you may wish to try 512 blocks. If you have system benchmark programs try the combinations. If not, stick with the default.

Choosing allocation unit size

The default allocation unit size is approximately 32 Kbyte blocks. For file systems that are smaller than 32 Kbyte blocks, the default allocation unit size will be the size from the start of the first allocation unit to the end of the file system.

The default makes all allocation units the same size, for simplicity. This strategy is not optional. Since the extent map is kept in powers of 2, it is best to have a power of 2 as the number of data blocks.

For large file systems (more than eight allocation units), you may wish to specify allocation units of 64 Kbyte data blocks. Fragmentation and overhead may be somewhat reduced by going to larger allocation units. The tradeoff is that the extent map per allocation unit gets larger as the *ausize* increases.

Alignment

The VxFS file system attempts to be geometry free. Since VxFS is extent-based, there is no need to specify "gap size", as VxFS attempts to schedule I/O to adjacent blocks as opposed to optimally spaced blocks. If you have a choice of disks for use with VxFS, choose the characteristics for best sequential performance.

The VxFS file system does provide facilities by which a specialty application can request allocation aligned to particular boundaries. If you are installing such an application, align your file system as follows:

1. Use *aufirst* to start the first allocation unit at an even multiple of the desired boundary.
2. Specify an *ausize* which is a multiple of the desired boundary.
3. Use *aupad* to pad the inode list so that the first data block in the allocation unit starts on a multiple of the desired boundary.

Example

Assume the system has a disk with 10 heads, 20 sectors per track, and 1000 cylinders. This disk has 200 sectors per cylinder or 200,000 sectors altogether. To make an aligned file system, the following command would be run:

```
# mkfs -F vxfs -o N /dev/ios0/rdisk000s1 200000
□□□□199996 sectors, 99998 blocks of size 1024 □
24960 inodes, 93652 data blocks □
indirect extent size 64, log size 64 □
4 allocation units of 24983 blocks, 6240 inodes, 23413 data blocks □
first allocation unit starts at block 66 □
overhead per allocation unit is 1570 blocks
```

The *o -N* option causes *mkfs*(1M) to output the parameters it computed for the file system without making it. By looking at the parameters, an attempt to align the file system can be made. Usually, the first thing to do is fix the allocation unit size. In this example, the allocation unit size is set to 25,000 sectors or 125 cylinders. Also, the *aufirst* is set to 200 which aligns the first allocation unit to a cylinder boundary.

```
# mkfs -F vxfs -o N,ausize=25000,aufirst=200 □□□/dev/ios0/rdisk000s1 200000
□□□200000 sectors, 100000 blocks of size 1024 □
24928 inodes, 93528 data blocks □
indirect extent size 64, log size 64 □
4 allocation units of 25000 blocks, 6232 inodes, 23432 data blocks □
first allocation unit starts at block 200 □
overhead per allocation unit is 1568 blocks □
last allocation unit has 23232 data blocks
```

The *mkfs*(1M) output indicates that the overhead per allocation unit is 1568 sectors. The first allocation unit has already been started on a cylinder boundary, and all the allocation units have been determined to be a cylinder size multiple.

If the overhead is padded to a cylinder boundary, the data blocks in each allocation unit will also be aligned on a cylinder boundary (*aupad*=1600-1568=32).

```
# mkfs -F vxfs -o N,ausize=25000,aufirst=200,aupad=32 □□□/dev/ios0/rdisk000s1 200000
□□□200000 sectors, 100000 blocks of size 1024 □
24928 inodes, 93528 data blocks □
indirect extent size 64, log size 64 □
4 allocation units of 25000 blocks, 6232 inodes, 23400 data blocks □
first allocation unit starts at block 200 □
overhead per allocation unit is 1600 blocks □
last allocation unit has 23200 data blocks
```

Applications can take advantage of aligned file systems by allocating storage to match the disk geometry. If the *VX_ALIGN* flag is set, any extents allocated to the file are aligned on a fixed extent size boundary relative to the start of the allocation unit. In the previous example, an application could set *VX_ALIGN* and a fixed extent size of 20 sectors (ten 1 Kbyte-blocks). Since the file system is aligned, any extents allocated to the file cover entire tracks. If the fixed extent size was set to 200 sectors, any extents would cover whole cylinders. This combination of an aligned file system and aligned extents can be used to tune applications to the I/O subsystem.

8.12 mount(1M)

The *mount* utility is used to mount VxFS file systems. The generic *mount* command also supports the *-V* option which displays the arguments as they are built on the command line.

A 32-bit file system can be mounted on 64-bit systems.

A 64-bit file system **cannot** be mounted on 32-bit systems.

This means that there is no downwards compatibility of file system versions.

Synopsis

mount [**-F** vxfs] [**generic_options**] [**-r**] [**-o** specific options] [special | mount_point]

Command options

-r Mount the file system for read only access.

-o *specific options*

rw|ro

Denotes the file system as being mounted *read/write* or for *read only* access. The default is *rw*.

suid|nosuid

This is similar to the *ufs* facility that enables *setuid* programs to execute with an effective UID according to the owner. This also applies to *setgid* permissions.

log|nolog

These flags cause the file system to be mounted with the intent log enabled or disabled. The default ist *log*.

blkclear

This flag causes all extents that become part of a file to be zeroed on disk before the inode can be modified to contain those extents.

badinode=delete|fsdisable|panic

The system administrator uses these options to determine the behavior of the system when it is mounted where an inode is to be marked as "bad".

badinode=delete marks the inode as "bad" (VX_AF_IBAD), which means that it will be removed when the next check (*fsck*) is performed (default).

badinode=fsdisable deactivates the file system but does not mark the inode as "bad". A complete file system check is required in this case to resolve the problem.

badinode=panic initiates a system panic; the inode is not marked as "bad".

remount

This flag allows a currently mounted read-only file system to be updated to read/write mount status without first having to unmount it.

datainlog|nodatainlog

These flags allow or disallow the logging of user data for synchronous writes.

snapof=snapof_special

This option is used to specify the primary file system for on-line backup. *snapof_special* is the block special file of a mounted VxFS file system.

snapsize=size

This option is used to specify the size of the snapshot file system for on-line backup.

mincache=direct|dsync|closesync

This option is used to change the default behavior of reads and writes. If the *direct* value is used, then nonsynchronous I/O behaves as if the *VX_DIRECT* caching advisory was set instead. If the *dsync* value is used, then non synchronous I/O behaves as if the *VX_DSYNC* caching advisory was set instead. If any of the values are used, files are synced to disk when they are not opened by any user no longer.

convosync=direct|dsync|closesync

This option is used to change the default behavior of synchronous reads and writes. If the *direct* value is used, then synchronous I/O behaves as if the *VX_DIRECT* caching advisory was set instead. If the *dsync* value is used, then synchronous I/O behaves as if the *VX_DSYNC* caching advisory was set instead. If the *closesync* value is used, then synchronous I/O behaves like asynchronous I/O. All three of these values sync the file to disk when it is not opened by any user no longer.

f the *convosync* option is used, the VxFS file system will not be POSIX compliant.

The default for these options are the combination of *rw*, *suid*, *log*, and *datainlog*. In Section "Choosing mount options" the *mount* options are being discussed in more detail.

As mentioned in Section "Temporary file system mode", the use of *nolog* file systems should be restricted to temporary file systems where possible, since in the event of a crash, the contents of files and the ability to recover them is uncertain. No ordered write concept exists for *nolog* VxFS file systems. It is recommended that, where possible, *nolog* file systems be remade before mounting after a crash rather than running a structural *fsck* against them.

When the *mount* command is issued, the generic *mount* command attempts to parse the arguments looking for an explicit *-F* option identifying the file system type. If one is given, then it tries to execute the specific *mount* command by first looking in */etc/fs/FSType* and then in the alternate */usr/lib/fs/FSType* directory. If no *-F* option is supplied, then the */etc/vfstab* file is searched for a file system matching the special file or mountpoint.

Example

If the following entry exists on a MX system in */etc/vfstab*:

/dev/ios0/sdisk600s7 □ /dev/ios0/rsdisk600s7 □ /mnt vxfs□ □0□ □yes□ □rw

then the commands

# **mount /mnt**

# **mount /dev/ios0/sdisk600s7**

# **mount /dev/ios0/sdisk600s7 /mnt**

result in the generation of the following command:

# **mount /dev/ios0/sdisk600s7 /mnt**

The specific *mount* command is then called as:

# **mount /dev/ios0/sdisk600s7 /mnt**

Notice that the *-F* argument is not passed to the specific *mount* command.

Since *mount* accesses and changes the */etc/mnttab* file to reflect the current set of mounted file systems, both the generic and the specific *mount* commands have locking mechanisms. The generic command creates a lock file called */etc/.mnt.lock* and then uses record locking to lock it. The VxFS-specific *mount* command record locks the */etc/mnttab* file.

8.13 ncheck(1M)

The *ncheck*(1M) utility generates a list of pathnames versus inode numbers for files that match specified criteria. The *ncheck* utility can be used for inode versus pathname cross-reference checks in a file system. Used without selection criteria, the command prints an exhaustive list of inodes with their associated file names. Optionally, certain selection criteria can be chosen which finds the inode/pathname pairs by inode, by block number (VxFS-specific *ncheck* only), or according to whether the inode represents a block/character special file or has its setuid/setgid mode bits on. With these selection criteria *ncheck* is useful for tracking down file system problems and security violations.

For example, if a structural *fsck* of a file system uncovers problems with a specific inode, the pathname can be determined by using *ncheck*. Likewise, if a specific block on the file system is noted as bad, the file name can be found through use of *ncheck*.

Printing is done one inode/pathname pair to a line. Each inode/pathname pair is tab-separated on the line with the inode preceding the pathname. Directories are distinguished by a "." character as the last pathname component, for example:

```
1345	/usr/src/cmd/.
```

When the file system structure is inconsistent, "???" denotes the parent of a parentless file and a pathname beginning with "..." denotes a loop. A pathname beginning with "***" denotes a directory entry whose nominal ".." entry is not in accord with the directory in which it was found. When not supplied with a specific list of special devices to check, the generic utility attempts to generate the cross-reference list for each special device in the *fsckpass* field located in the */etc/vfstab* file. Additional information for this process can be found in the manual page for *ncheck*(1M).

When provided with a list of special files, the generic *ncheck* uses the *FSType* provided as the option argument to *-F* as the indicator of which file system-specific command to invoke. If no *-F* is specified, the generic command attempts to determine the FSType of the particular device(s) from the */etc/vfstab*. Once the appropriate type-specific version of *ncheck* is determined, it is run against the named device.

The VxFS-specific *ncheck* command assures that the specified device file is truly a VxFS file system by checking its *VX_MAGIC*, *VX_CHECKSUM*, and *VX_VERSION* in the superblock (The format of the superblock is specified in the man pages at *fs*(4)).

Synopsis

ncheck -F vxfs [**generic_options**] [**-o m**, **b**=n] special...

Command options

-o *m* Will additionally print mode information.Output example:

mode 40755 uid 0 gid 0 ino 3 /lost+found/.

-o b=*n*

Will print the inode/pathname pairs for just those inodes that contain the blocks specified. Multiple block number searches can be contained in one invocation of the command by using multiple, comma-separated *b=n* arguments to *-o*, for example:

```
ncheck -o b=571,b=678,b=1245 /dev/vol/test1
```

8.14 volcopy(1M)

The *volcopy* utility is used to make copies of a file system. The main advantage of *volcopy*(1M) over the *dd*(1) command, is that *volcopy* supports operation over multiple tape reels. This includes support for labeled tapes (See *labelit*(1M) for a description of *volcopy* format labeling.).

Synopsis

volcopy [**-F** vxfs] [**generic_options**] [**current_options**]\

[**-a**] fsname srcdevice volname1 destdevice volname2

Command options

-F This is the current file system type.

For copying VxFS file systems, the *-F vxfs* argument is necessary or the copy will not proceed and the following message is displayed:

vxfs volcopy: /dev/ios0/special_file is not a vxfs filesystem 0 reel(s) completed

where /dev/rdsk/special_file is the name of the source device for the copy.

-a Invokes a verification sequence requiring a positive operator response instead of the standard 10-second delay before the copy is made.

fsname

The mounted name of the file system being copied

srcdevice

The device from which the copy of the file system is being extracted.

volname1

The volume from which the copy of the file system is being extracted.

destdevice

The target device to which the file system will be written.

volname2

The target volume to which the file system will be written.

The current suboptions are supported by the VxFS-dependent *volcopy*:

-e Tells *volcopy* not to try to calculate the number of blocks to put on tape but to wait until the end of tape mark is reached and then switch to a new tape.

-feet

If a target device is a tape drive, then the multiple tape support requires that the user provide the program with the length of the tape. This can either be done using the *-feet* command line option, or else the length is prompted for before the copy is started.

-bpi

The tape density is required by the program and should be selected from one of 800, 1600, or 6250 bits per inch. This can also be supplied on the command line using the *-bpi* option.

-buf

Provides a double buffering facility that can provide some parallelization of the reading and writing of the file system. This may save time on some systems.

-y These override conditions can be overridden by just typing **y** or all override conditions can be overridden by including the *-y* flag on the command line.

-s Calls for the default (DEL if wrong) 10 second delay.

-reel

Specifies the reel.

-r Specifies the decimal number of 512-byte physical blocks for each raw disk I/O transfer.

-block

Specifies the decimal number of 512-byte physical blocks to perform as each single I/O transfer.

-nosh

Escape to shell is not possible.

The *volcopy* utility works only with character-special files. The VxFS-specific *volcopy* determines if the source or target device is a tape drive by virtue of its name containing either *rmt*, or *rtp*. If a source device is a tape device, when the end of each input tape is reached, *volcopy* pauses and prompts the user as follows:

Changing drives? (type RETURN for no,□
device name (e.g. ´/dev/ios/rstape004´) for yes:

If RETURN is typed, *volcopy* responds with:

Mount tape *N* Type volume-ID when ready:

Where *N* is the sequence number expected to be read next. The *volume-ID* prompt allows the user to specify a different tape volume name for each reel.

When the next reel is mounted and return is pressed, a check of the tapes label is performed to ensure that the correct reel name and sequence number is contained in the tapes label.

If a target device is a tape drive, then the multiple tape support requires that the user provide the program with the length of the tape. This can either be done using the *-feet* command line option, or else the length is prompted for before the copy is started.

The tape density is also required by the program and should be selected from one of 800, 1600, or 6250 bits per inch. This can also be supplied on the command line using the *-bpi* option. With some technologies, the length of the tape may need to be a number other than the actual length of the tape in order to reflect the actual capacity of the tape. For example a cartridge tape drive for the i386 with a tape length of 600 feet should be described to *volcopy* as a tape with a density of 6250 BPI and a length of 2048 feet in order to reflect its actual capacity.

Once the length and density values are known by *volcopy*, the first tape volume name, length and density are displayed along with the total number of reels that are required for the copy.

Several conditions are checked for by *volcopy* before a copy commences that may need to be overridden. The first check is to ensure that a header exists on both the source and target devices and that the file system names on both agree. Next a check is made that the target device contains a file system or file system label that is at least 48 hours older than the source file system. Message is generated if any of these conditions is found.

These conditions can be overridden by just typing *y* or all override conditions can be overridden by including the *-y* flag on the command line. The same messages are still produced, but the output of *YES* replaces the pause for user input.

The *-a* flag causes the 10 second (DEL if wrong) message to be replaced with an explicit (*y* or *n*) prompt and also has the side effect of causing all normally unpredictable prompts for an override to be omitted. This flag, when used in conjunction with the *-y* flag, can be used to set up automated scripts to perform a number of *volcopy* commands with no manual intervention, since no questions requiring a response is asked, and the 10 second delay is also omitted.

Throughout the *volcopy* execution, **DEL** can be typed and the user is prompted if he wants a shell or if he wants to quit from the copy operation.

The *-buf* option provides a double buffering facility that can provide some parallelization of the reading and writing of the file system. This may save time on some systems.

Tapes can be labeled using the *labelit* utility.

8.15 vxdump(1M)

vxdump is a utility for backing up VxFS file systems. It is used in conjunction with the VxFS utility *vxrestore* to provide backup and restore facilities for VxFS file systems. *vxdump* provides for both full and incremental backups.

Synopsis

vxdump [**options**]filesystem

Command options

0- 9

Indicates the level of the backup to be performed. All files in the file system that have been modified since the last *vxdump* at a lower backup level are archived. The information regarding backup dates and levels is kept in the data file */etc/dumpdates*. */etc/dumpdates* is a plain ASCII file and can be edited by users with appropriate privileges, but updates are best left to the programs like *vxdump*.

-b *factor*

Specifies a blocking factor for tape writes. Blocking will be in terms *factor* (512 byte) blocks. See the reference manual pages for defaults and limits on blocking.

-c

Assumes a cartridge tape device. This has consequences for assumptions about blocking and tape density. See the reference manual pages for details.

-d *bpi*

Specifies tape density in *bpi*.

-f *dump-file*

Writes the archive into *dump-file*. The default assumes */dev/rmt8*. If "-" is specified as the *dump-file* the backup is written on *stdout*. This following can be used to pipe directly to the restore program *vxrestore* or to some other program for writing the tape:

```
# vxdump -3uf - /dev/rvol/bkup | vxrestore xf -□
# vxdump -3uf - /dev/rvol/bkup | dd bs=512k \ □
> /dev/ios0/rstape003
```

-M *megabytes*

Maximum amount of data to write to each tape, given in megabytes.

-n Notifies users in the operator group if *vxdump* needs attention.

-s *size*

Specifies the size of the volume to which the backup is being performed. When the specified size is reached, *vxdump* waits for you to change the volume. *vxdump* interprets the specified size as the length in feet for tapes and cartridges, and as the number of 1024-byte blocks for diskettes.

-t *tracks*

Specifies the number of tracks for a cartridge tape. The default is nine tracks.

-u Specifies that *vxdump* should add an entry to the file */etc/dumpdates*, for each file system successfully backed up that includes the file system name, date, and back up level.

-w Lists the file systems that need backing up. This information is taken from the files */etc/dumpdates* and */etc/vfstab*.

-W Similar to the *-w* option, except that the *-W* option includes all file systems that appear in /etc/dumpdates, along with information about their most recent backup dates and levels.

The *vxdump* utility works on simple VxFS file systems but also makes provision for snapshot mounted file systems to support on-line backup. (See Chapter "On-line backup" of this manual for a discussion of VxFS on-line backup facilities.)

In general *vxdump* is issued with options (described in the following) that specify the characteristics of the medium onto which the backup is to go and the character device associated with the file system to be backed up. If *vxdump* determines that the device contains a snapshot of a VxFS file system, it takes appropriate actions to read the device as a snapshot file system. No special provisions are needed for handling a snapshot mounted file system when invoking *vxdump*.

Information about the tape characteristics onto which the backup is written (blocking factor, bpi, number of feet per tape, etc) is used to determine how to write to the tape device. The reference manual pages for *vxdump* give the most detail on how to select options that best fit your media characteristics. In addition to the options mentioned in the preceding, *vxdump* has options to query the */etc/dumpdates* database and to specify how to handle diagnostic messages that may be generated when running the program.

Example

A sample invocation of *vxdump* looks like the following:

```
# vxdump -3uf /dev/ios0/rstape003 /dev/rvol/bkup
```

This example does a level 3 backup onto */dev/ios0/rstape003* and updates the */etc/dumpdates* data file.

In the case where the file system on the */dev/rvol/bkup* is a full VxFS file system, the *vxdump* should be done on the unmounted file system to assure a consistent backup. In the case where a block device associated with */dev/rvol/bkup* is mounted as a snapshot mount of another VxFS file system, the backup can be performed while the primary file system is mounted and active.

After a backup such as the one in the preceding example, a subsequent query such as:

```
# vxdump -W
```

should show this backup as the last done for */dev/rvol/bkup*.

8.16 vxrestore(1M)

vxrestore is a utility for restoring files previously written to an archive by the program *vxdump*. *vxdump* archives files from VxFS file systems. *vxrestore* is only able to read this type of archive, but the restore can be onto any of the standard file system types.

Synopsis

vxrestore [**-irRtxdhmvy**] [**-b** factor] [**-e** extent_op] [**-f** dump_file]\ □
[**-s** n] filename ...

Command options

-i The user is provided with an interactive interface that allows browsing through the backup tapes directory hierarchy and selecting individual files to be extracted. The following section, Interactive commands When the -i option is selected, vxrestore enters interactive mode. Interactive commands are reminiscent of the shell. For those commands that accept an argument, the default is the current directory. The reference manual pages for vxrestore give a complete description of these commands and detail where interactions with other command line options affect behavior., details the commands that are available in this interactive mode.

-r Restores the entire tape into the current directory.

-R *vxrestore* leaves information in the working directory as to which tape it was working on for multivolume restores. In the event of an interruption, this information can used to determine where to resume processing. Use this option when such a resumption is necessary.

-t Lists a table of contents for the archive. This option can be modified using the *-h* option.

-x Extracts the named files from the tape. Directories are extracted recursively unless *-h* is used. If no *filename* argument is given, the root directory is extracted. This results in the entire tape being extracted unless the *-h* modifier is in effect.

-d Turn on debugging output.

-h Extracts the actual directory, rather than the files that it references in order to prevent hierarchical restoration of complete subtrees from the tape.

-m Extracts by inode numbers rather than by filename to avoid regenerating complete pathnames. This is useful if only a few files are being extracted.

-v *vxrestore* displays the name of each file it restores, preceded by its file type.

-y Do not ask whether to abort the restore in the event of tape errors. *vxrestore* tries to skip over the bad tape blocks and continue as best it can.

-b *factor* Specify the blocking factor for tape reads. By default, *vxrestore* attempts to determine the block size of the tape; a tape block is 512 bytes.

-e *extent_op* Specifies how to handle extent attribute information. This option specifies the required persistence of extent attributes when restoring files that had preallocated space or fixed extent sizes. Valid values for *extent_op* are:

warn
issues a warning message if extent attribute information cannot be kept (the default behavior if this option is not specified)

force
fails the file restore if extent attribute information cannot be kept

ignore
ignores extent attribute information

-f *dump_file* Redirects *vxrestore* to use the dump-file. By default it uses */dev/rmt8* as the file to restore from. "-" can be used to force reading from the standard input. Using the "-" option allows *vxdump*(1M) and *vxrestore* to be used in a pipeline to backup and restore a file system:

```
# vxdump 0f - /dev/bkup | (cd /mnt; vxrestore xf -)
```

-s *n* Skip to the *n*th file when there are multiple backup files on the same tape. For example, the command:

```
# vxrestore xfs /dev/bkup 5
```

would position you at the fifth file on the tape.

Interactive commands

When the *-i* option is selected, *vxrestore* enters interactive mode. Interactive commands are reminiscent of the shell. For those commands that accept an argument, the default is the current directory. The reference manual pages for *vxrestore* give a complete description of these commands and detail where interactions with other command line options affect behavior.

ls [directory] List files in a specified directory or the current directory.

cd [directory] Change to the named directory directory within the archive hierarchy.

pwd Print the full pathname of the current working directory.

add [filename] Add the current directory, or the named file or the specified directory to the list of files to extract.

delete [filename] Delete the current directory, or the named file or directory from the list of files to extract.

Extract all files on the extraction list from the archive tape.

extract Toggle the status of the v modifier.

verbose Display a summary of the available commands.

help Exit vxrestore immediately, even if the extraction list is not empty.

vxrestore is a derived from the *ufsrestore* program that is part of the standard distribution of the UNIX system. *vxrestore* provides the same facilities as *ufsrestore*, including an interactive mode that allows browsing through the directory hierarchy of the archive and selecting individual files for extraction.

Extent attribute preservation

vxrestore provides for the preservation of extent attributes of files in its source archive.

As described in Section "Extent-based allocation", the VxFS file system allocates disk space to files in extents or groups of one or more adjacent blocks. The VxFS file system also defines an application interface that allows programs to control some aspects of the extent allocation that take place for a given file. The extent allocation controls associated with a file are referred to as the **extent attributes** of the file.

The archives created by *vxdump* contain the extent attribute information for the files in the source file system. *vxrestore* uses this information to recreate the source files with the same attributes, provided the restore is done onto a VxFS file system. These attributes are lost if the files are restored onto a file system that is not a VxFS file system. The *-e* option controls how the extent attributes of files in the archive are to be handled by *vxrestore*.

9 Messages

9.1 Messages of the administrative commands

The following diagnostic or error messages may appear when using the various administrative commands and utilities. Messages which may appear while using different commands are summarized in Section "Miscellaneous messages". User action is included to assist in solving system problems, or to correct invalid input.

9.1.1 Miscellaneous messages

The messages in this section may appear when using different commands. Instead of *command* the name of the command will be displayed.

Message
vxfs *command*: fs_open: internal error

There is an internal error in the command that is calling *fs_open*.

This should be reported as a bug to your customer support organization. Try a different invocation to see if it gets the same error.

Message
vxfs *command*: open of *special* failed: *reason*

vxfs *command*: stat of *special* failed: *reason*

open(2) or *stat*(2) failed for the specified block or character special file. *reason* is the error returned by the system call.

Verify that *special* exists, is a block or character special file, is available for reading (and writing, if appropriate), and that the user has appropriate privileges to open the file.

Message
vxfs *command*: malloc of space for superblock failed

Dynamic memory allocation failed.

Check the system memory usage. If the system is low on memory, try reducing memory usage or adding swap space. If the system isn't low on memory, check the process size against the maximum process size and increase the maximum if necessary.

Message
vxfs command: read of superblock on *special* failed: *error_msg*

A disk read of the superblock from the file *special* failed completely with the error described by *error_msg*. This can occur if there is a disk error such that none of the read can be satisfied. More commonly, this will occur if a read past the end of the disk is attempted, or the device is offline for some reason.

Check *special* and the status of the associated device to be sure it's online. Check the console log for possible I/O failure messages. If there was a device failure then diagnose and repair the problem. If the device contains a VxFS file system, then schedule a full file system check at the next administrative period. Information for how to perform a full file system check is located in the Section "fsck(1M)", and in the *fsck* manual page.

Message
vxfs *command*: invalid vxfs superblock magic: *special*

vxfs *command*: unrecognized vxfs version number: *special*

vxfs *command*: invalid vxfs superblock checksum: *special*

When examining the superblock of a VxFS file system, invalid data was found.

Check that the *special* has been specified correctly and that the file system is a VxFS file system. If the *special* is a snapshot file system, the superblock may have been read using the *VX_SNAPREAD ioctl* on the mountpoint where *special* is mounted and the data in the superblock may be different from that at the beginning of *special*. If the command is invoked correctly and *special* is not a snapshot file system, run *fsck* with a full file system check at once. Information for how to perform a full file system check is located in the Section "fsck(1M)", and in the *fsck* manual page.

Message
vxfs *command*: *special* is not a block or character special file

special contains a snapshot file system, but is not a block or character special file. When invoked on a snapshot file system, the command requires that *special* is the block or character special file of the mounted file system. Since *special* is neither, it is impossible for the command to determine the mountpoint of the snapshot.

Verify the command is being used correctly and that *special* was correctly specified.

Message
vxfs *command*: snapshot mountpoint for *special* (*block_special*) not found

When invoked on a snapshot file system, the command requires to be able to identify the mountpoint of the snapshot file system in order to correctly access the data. *special* contains a snapshot file system, but the mountpoint for the snapshot file system cannot be determine by examining */etc/mnttab*. If *special* is a character special file the name was converted to *block_special*, the corresponding block special file, before searching */etc/mnttab*.

Verify the command is being used correctly. If *special* is a snapshot file system, it must be mounted for the command to work. If *special* is a character special file, try reinvoking the command on the corresponding block special file used when the snapshot was mounted.

Message
vxfs *command*: file system *mount_point* for *special* (*block_special*) not snapshot file system

When invoked on a snapshot file system, the command requires to be able to identify the mountpoint of the snapshot file system in order to correctly access the data. *special* contains a snapshot file system and *mount_point* is the mountpoint for *special* deduced from reading */etc/mnttab*, but *mount_point* is not a snapshot file system or doesn't match data in *special*. If *special* is a character special file the name was converted to *block_special*, the corresponding block special file, before searching */etc/mnttab*.

Verify the command is being used correctly. If *special* is a character special file, try reinvoking the command on the corresponding block special file used when the snapshot was mounted. The contents of */etc/mnttab* may be incorrect, or the snapshot file system at *mount_point* may have been disabled due to an I/O error or running out of space. Check the console log for I/O errors or file system disabled messages.

Message
vxfs *command*: snapshot read of superblock on *mount_point* (special) failed: *error_msg*

mount_point has been deduced to be the mountpoint of the snapshot file system on *special* based on information in */etc/mnttab*. An attempt to use the *VX_SNAPREAD ioctl* on the file system at *mount_point* failed with an error which is explained by *error_msg*. If *special* is a character special file the name was converted to the corresponding block special file before searching */etc/mnttab*.

The snapshot file system may have become disabled due to an I/O error or running out of space for changed blocks. Check the console log for I/O errors or file system disabled messages. Verify that the file system at *mount_point* is accessible and that the */etc/mnttab* entries are correct.

Message
vxfs *command*: fs_datalloc() failed

The call to *fs_datalloc* failed, probably due to a failure in *malloc*.

Check the system memory usage. If the system is low on memory, try reducing memory usage or adding swap space. If the system isn't low on memory, check the process size against the maximum process size and increase the maximum if necessary.

Message
vxfs *command*: getblk can't malloc *size* bytes

Buffer space could not be allocated.

Check to see that there is sufficient swap space with *swap*(1M)

Message
vxfs *command*: getblk read fail bno *bno*

A read request at file system block *bno* failed.

Check console log for possible I/O error.

Message
vxfs *command*: getblk write fail bno *bno*

vxfs *command*: putblk write fail bno *bno*

A write request at file system block *bno* failed.

Check console log for possible I/O error.

9.1.2 Messages of getext(1)

Message
vxfs getext: cannot open *filename* - *perror_string*

The open of the file named on the command line failed. See the *open*(2) manual page for reasons that *open* can fail. *perror_string* contains a description of the error.

Verify the existence of the named file. Check permissions on the source file.

Message
vxfs getext: cannot statvfs *filename* - *perror_string*

The *statvfs* of the file named on the command line failed. See the *statvfs*(2) man page for reasons that it can fail. *perror_string* contains a description of the error.

Verify the existence of the named file. Check permissions on the source file.

Message
vxfs getext: *filename* is not on a vxfs file system.

getext uses the *statvfs* system call on the named file to determine the type of the file system on which it resides as well as the block size of that file system. This message indicates that *getext* discovered that the named file doesn't reside on a VxFS file system.

The *getext* command cannot be used on this file.

Message
vxfs getext: cannot get attributes for *filename* - *perror_string*

The *ioctl* call which is used to get the attribute information for the named file has failed. This can only happen if the named file is not a regular file. *perror_string* contains a description of the error.

Check the command line.

9.1.3 Messages of setext(1)

Message
vxfs setext: cannot open *filename* - *perror_string*

The *open* of the file named on the command line failed. See the *open*(2) manual page for reasons that *open* can fail. *perror_string* contains a description of the error.

Verify the existence of the named file. Check permissions on the source file.

Message
vxfs setext: cannot statvfs *filename* - *perror_string*

The *statvfs* of the file named on the command line failed. See the *statvfs*(2) manual page for reasons that it can fail.

Verify the existence of the named file. Check permissions on the source file.

Message
vxfs setext: *filename* is not on a vxfs file system.

setext uses the *statvfs* system call on the named file to determine the *FStype* of the file system on which it resides. This message indicates that *setext* has discovered that the named file doesn't reside on a VxFS file system.

The *setext* command cannot be used on this file.

Message
vxfs setext: cannot set attributes for *filename* - *perror_string*

The *ioctl* call, which is used to set the attribute information for the named file has failed. This failure will occur if there is a mismatch of flags and options.

This can happen also if the named file is not a regular file, or if the reservation specified for a *-r* option is larger than would be allowed by the current *ulimit*, or if there is insufficient space on the file system, or if *chgsize* is specified and the user is not the super user.

Check the command line, *ulimit* and the file type.

9.1.4 Messages of df(1)

Message
vxfs df: cannot stat *directory*|*special*

The device cannot be stated, The possible causes are:

- *df* does not have permissions to read and search the directories in the path of the device.
- The device is off-line.
- The */etc/mnttab* file has been corrupted.

The message that displays indicates why the error has occurred. User action is dependent upon the displayed error message.

Message
vxfs df: cannot statvfs *path reason*

The */etc/mnttab* file indicated that the file system was mounted, but the call to *statvfs*(2) to get the resource statistics failed. There are several reasons why this error might be displayed:

- *df* does not have permissions to do the *statvfs*.
- There is a problem on the file system containing the pathname.
- */etc/mnttab* is corrupted.

The message that displays indicates why the error has occurred. User action is dependent upon the displayed error message.

9.1.5 Messages of ff(1M)

Message
vxfs ff: n option *perror_diagnostic*

If the option argument to *-n* was not a valid file or couldn't be *stat*(2) ed, *ff* issues this message using *perror*(3C) to attempt to provide an explanation for the failure.

Check the command line.

Message
vxfs ff: *device* NOTE! *number* pathnames only partially resolved

If file system corruption is detected (for example, loops in the paths), this diagnostic will be printed.

Schedule a full file system check at the next administrative period. Information for how to perform a full file system check is located in the Section "fsck(1M)", and in the *fsck* manual page.

Message
vxfs ff: No files were selected.

None of the inodes on the file system satisfied the search criteria provided.

None.

Message
vxfs ff: cannot malloc link space

vxfs ff: cannot malloc inode space

vxfs ff: cannot malloc directory map space

vxfs ff: insufficient memory for directory buffer

vxfs ff: calloc failed in inodes()

Dynamic memory allocation failed.

If the *ff* command is selecting a lot of files, try to break up the command into multiple invocations that each do part of the selection. If that fails, check the system memory usage. If the system is low on memory, try reducing the memory usage or adding swap space. If the system is not low on memory, check the process size against the maximum process size and increase the maximum if necessary.

Message
vxfs ff: *number* files selected

ff prints the total number of files satisfying search constraints on the *stderr*. (The filenames/inode numbers, etc, are printed on *stdout*).

None.

Message
vxfs ff: *number* link names detected

When *-l* is selected, this information is also printed on *stderr*.

None.

Message
vxfs ff: directory block *number* in ino *number* is bad

vxfs ff: directory block *number* for ino *number* is bad

vxfs ff: immediate directory block in ino *number* is bad

File system corruption discovered.

Try running the command again. If the message isn't repeated, then file system activity was the cause of the problem. If the message is repeated, schedule a full file system check at the next administrative period. Information for how to perform a full file system check is located in the Section "fsck(1M)", and in the *fsck* manual page.

Message
vxfs ff: parent chain for inode *number* is too long

ff internal limit on the acceptable depth of pathnames is exceeded.

Try running the command again. If the message isn't repeated, then file system activity was the cause of the problem. If the message is repeated and it is necessary to find the full pathname of the file, use the *find*, *ls*, and *grep* commands to find the file by searching through the file tree.

Message
vxfs ff: read failed in getblk()

Read of a particular file system block failed.

Schedule a full file system check at the next administrative period. Information for how to perform a full file system check is located in the Section "fsck(1M)", and in the *fsck* manual page.

Message
vxfs ff: -i value, *number*, ignored

Invalid inode number discovered in the *ilist* option argument to *-i*.

Check the command line.

Message
vxfs ff: program limit of *number* exceeded for *name*

ff operations have overflowed the bounds of the given internal resource. For example, *ff* when the *-l* option is specified, all links that are discovered in a table will be tracked. If the subject file system has more links than will fit into this table, this diagnostic is printed.

Try breaking the command up into multiple invocations of *ff* so each invocation does a subset of the selection desired.

Message
vxfs ff: *special_file* - *reason*

An error occurred and *reason* is the error returned. *ff* issues this message using *perror*(3C) to attempt to provide an explanation for the failure.

Verify that *special* exists, is a block or character special file, is available for reading (and writing, if appropriate), and that the user has appropriate privileges to open the file.

9.1.6 Messages of fsadm(1M)

Message
vxfs fsadm: [-b] option may not be used with [-DEde] options

Reorganization and resizing cannot be performed together.

If you want to reorganize a file system and to resize it as well, please keep to the order as follows:

- Reorganizing the file system and increasing its size:

  The file system is to be increased in size first, since the added space offers greater flexibility to the reorganization routines.

- Reorganizing the file system and reducing its size:

  Do the reorganization first to get free resources which are required to run the process of reduction.

Message
vxfs fsadm: filesystem *mount_point* doesn
′t match filesystem superblock on *special*

vxfs fsadm: use [-r] to specify a raw device

The data associated with the file system mounted at *mount_point* doesn't match the data in the superblock on the special file *special* indicating that *special* is not the file system mounted at *mount_point*.

If the *-r* option was used to specify the raw device then the wrong device was specified. Check the command invocation. If the *-r* option was not used then */etc/vfstab* contains an incorrect entry for the file system or the heuristics used by *fsadm* to convert a block special pathname to the corresponding character special yield the incorrect result. Use the *-r* option to specify the raw device.

Message
vxfs fsadm: *mount_point* is not a mount_point in */etc/vfstab* and the raw device cannot be deduced

vxfs fsadm: use [-r] to specify a raw device

The specified mountpoint was not in */etc/vfstab*, the heuristics to deduce the raw device path based on the block device path failed, and *fsadm* could not determine the pathname of the raw device.

Use the *-r* option to specify the raw device.

Message
vxfs fsadm: cannot stat *mount_point*

vxfs fsadm: cannot statvfs *mount_point*

Could not *stat*(2) the mountpoint.

Check the command line and try *mount*(1M). If *mount_point* is correct check permissions of mountpoint and calling process.

Message
vxfs fsadm: *mount_point* is not the root inode of a vxfs file system

vxfs fsadm: *mount_point* is not a vxfs file system

Either the file system is not a VxFS file system, or a mountpoint was not specified.

Check the command line.

Message
vxfs fsadm: *mount_point* is mounted read only

The file system is mounted read only.

Remount the file system as read/write.

Message
vxfs fsadm: cannot read superblock of *raw_device*

The superblock of the raw device could not be read. Either the device is offline or there is a permissions problem.

Verify that the permissions of the raw device are open enough for access and that the device is online.

Message
vxfs fsadm: *raw_device* has unknown file system magic

Either the file system is not VxFS or the superblock has been corrupted.

Try running *fstyp*(1M). Check */etc/vfstab*.

Message
vxfs fsadm: cannot open *lockfile*

The *fsadm*(1M) utility uses a lock file to ensure that only one administrative function is currently active. The lock file could not be opened.

Check to see that the *lost+found* directory is present. Check the console log.

Message
vxfs fsadm: cannot lock *lockfile*

The *fsadm*(1M) utility uses a lock file to ensure that only one administrative function is currently active. The lock file could not be locked.

Check the permissions of calling process.

Message
vxfs fsadm: cannot lock *lockfile*

vxfs fsadm: lock for *lockfile* is held by pid *pid*

Another process is running *fsadm*(1M).

Check to see why the other instance of *fsadm*(1M) is running.

Message
vxfs fsadm: write failure at block *block_number*

vxfs fsadm: expected to write *expected*, wrote *actual*

vxfs fsadm: seek error at block *block_number*

vxfs fsadm: read failure at block *block_number*

vxfs fsadm: expected to read *expected*, read *actual*

An I/O failure has occurred.

Check the console log.

Message
vxfs fsadm: file system overhead is *overhead* sectors - cannot shrink to *size*

File system overhead includes the superblock, log, resource maps and inode list structures. The requested size represents an attempt to remove blocks from the file system that are used by these vital structures.

The requested size must be increased. If a smaller file system is needed it will have to be rebuilt. See *mkfs* (1M) for details.

Message
vxfs fsadm: *mount_point* is currently *size* sectors

The requested size represents an attempt to resize the file system to its current size – neither expanding nor shrinking it.

Check command line or choose a different size.

Message
vxfs fsadm: cannot shrink *mount_point* - inodes/blocks are currently in use.

The requested size represents an attempt to shrink the file system, however, resources in the area to be "shrunk out" of the filesystem are currently in use.

File system reorganization will be needed to try and free up the resources that prevent shrinking. Reorganization compresses blocks within an allocation unit, freeing blocks at the end of the file system which allows the file system to be shrunk. Inodes cannot be moved, so if they are allocated within the allocation unit, the file system cannot be shrunk below that allocation unit; allocation nodes within the allocation unit must be found and the files moved.

Message
vxfs fsadm: cannot malloc map space

vxfs fsadm: cannot realloc map space

Space to hold the image of a resource map could not be allocated.

Check to see that there is sufficient swap space using *swap*(1M)

Message
vxfs fsadm: cannot read emap for au *number*

vxfs fsadm: cannot read imap for au *number*

An I/O failure has occurred.

Check the console log.

Message
vxfs fsadm: attempt to resize *raw_device* failed with errno *errno*

A resize failed.

Check the permissions, command line, and console log.

Message
vxfs fsadm: dirsort failed inode *ino*, errno *errno*

A directory reorganization request failed for a reason other than activity during the reorganization. This message indicates a possible permissions problem, or I/O failures.

Check the permissions and the console log.

Message
vxfs fsadm: directory *ino* block *bno* offset *off* has bad reclen *len*

vxfs fsadm: directory *ino* block *bno* offset *off* has bad namlen *len*

A directory entry has an invalid header. This is most probably because the directory was in transition, if this message occurs every time *fsadm* is run, then the error may be due to file system damage.

Schedule a full file system check at the next administrative period.

Message
vxfs fsadm: cannot malloc extent table

Space to hold the list of extents for an allocation unit could not be allocated.

Check to see that there is sufficient swap space using *swap*(1M).

Message
vxfs fsadm: file system *mount_point* not in /etc/mnttab

The file system specified by *mount_point* could not be found in */etc/mnttab*.

Check the command invocation. Check the permissions on */etc/mnttab* and the processes permissions and have the system administrator change them if incorrect. Check the contents of */etc/mnttab* by running *mount* and verifying that *mount_point* appears in the output. If not, the file system is either unmounted, or there is an error in */etc/mnttab*.

9.1.7 Messages of fscat(1M)

Message
vxfs fscat: arg '*offset*' must evaluate to a valid offset

When the *-o* option is used, the specified offset must be non-negative, but the value specified evaluated to *offset*, which is negative.

Check the command line to verify the value for *offset* is valid.

Message
vxfs fscat: offset or length exceeds file system size

The *-o* and/or the *-l* options were used to specify the offset that *fscat* should start at or the number of bytes it should cat. Either offset is larger then the size of the file system or offset + length is larger than the size of the file system.

Review the command line options used and compare them to the size of the file system. Consider a length of 0 if you want to cat the entire file system.

Message
vxfs fscat: read error at offset *offset* on *special*

A disk read from the file *special* failed. Commonly, this will occur if a read past the end of the disk is attempted, or the device is offline for some reason. If the file system corresponding to *special* is a snapshot file system, the read error was actually returned from the *VX_SNAPREAD ioctl*(2) and is probably accompanied by a message on the system console.

Check *special* and the status of the associated device to be sure it's online. Check the console log for possible I/O failure messages or snapshot file system errors. If there was a device failure then diagnose and repair the problem. If the device contains a VxFS file system, then schedule a full file system check at the next administrative period. Information for how to perform a full file system check is located in the Section "fsck(1M)", and in the *fsck* manual page.

Message
vxfs fscat: write error on output

A *write*(2) system call failed. This may be due to an error on a device, a file system running out of space for the output file, or exceeding the *ulimit*(2) for the process.

If the output has been redirected to a device, verify the device has enough space on it to hold the output. If the output is to a file, verify the file system has enough space to hold the output and that the output doesn't exceed *ulimit*(2) for the process. Check the console log for possible I/O failure messages.

Message
vxfs fscat: malloc of space for superblock failed

vxfs fscat: unable to malloc *size* byte buffer

Dynamic memory allocation failed. In the case of the second message, an allocation of *size* bytes failed.

Check the system memory usage. If the system is low on memory, try reducing memory usage or adding swap space. If the system isn't low on memory, check the process size against the maximum process size and increase the maximum if necessary.

9.1.8 Messages of fsck(1M)

The following messages and diagnostics of *fsck*(1M) are separated by the different phases where they occur. Refer to the following sections:

"Messages during initiation phase" ...,

"Messages from pass 1" ...,

"Messages from pass 2" ...,

"Messages from pass 3" ...,

"Messages from pass 4" ....

9.1.8.1 Messages during initiation phase

Message
vxfs fsck: cannot read superblock of *special*

A disk read of the superblock from the file *special* failed to return sufficient data. This can occur if there is a disk error such that the read cannot be satisfied. More commonly this will occur if the device is offline for some reason. If the *special* is a snapshot file system, it may be disabled causing the read to fail.

Check *special* and the status of the associated device to be sure it's online. Check the console log for possible I/O failure or file system disabled messages. If there was a device failure then diagnose and repair the problem. If a snapshot file became disabled, unmount and remount it with more space available.

Message
vxfs fsck: intent log marked bad in superblock

vxfs fsck: superblock indicates that intent logging was disabled

vxfs fsck: cannot perform log replay

The superblock for the file system indicates that the intent log portion of the file system has had an I/O failure. Log replay will not be performed

A full file system check is required.

Message
vxfs fsck: a full file system check is required

Log replay has been completed, or could not be run, and the file system is marked as needing a full file system check. If the log replay fails, it is generally due to I/O failure. Other messages may appear to clarify the failure.

A full file system check is required.

Message
vxfs fsck: a full file system check may be required

Log replay has been completed, or could not be run, and the file system is not marked as CLEAN. If the log replay fails, it is generally due to I/O failure. Other messages may appear to clarify the failure.

A full file system check is required.

Message
vxfs fsck: invalid superblock magic *magic*

vxfs fsck: invalid superblock version *version*

vxfs fsck: invalid superblock checksum

vxfs fsck: invalid superblock flags *flags*

When checking the superblock, invalid data was found. If performing log replay, *fsck*(1M) will terminate.

Check that the device has been specified correctly and that the file system is a VxFS file system. If the specified information is correct, rerun *fsck* with a full file system check.

Message
vxfs fsck: rebuild from auxiliary? (ynq)

When checking the superblock, invalid data was found. It is possible to rebuild the superblock from the first allocation unit header.

y: Rebuild from super auxiliary.

Message
vxfs fsck: alternate superblock not found

An alternate superblock was not found. This message indicates that there has been substantial file system corruption.

The file system will need to be restored from backups.

Message
vxfs fsck: incomplete resize discovered - restoring alternate superblocks

vxfs fsck: incomplete resize discovered - restore alternate superblocks? (ynq)

There was a system failure during a resize operation (see *fsadm*(1M) for details of resize operation). The alternate copies of the superblock within the allocation unit headers, which are updated with the new file system size before the superblock, must be restored.

y: Restore the auxiliary superblocks.

Message
vxfs fsck: au header *aun* invalid magic *magic in hex*

vxfs fsck: invalid auxiliary superblock checksum in au *aun*

vxfs fsck: invalid auxiliary superblock create time in au *aun*

vxfs fsck: invalid au header number aun found au_aun

When checking allocation unit headers, invalid data was discovered. If performing a log replay, the log replay terminates.

Rerun *fsck* with a full file system check.

Message
vxfs fsck: rebuild from superblock (ynq)?

When checking allocation unit headers, invalid data was discovered. It is possible to rebuild an allocation unit header from the superblock of the file system.

y: Rebuild from superblock.

Message
vxfs fsck: a delayed write error has occurred

restarting replay using synchronous write

there may be duplicates of messages from the first pass of replay

For performance reasons, intent log replay buffers write operations. When a write error occurs, the log replay operation is restarted, using nonbuffered write operations, to ensure that the proper sequence of updates occurs. The file system will be marked as requiring a full *fsck*. Pertinent messages will display after the log replay is complete to alert you to the fact that a full *fsck* will need to be run.

Check the console log for the location of the write error.

Message
vxfs fsck: cannot mark inode *ino* bad

When an inode is found to be invalid, the inode is marked as bad to ensure that the inode cannot be accessed. The write to inode block failed.

A full file system check needs to be run. Check the console log for the location of the write error.

Message
vxfs fsck: Replay complete - marking superblock as CLEAN

The intent log replay has successfully completed. The superblock is marked CLEAN to inform *mount*(1M) that the file system may be safely mounted.

None.

Message
vxfs fsck: cannot malloc au buffer

vxfs fsck: cannot malloc au summary space

vxfs fsck: cannot malloc inode map space

The *fsck*(1M) utility uses *malloc*(3C) to dynamically allocate work space. The attempt failed.

Check to see that there is sufficient swap space with *swap*(1M).

Message
vxfs fsck: read failure at *off*

vxfs fsck: cannot read au header at block *bno*

A read request has failed.

Check the console log for possible error.

Message
vxfs fsck: cannot update superblock

vxfs fsck: cannot write au header at block *bno*

A write request has failed.

Check console log for possible error.

Message
vxfs fsck: can't open checklist file: /etc/vfstab

Attempt to open */etc/vfstab* failed. Either the file has been removed, or the command does not have permissions to read the file.

Check the existence of */etc/vfstab* and the permissions of the file and calling process.

Message
vxfs fsck: sanity check failed: cannot stat *device*

The *stat*(2) of the device failed.

Check the command line for possible invocation errors (invalid device name). Check the console log for possible messages.

Message
vxfs fsck: sanity check: *device* already mounted
vxfs fsck: sanity check: *device* already mounted on *mount_point*

vxfs fsck: sanity check: *device* already mounted as *block_special* on *mount_point*

fsck is attempting to check a mounted file system, based on information in */etc/mnttab*.

Check the command line. File systems must be unmounted to be repaired. In the second case, *device* is a character special file that was converted to the corresponding block special file (*block_special*) before searching */etc/mnttab*.

Message
vxfs fsck: sanity check: *device* needs checking

The file system is not CLEAN.

A file system check is required.

Message
vxfs fsck: sanity check: *device* OK

The file system is CLEAN.

None.

Message
vxfs fsck: cannot malloc transaction space

vxfs fsck: cannot malloc transaction status space vxfs fsck: cannot malloc directory action space

vxfs fsck: cannot malloc inode action space

vxfs fsck: cannot malloc logged data status space

The intent log replay uses *malloc*(2) to dynamically allocate work space. The attempt failed. The *fsck* utility will exit.

Check to see that there is sufficient swap space with *swap*(1M).

Message
vxfs fsck: Inode modification for invalid inode number *inode*

An intent log entry for an inode modification request specified an invalid inode number. Either the intent log has been corrupted or there is a serious software error. The intent log replay will terminate.

If there has been accidental corruption, run a full file system check. If this problem reoccurs, contact your service representative.

Message
vxfs fsck: inode *number* is marked bad

An intent log entry for an inode modification request specified an inode which was discoved to have been marked bad by the kernel.

A full file system check will be required.

Message
vxfs fsck: impossible log function *function* in undo

An invalid intent log function code was found. Either the intent log has been corrupted or there is a serious software error. The intent log replay will terminate.

If there has been accidental corruption, run a full file system check. If this problem reoccurs, contact your service representative.

Message
vxfs fsck: bad inode operation *op* inode *ino*

vxfs fsck: bad maptype in inode map operation op = *op* maptype = *typ* inode *ino*

vxfs fsck: bad inode map operation op = *op* maptype = *typ* inode *ino*

An intent log entry for a free inode map or extent inode operation map update contained an invalid entry. Either the intent log has been corrupted or there is a serious software error. The intent log replay will terminate.

If there has been accidental corruption, run a full file system check. If this problem reoccurs, contact your service representative.

Message
vxfs fsck: bad indirect truncation inode *ino*

vxfs fsck: bad indirect truncation level *level*

vxfs fsck: bad indirect truncation block *bno*

vxfs fsck: invalid indirect extent truncation offset *off* extent *bno*

An intent log entry to remove an entry from an indirect address extent has invalid data. Either the intent log has been corrupted or there is a serious software error. The intent log replay will terminate.

If there has been accidental corruption, run a full file system check. If this problem reoccurs, contact your service representative.

Message
vxfs fsck: bad indirect allocation inode *ino*

vxfs fsck: bad indirect allocation level *level*
vxfs fsck: bad indirect allocation block *bno*

vxfs fsck: invalid indirect extent allocation index *index* extent *bno*

An intent log entry that attempts to add an entry to an indirect address extent has invalid data. Either the intent log has been corrupted or there is a serious software error. The intent log replay will terminate.

If there has been accidental corruption, run a full file system check. If this problem reoccurs, contact your service representative.

Message
vxfs fsck: bad file offset in logged write replay: *bno* inode *ino*

vxfs fsck: bad block in logged write replay: *bno* inode *ino*

vxfs fsck: bad indirect block in logged write replay: *bno* inode *ino*

An intent log entry that attempts to write logged data has invalid data. Either the intent log has been corrupted or there is a serious software error. The intent log replay will terminate.

If there has been accidental corruption, run a full file system check. If this problem reoccurs, contact your service representative.

Message
vxfs fsck: bad directory removal inode *ino*

vxfs fsck: bad directory removal block *bno*

vxfs fsck: bad directory removal entry, block *bno*

An intent log entry that attempts to remove an entry from a directory has invalid data. Either the intent log has been corrupted or there is a serious software error. The intent log replay will terminate.

If there has been accidental corruption, run a full file system check. If this problem reoccurs, contact your service representative.

Message
vxfs fsck: directory block *bno* in inode *ino* is bad

When attempting to update a directory block during intent log replay, the directory block was found to contain invalid data. The containing directory will be marked bad.

A full file system check will be required.

This condition causes a directory to be lost, which will orphan all files referenced by that directory. A restore from backup is recommended.

Message
vxfs fsck: bad directory add inode *ino*

vxfs fsck: bad directory add block *bno*

vxfs fsck: bad directory add entry, block *bno*

An intent log entry that attempts to add an entry to a directory has invalid data. Either the intent log has been corrupted or there is a serious software error. The intent log replay will terminate.

If there has been accidental corruption, run a full file system check. If this problem reoccurs, contact your service representative.

Message
vxfs fsck: bad directory reorg inode *ino*

vxfs fsck: bad directory reorg block *bno*

An intent log entry to compress a directory block has invalid data. Either the intent log has been corrupted or there is a serious software error. The intent log replay will terminate.

If there has been accidental corruption, run a full file system check. If this problem reoccurs, contact your service representative.

Message
vxfs fsck: bad emap op block *bno* len *length*

An intent log entry to update a free extent map has invalid data. Either the intent log has been corrupted or there is a serious software error. The intent log replay will terminate.

If there has been accidental corruption, run a full file system check. If this problem reoccurs, contact your service representative.

9.1.8.2 Messages from pass 1

Message
vxfs fsck: inode *ino* marked bad, allocation flags *flags* clear? (ynq)

The inode was marked bad by the kernel sometime during the period when the file system was last mounted and in use.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has invalid mode *mode* clear? (ynq)

The mode or type of file is invalid.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: partially allocated inode *ino*, mode *mode* clear? (ynq)

An inode has a zero mode, but other fields are not zero.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has invalid size *size* clear? (ynq)

The file size is a negative number.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has invalid extent size *size* clear? (ynq)

vxfs fsck: inode *ino* has invalid reservation *reserv* clear? (ynq)

A fixed extent size reservation was specified for the file, and is negative or larger than the size of an allocation unit, or there was a reservation for the file that was negative or greater than the number of blocks in the file system.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has invalid allocation flags *flags* clear? (ynq)

The allocation flags *i_aflags* field has an invalid value.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has invalid extended operation *op* clear? (ynq)

The extended operation flags field *i_eopflags* has an invalid value.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* not regular file and has extended operation *op* clear? (ynq)

An extended operation that can only happen for regular files is present on a file that is not a regular file.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has invalid extended operation data *offset* clear? (ynq)

The inode extended operations flag field indicates that an extent is to be cleared, but the offset in the file specified by *i_eopdata* is invalid.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has invalid link count *count* clear? (ynq)

The link count is less than zero, or greater than the number of files in the file system.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: root inode is not a directory clear? (ynq)

The *root* inode is not a directory. The program will terminate if there is an attempt to clear the *root* inode. The file system must be restored from backups.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has invalid parent inode *parent* clear? (ynq)

The *i_dotdot* field of the inode contains the ".." or parent directory for each directory inode. This field did not specify a valid inode number.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has bad orgtype *orgtype* clear? (ynq)

The VxFS file system supports several inode organizations. These organization types (orgtypes) are internally represented as numbers. The current choices are *EXT4* (orgtype 1), or *IMMED* (orgtype 2). This message indicates that an invalid organization type field was found.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* is special with orgtype *orgtype* clear? (ynq)

A special file cannot be immediate or have data extents.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has orgtype IMMED with mode *mode* clear? (ynq)

A regular file cannot be immediate.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has orgtype IMMED with size *size* clear? (ynq)

An immediate file has its data entirely in the inode and thus must have an inode size in the range 0 to 96.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* has orgtype IMMED with blocks *blocks* clear? (ynq)

An immediate file cannot have data blocks, and thus *i_blocks* must be 0.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* special with size *size* clear? (ynq)

A special file cannot have a nonzero file size.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* special with blocks *blocks* clear? (ynq)

A special file cannot contain data blocks.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* special with extent size *size* clear? (ynq)

A special file cannot have a fixed extent size.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* is an immediate directory with size *size* clear? (ynq)

Immediate directories must have a file size of 96.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* is a directory with extent size *size* clear? (ynq)

Directories may not have fixed extent sizes.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* is a directory with size *size* clear? (ynq)

Directories larger than 96 bytes must have a file size that is a multiple of the file system block size.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* is a symlink with extent size *size*

Symlinks cannot have a fixed extent size.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* is a symlink with reservation *size*

Symlinks cannot have a space reservation.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* sparse - mode is *mode in octal* clear? (ynq)

A symlink or directory is sparse.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* bad extent, start *bno* length *len* clear? (ynq)

A data extent was found that is not in the range of valid data extents for the file system.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* sparse directory, start *bno* length *len* clear? (ynq)

Directories may not be sparse. A sparse file is a file that has an area in the middle of the file that is not allocated.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* extent after EOF, start *bno* length *len* clear? (ynq)

vxfs fsck: inode *ino* has indirect address extent after EOF clear? (ynq)

An extent was found after a direct extent with a size of zero was discovered. When a file is sparse, there may be direct extents that do not have a *bno* in the start field; a size must be marked in the inode for these extents. This information is necessary for these extents in order to determine the offset in the file of all direct extents.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* invalid indirect address extent clear? (ynq)

The file had either an indirect extent size and no indirect address extens, or an indirect address extent and no indirect extent size.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* extended operation completion failed clear? (ynq)

The inode had a pending extended inode operation to perform. When this operation was performed, it could not be completed.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* dup/bad blocks found clear? (ynq)

The file had either invalid data extents, or data extents which were duplicates of data extents found in another file.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* found *actual* blocks expected *expected* clear? (ynq)

The number of blocks actually allocated to the file does not match the *i_blocks* field. This is common in the *ufs* file system but in VxFS it should not occur, thus the file is considered to be bad.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* found size *actual* expected *expected* clear? (ynq)

The file size or reservation and the location of the last data extent in the file do not correspond.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* bad extent start *bno* len *len* clear? (ynq)

An invalid extent start was found in the inode.

y: Clear the inode. The inode will be cleared and marked as available.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: continue to search? (ynq)

Once a bad extent has been found, should the remaining extents in the file be verified?

y: Check the remaining extents of this inode.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* dup block - extent start *bno* offset *off* continue? (ynq)

A block is used in more than one file. Since it is not possible to determine which, if any, of the referencing files the block actually belongs to, all files that have duplicate blocks are considered to be bad.

y: Check the remaining extents of this inode.

Message
vxfs fsck: pass1b - checking for more dup blocks

Once there have been duplicate blocks found, the inode list is rescanned to find the original reference to the duplicate blocks.

None.

Message
vxfs fsck: inode sanity checks have not been completed - restart? (ynq)

If a failure is detected during pass1 or pass1b which causes an inode to be removed after it has updated internal reference counts, another pass needs to be made to recalculate reference counts.

y: Restart

n: Go on to pass 2

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: dup block list overflow on inode *ino*

When duplicate blocks are found, the list of duplicates are saved so that the original reference to the blocks may be found. If the list becomes full this message is produced.

Use *fsdb*(1M) to clear the offending inodes or restore the file system from backups.

9.1.8.3 Messages from pass 2

Message
vxfs fsck: directory ino block *bno* bad hash count *count* clear block? (ynq)

At the start of each directory block is the number of hash chains. An invalid number of hash chains is present in this block.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory ino block *bno* bad free count *count* clear block? (ynq)

As part of the header area of the directory block, a count of the free space in the block is maintained. The count in this block is invalid.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* has reclen *len* clear block? (ynq)

A directory entry has a record length that is smaller than the minimum size of 12, or that is so large that it cannot fit in the directory block.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* bad namlen *namlen* reclen *reclen* ino *ino* clear block? (ynq)

A directory entry has a name length that is negative or too large.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* references self ino *ino* clear block? (ynq)

The inode number in a directory entry is the inode number of the directory containing the directory block, thus making the entry self-referential. In the VxFS file system, the "." and ".." entries are in the inode itself, not in a directory block.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* references parent ino *ino* clear block? (ynq)

The inode number in a directory entry is the inode number of the parent directory. In the VxFS file system, the "." and ".." entries are in the inode itself, not in a directory block.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* has ino *ino* clear block? (ynq)

The inode number in a directory entry is not a valid inode number for the file system.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* partial entry ino *ino* namlen *namlen* clear block? (ynq)

A directory entry must have either a zero inode reference and zero name length, or a nonzero inode reference and name length. In other words, a directory entry must consist of both an inode number and a name.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* has name '.' clear block? (ynq)

In the VxFS file system, the "." and ".." entries are in the inode itself, not in a directory block.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* has name '..' clear block?(ynq)

In the VxFS file system, the "." and ".." entries are in the inode itself, not in a directory block.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* name contains a slash

clear block? (ynq)

Since the slash character is the component separator in a pathname, a directory entry name may not contain a slash.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* name contains a NULL

clear block? (ynq)

Null characters are not permitted in directory entry names.

y: Clear the block. The inode will be initialized as an empty directory block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* references free inode ino *ino* remove entry? (ynq)

The directory entry references a free inode. The entry must be removed.

y: Clear the entry.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* parent mismatch ino *ino* parent *parent* remove entry? (ynq)

The directory entry references an inode which is itself a directory. The parent specification *i_dotdot* in the referenced directory is not to the inode containing this directory block. The entry must be removed.

y: Clear the entry.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* rebuild header? (ynq)

When a directory entry is removed in one of the preceding cases, the free count and hash chains in the directory block need to be updated to reflect the removal.

y: Rebuild the header.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* free count mismatch

expected *wanted* found *actual* fix? (ynq)

The free count in the directory header did not match the amount of space used by the directory entries in the block.

y: Recalculate the free count.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory *ino* block *bno* offset *off* hash chain mismatch

fix? (ynq)

The hash chain linkage does not match the directory entries.

y: Remake hash chains.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: inode *ino* contains invalid directory blocks clear? (ynq)

A directory contains invalid blocks. The directory is considered to be bad.

y: Clear the inode.

It is advisable **always** to answer "yes" to this message.

9.1.8.4 Messages from pass 3

Message
vxfs fsck: inode *ino* link count is *actual* should be *desired* adjust? (ynq)

The link count in the inode does not match the references to the inode. This will occur when files are removed during passes 1 and 2.

y: Adjust the link count.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: unreferenced file, ino *ino* reconnect? (ynq)

The file is not referenced in any directory block. The likely cause is that a directory block or directory inode was removed during passes 1 and 2.

y: Connect the inode to *lost+found*.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: sorry cannot reconnect clear? (ynq)

The *lost+found* directory is missing or full, or an I/O error occurred while trying to reconnect.

y: Clear the inode.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: directory loop starting at *ino* ending at *ino*, fix? (ynq)

There is a loop when following the ".." entries from the inode named as the starting inode. The ".." entry of the ending inode in the message is the starting inode.

y: Connect the inode to *lost+found*.

It is advisable **always** to answer "yes" to this message.

9.1.8.5 Messages from pass 4

Message
vxfs fsck: au *aun* imap incorrect - fix (ynq)?

The free inode map for the given allocation unit does not match the actual free inodes in the allocation unit.

y: Update the map

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: au *aun* emap incorrect - fix? (ynq)

The free extent map for the given allocation unit does not match the actual free extents in the allocation unit.

y: Update the map

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: au *aun* secondary emap incorrect - fix? (ynq)

The secondary portion of the free extent map for the given allocation unit does not match the actual free extents in the allocation unit.

y: Update the map

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: au *aun* summary incorrect - fix? (ynq)

The allocation unit summary block does not match the amount of available resources in the allocation unit.

y: Update the summary block.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: OK to clear log? (ynq)

When performing a full file system check, the intent log is zeroed.

y: Clear the log

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: free inode count incorrect found *actual* expected *used* fix? (ynq)

The *fs_ifree* field of the superblock does not match the number of free inodes in the file system

y: Update the count

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: free block count incorrect found *actual* expected *used* fix? (ynq)

The *fs_free* field of the superblock does not match the number of free blocks in the file system

y: Update the count

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: free extent vector incorrect fix? (ynq)

The *fs_efree* vector in the superblock does not match the number of free extents by size in the file system

y: Update the counts.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: set state to CLEAN? (ynq)

The file system state is set to CLEAN to indicate that the file system may be safely mounted.

y: Set state to CLEAN.

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: some queries were answered negatively - can′t set state to CLEAN

During the passes, "no" was answered to some question.

The full *fsck* sequence will be restarted.

Message
vxfs fsck: cannot update superblock

The superblock could not be written.

Check console log for possible I/O error.

Message
vxfs fsck: file system checks have not been completed - restart? (ynq)

During the passes, NO was answered to some question, or an update of file system structure failed.

y: Restart from pass 1

It is advisable **always** to answer "yes" to this message.

Message
vxfs fsck: cannot malloc inode linkage space

vxfs fsck: cannot malloc inode parent space

vxfs fsck: cannot malloc link count space

vxfs fsck: cannot malloc inode map space
vxfs fsck: cannot malloc extent map space

The *fsck*(1M) utility uses *malloc*(3C) to dynamically allocate work space. The attempt failed.

Check to see that there is sufficient swap space with *swap*(1M).

Message
vxfs fsck: cannot malloc temporary emap space

Temporary space for extent map work space could not be allocated.

Check to see if there is sufficient swap space with *swap*(1M).

9.1.9 Messages of fsdb(1M)

There are two types of error messages: Those that are generated when *fsdb* is invoked, and those that are generated in interactive mode. Error messages generated at invocation are written to *stderr*. Following the error message *fsdb* exits with a non-zero exit code. Error messages generated in interactive mode are sent to *stdout*. *fsdb* does not exit when these messages are generated.

Message
vxfs fsdb: cannot read superblock

fsdb could not read the superblock on the special device specified. Either the read failed with an error, or *fsdb* was unable to seek to the superblock offset.

Check the input device. If it is a valid VxFS file system, then schedule a full file system check at the next administrative period.

Message
vxfs fsdb: snapshot file system

This file system is a snapshot file system.

Note that most structure printing commands will not give useful results.

Message
vxfs fsdb: bad superblock checksum - setting block size to 1024

vxfs fsdb: bad superblock magic - setting block size to 1024

There is something wrong with the data in the superblock. Probably this isn't a VxFS filesystem.

Check the input device. If it is a valid VxFS file system, then schedule a full file system check at the next administrative period.

Message
vxfs fsdb clri: bad inode *nnn*

fsdb was invoked with the *-z* option to clear an inode. The inode number specified, *nnn* is not in the range of valid inodes for the file system.

Check the command line. Verify the inode number that you wish to clear, and try again.

Message
vxfs fsdb: cannot read block *nnn*

vxfs fsdb: cannot write block *nnn*

vxfs fsdb clri: cannot read block *nnn*

vxfs fsdb clri: cannot write block *nnn*

A read or write to the disk failed while attempting to clear an inode.

Check the device. Make sure it is the correct device and if the failure was on a write, then make sure the device isn't write protected. If there is a hard failure on the device and it contains a valid VxFS file system, then schedule a full file system check at the next administrative period.

Message
vxfs fsdb: unrecognized command

This message is generated if a command is unrecognized or if a valid command is invoked with arguments of the wrong type or number.

Verify that the command name being used is valid, that it is invoked with the correct number of arguments, that the arguments are of the correct type (strings or numbers), and that the arguments and command are in the correct order. Use the help screen and manual page for more information.

Message
vxfs fsdb: cannot seek to *nnn*

This message can be generated from several places within *fsdb*. In every case, the byte offset to seek to is negative (or very large), which is invalid. The most common way to generate this error is to specify a negative block number (through relative addressing) to the b command. Another possibility is that a block number being read from an inode, or some other file system structure, is invalid.

Verify the offset being used for the failing command.

Message
vxfs fsdb: max read is *mmm* requested *nnn*

A user request requires more than the maximum I/O size to be read from the disk. The maximum I/O size (generally 65536) was determined when *fsdb* was compiled. A large read request can be generated by trying to print a large number of items. For example *p 0x100000 x* will try to print one million hex integers, which is implemented as a 4 Mbyte read.

Try a smaller print command.

Message
vxfs fsdb: ?

The *?* error message is printed when a command is interrupted by SIGINT. Any currently executing command (such as printing a large number of items) is interrupted, and *?* is printed on a line by itself.

If you didn't want to stop the command then run it again.

Message
vxfs fsdb: lseek failed at *nnn*

This message might be generated if the special device specified was not a regular file or special file, or if the file is been accessed over RFS, and the server system fails. These are the only two cases for this message to be generated.

Make sure the *fsdb* command was invoked with the correct special file.

Message
vxfs fsdb: read failed at *bbb* requested *nnn* got *rrr*

A disk read failed at byte offset *bbb*, returning only part of the data that was requested. This can occur if there is a disk error such that only the first part of the read is satisfied. More commonly, this will occur if a read crosses over the last block of the file system.

Check the offset and the special device. If the offset is invalid, then set a valid offset and try the command again. If there was a device failure and the device contains a VxFS file system, then schedule a full file system check at the next administrative period.

Message
vxfs fsdb: read failed at *nnn* error = *e*

A disk read failed with *errno e*. This can occur if there is a disk error or if a read past the end of the disk is attempted.

Check the offset and the special device. If the offset is invalid, then set a valid offset and try the command again. If there was a device failure and the device contains a VxFS file system, then schedule a structural check at the next administrative period.

Message
vxfs fsdb: offset or length is not a block multiple

This message occurs when attempting to write a chunk of data to disk that is not block-aligned. It should never occur during normal operation, and indicates a problem with *fsdb*.

This should be reported as a bug to your customer support organization.

Message
vxfs fsdb: maximum write is *nnn*

This message occurs when attempting to write a chunk of data to disk that is larger than a compiled in constant. Since all data is read from disk before being written, this should never occur during normal operation, and indicates a problem with *fsdb*.

This should be reported as a bug to your customer support organization.

Message
vxfs fsdb: write failed at *bbb* requested *nnn* wrote *rrr*

A disk write failed at byte offset *bbb*, writing only part of the data that was requested. This can occur if there is a disk error such that only the first part of the write is done. More commonly, this will occur if a write crosses over the last block of the file system.

Check the offset and the special device. If the offset is invalid, then set a valid offset and try the command again. If there was a device failure and the device contains a VxFS file system, then schedule a full file system check at the next administrative period.

Message
vxfs fsdb: write failed at *bbb* error = *e*

A disk write failed with *errno e*. This can occur if there is a disk error such that none of the write can be done. More commonly, this will occur if a write beyond the end of the disk is attempted.

Check the offset and the special device. If the offset is invalid, then set a valid offset and try the command again. If there was a device failure and the device contains a VxFS file system, then schedule a full file system check at the next administrative period.

Message
vxfs fsdb: only double words require two values

This error occurs when two values are given for assigning to a non-double-word or finding a non-double-word pattern.

Check the command entered. Either change the assignment to be a double-word or enter one value.

Message
vxfs fsdb: double words require two values

This error occurs when only one value is given for assigning to a double-word or finding a double-word pattern.

Check the command entered. Either change the assignment to be a non-double or enter two values for the halves of the double.

Message
vxfs fsdb: no current offset, cannot make relative offset change

This error message occurs when there is no current position, and an attempt is made to set a position relative to the current position. This can only occur when *fsdb* is first invoked, before any current position has been set.

Set an absolute current position.

Message
vxfs fsdb: cannot change offset to: *nnn*

This message only occurs when attempting to set the current offset to a negative value by specifying a negative number to a positioning command which would cause the current position to be set to an offset less than 0.

Check the command entered and the current offset.

Message
vxfs fsdb: bad alignment

This message is generated when attempting to print a half-word, word, or double-word when not aligned on a half-word, word, or word boundary (double-words require alignment only to a word boundary).

Check the command entered and the current offset.

Message
vxfs fsdb: no current inode

vxfs fsdb: no current allocation unit

vxfs fsdb: no current directory entry

This message is generated when there is no current inode set − "inode", in this instance, meaning an allocation unit and directory entry−because the user has never defined an absolute inode, and an attempt is made to seek to an inode via relative addressing, or to return to the current inode.

In addition, the inode messages – "inode" now specifically referring to a directory entry – can be displayed when there is an attempt to alter an inode field, or to seek to an address based on the current inode, and there is no current inode or directory entry.

Set a current inode (allocation unit, directory entry) and retry the command.

Message
vxfs fsdb: no current directory block

An attempt was made to set the current position to the current directory block (via the *cdb* command) when there is no current directory block set, or an attempt was made to set the current directory entry when there is no current directory block set (via the d command).

Set a current directory block and try again.

Message
vxfs fsdb: invalid inode number *nnn* acceptable range *min* to *max*

vxfs fsdb: invalid allocation unit number *n* acceptable range *min to max*

An attempt has been made to seek to an invalid inode number (or au) either through absolute or relative addressing. The message indicates the range of valid inodes (allocation units), and the number that was attempted.

Check the command and the current offset.

Message
vxfs fsdb: *m* not valid directory entry number

vxfs fsdb: only *n* directory entries

These messages can be generated when there is an attempt to seek to an invalid directory entry number. The first is displayed when a negative directory entry is requested. The second is displayed when a directory entry is requested that exceeds the number of entries in the current directory block.

Check the command and the current directory entry.

Message
vxfs fsdb: internal error

This message is only generated when an internal program error has occurred while changing a directory entry or directory block header.

This should be reported as a bug to your customer support organization.

Message
vxfs fsdb: out of range

This message is displayed if there is an attempt to change a direct extent, or direct extent size and the extent number is larger (or smaller) than the valid number of direct extents (*de* and *des* commands). It can also be generated by an attempt to set the current position (and current block) to an extent larger or smaller than the number of direct extents (*a* command).

Check the command.

Message
vxfs fsdb: *n* does not contain a valid extent

This message is generated if the a command is used to attempt to set the current position and current block to an invalid extent. An invalid extent is one that has a non-positive starting number or a non-positive size. For the purposes of the *a* command, block 0 is not considered a valid starting address.

Check the command.

Message
vxfs fsdb: bad superblock - cannot determine log position

The magic number in the superblock is incorrect, so the superblock cannot be used to determine the start and end of the log. Therefore, the *fmtlog* command cannot print the log correctly, and so generates this message.

None.

Message
vxfs fsdb: log area does not contain valid log entries

vxfs fsdb: log insanity detected at *offset*

An attempt to print the log failed because the log does contain valid entries. This may be because the log is cleared; a full *fsck*(1M) and *mkfs*(1M) will clear the log. It may also be because a disk error has corrupted the log.

If there was a device failure and the device contains a VxFS file system, then schedule a full file system check at the next administrative period. Otherwise, examine the superblock to determine the boundaries of the log, and print individual log entries manually.

Message
vxfs fsdb: bad inode extop operation□
inode number: *bb* old map val: *xx*

vxfs fsdb: bad inode operation□
inode number: *bb* old map val: *xx*

vxfs fsdb: bad maptype in inode map operation

These three errors indicate that a *VX_IETRAN* subfunction of a transaction contained invalid data in one of its fields. This indicates a corruption of the log (assuming that a valid log entry is being examined).

None.

Message
vxfs fsdb: Unknown Subfunction

This message is displayed when a log entry contains an unknown subfunction number. This may indicate a corruption of the log (assuming that a valid log entry is being examined). The corruption may be the result of a disk I/O error.

If there was a device failure then schedule a full file system check at the next administrative period.

9.1.10 Messages of fstyp(1M)

There are no messages which are specific for *fstyp*(1M). Messages which could appear when using *fstyp*(1M) you can find in Section "Miscellaneous messages".

9.1.11 Messages of labelit(1M)

Message
vxfs labelit: '-n' option for tape only

vxfs labelit: '*special*' is not a valid tape name

vxfs labelit: Valid tape names begin with '/dev/rmt' or '/dev/rtp'

These messages will be displayed if the special file from the command line is not a tape device and the *-n* flag was set. The program will exit with a return code of 33.

Check the special file to make sure it is a tape device. If it is not a tape device, then do not use the *-n* option.

Message
vxfs labelit: media not labeled!

A tape device was found to not contain a valid tape label. Unless the *-n* flag is specified, the tape must contain a valid label. The program will exit with a return code of 33.

Use the *-n* option on the command line.

Message
vxfs labelit: cannot read superblock

A read of the superblock failed. The device probably does not contain a valid VxFS file system. The program will exit with a return code of 33.

Check the special file. If the special file is a disk containing a valid VxFS file system, then schedule a full file system check at the next administrative period.

Message
vxfs labelit: cannot write label

If the label could not be written, this message will be displayed and the program will exit with a return code of 33.

Check the special file. If it is a tape, make sure it isn't write protected. If the special file is a disk containing a valid VxFS file system then schedule a full file system check at the next administrative period.

Message
vxfs labelit: cannot write superblock

A write of the superblock failed. This message will be displayed and the program will exit with a return code of 33.

Check the special file. If it is a tape, make sure it isn't write protected. If the special file is a disk containing a valid VxFS file system then schedule a full file system check at the next administrative period.

9.1.12 Messages of mkfs(1M)

Message
vxfs mkfs: cannot read superblock

An attempt to read the superblock from the *special* device failed.

Check the command line. If the device specification is valid, check the console log for possible I/O errors.

Message
vxfs mkfs: not a valid file system

When the *-m* option is specified with *mkfs*, an attempt is made to determine the file system type.

Check the command line. Try *fstyp*(1M) to determine file system type.

Message
vxfs mkfs: *special* is not writable

Device cannot be opened for writing.

Check the command line and permissions for the device.

Message
vxfs mkfs: *special*: cannot stat

stat(2) of *special* failed.

Check the command line and permissions.

Message
vxfs mkfs: *special* is mounted, can't mkfs

vxfs mkfs: *special* is mounted as *block_special*, can't mkfs

The file system is already mounted. In the second case, *special* has been converted to the corresponding block special file for the purposes of searching */etc/mnttab*.

Check the command line and mounted file systems. If the file system isn't actually mounted, but the *mount*(1M) insists that it is then */etc/mnttab* is corrupted. *umount*(1M) the file system to clear the */etc/mnttab* entry.

Message
vxfs mkfs: available space is *size* blocks

vxfs mkfs: allocation unit size is *size* blocks

The requested allocation unit size is greater than the available space for allocation units given the requested file system size.

Check the command line.

Message
vxfs mkfs: overhead per allocation unit is *size* blocks

vxfs mkfs: allocation unit size is *size* blocks

The requested allocation unit size is not sufficient to hold the resource maps and inode list.

Check the command line.

Message
vxfs mkfs: INTERNAL ERROR: internal parameter inconsistency (*imapsize*)

vxfs mkfs: INTERNAL ERROR: internal parameter inconsistency (*emapsize*)

vxfs mkfs: INTERNAL ERROR: internal parameter inconsistency (*ilbsize*)

vxfs mkfs: cannot create file system

The file system cannot be created. A previous message indicates the reason for failure.

Check the command line.

Message
vxfs mkfs: seek error at block *bno*

A seek operation failed at the specified block.

Check the command line. If the device specification is valid, check the console log for possible I/O errors.

Message

vxfs mkfs: write failure at block *bno*

A write operation failed at the specified block.

Check the command line. If the device specification is valid, check the console log for possible I/O errors.

Message

vxfs mkfs: cannot seek to superblock

vxfs mkfs: cannot read to superblock

An attempt to access the superblock of the file system failed.

Check the command line. If the device specification is valid, check the console log for possible I/O errors.

Message

vxfs mkfs: cannot write superblock

An attempt to write the superblock of the file system failed.

Check the command line. If the device specification is valid, check the console log for possible I/O errors.

Message

vxfs mkfs: *device*: sector size (*size*) is too big for FSType (8192)

vxfs mkfs: *device*: sector size (*size*) is too big for FSType I/O (*VX_MAXIO*)

vxfs mkfs: *device*: sector size (*size*) is larger than PAGESIZE (*PAGESIZE*)

vxfs mkfs: *device*: sector size (*size*) is too small for FSType (*ATOMIC_IOSZ*)

vxfs mkfs: bad sector size *size* from *device*

The requested size is not supported on this system.

Check the command line. Use numeric size instead of "-" (dash).

Message

vxfs mkfs: Cannot get device info for *device*

An attempt to get the device info failed.

Check the command line. If the device specification is valid, check the console log for possible I/O errors.

Message

vxfs mkfs: bad partition size *size* from device

The requested size is not supported on this system.

Check the command line. If the device specification is valid, check the console log for possible I/O errors. Use numeric size instead of "-" (dash).

9.1.13 Messages of mount(1M)

Message
vxfs mount: warning: bad mincache type: *value*

vxfs mount: warning: bad convosync type: *value*

vxfs mount: warning: bad badinode type: *value*

An invalid type was specified with either the *mincache* or the *convosync* mount options. The valid options are *direct*, *dsync* and *closesync*.

Check the command line.

Message
vxfs mount: cannot lock mnttab

This message results when the file */etc/mnttab* cannot be locked. The error resulting from the *lockf* call is printed and the program exits with a return code of 32. This error might occur because of the lock table becoming full.

Check the */etc/mnttab* file. Determine why the *lockf* failed and correct the problem.

Message
vxfs mount: not super user

If the *mount* system call determines that the caller is not the superuser, then an *EPERM errno* return from the system call causes this message to be generated. The program returns an exit return of 33.

Notify the system administrator, or become the superuser and try again.

Message
vxfs mount: *special* write-protected

An attempt was made to mount a file system for read/write access on a device that can't be opened for write access.

Make sure you can write to the special device.

Message
vxfs mount: *special* is corrupted. needs checking

If the state of the file system is not *VX_CLEAN*, this message is generated and the program returns with an exit code of 33.

Run *fsck* and try again. If the failure happens again run a full *fsck*.

Message
vxfs mount: cannot mount *name*

If the *mount* system call returned with an *errno* that does not indicate any of the preceding error conditions, this error message displays, and the program exits with a return code of 33.

Try to determine what component failed based on the *errno*.

Message
vxfs mount: warning: *fstr* mounted as *mount_point*

If the file system mounted has a mountpoint name that disagrees with the file system-dependent string as returned by *statvfs*(2), then this warning message is displayed.

The *labelit*(1M) utility should be used to label the file system.

Message
vxfs mount: *device* disk image is incompatible with this system

The block size or size of a structural component is not supported by this system.

It is likely that a file system is meant for another system, or another type of device on a system that supports multiple device types.

9.1.14 Messages of ncheck(1M)

Message
vxfs ncheck: Too many inode numbers as arguments to -i.

The *-i ilist* option can accept a maximum of 32 members of its *ilist* option argument.

Run the command with fewer inode numbers specified on the command line.

Message
vxfs ncheck: No devices specified.

The generic command is the only *ncheck* that allows invocation without a specific special file or collection of special files. When invoked through the generic command, the VxFS-specific command should always receive appropriate special file arguments. This diagnostic message indicates that the specific command was directly invoked without proper special file arguments.

Check the command line.

Message
vxfs ncheck: cannot malloc inode space for *special*.

Dynamic memory allocation failed.

Check the system memory usage. If the system is low on memory, try reducing memory usage or adding swap space. If the system isn't low on memory, check the process size against the maximum process size and increase the maximum if necessary.

Message
vxfs ncheck: Too many special files (increase ilist array).

When any of the constraining options is selected, (*-i*, *-s* or *-ob*), all inodes that satisfy one of the options are stored in an internal table. The subject file system has more inodes satisfying these selection criteria than fits in the table.

Try breaking the command up into multiple invocations of *ncheck* so that each invocation does a subset of the selection desired.

Message
vxfs ncheck: Directory read buffer allocation failed.

Dynamic memory allocation failed.

Check the system memory usage. If the system is low on memory, try reducing memory usage or adding swap space. If the system isn't low on memory, check the process size against the maximum process size and increase the maximum if necessary.

Message
vxfs ncheck: no more room for names.

The internal limit on the number of characters in all pathnames satisfying selection criteria has be exceeded.

Try breaking the command up into multiple invocations of *ncheck* so that each invocation does a subset of the selection desired.

Message
vxfs ncheck: name buffer allocation failed.

Dynamic memory allocation failed.

Check the system memory usage. If the system is low on memory, try reducing memory usage or adding swap space. If the system isn't low on memory, check the process size against the maximum process size and increase the maximum if necessary.

Message
vxfs ncheck: no more room for directory buffers.

The internal limit on the number of directories that can be tracked for pathname generation has been exceeded.

Try breaking the command up into multiple invocations of *ncheck* so that each invocation does a subset of the selection desired.

Message
vxfs ncheck: directory buffer allocation failed.

Dynamic memory allocation failed.

Check the system memory usage. If the system is low on memory, try reducing memory usage or adding swap space. If the system isn't low on memory, check the process size against the maximum process size and increase the maximum if necessary.

Message
vxfs ncheck: read error *block_number*.

ncheck read(2) of *block number* failed. *ncheck* uses *perror*(3C) to provide further aid in diagnosing the reason for this failure.

Try again. If the failure persists, then schedule a full file system check at the next administrative period.

Message
vxfs ncheck: ibuf allocation failure.

vxfs ncheck: ibuf reallocation failure.

Dynamic memory allocation failed.

Check the system memory usage. If the system is low on memory, try reducing memory usage or adding swap space. If the system isn't low on memory, check the process size against the maximum process size and increase the maximum if necessary.

Message
vxfs ncheck: inode list array full.

When any of the constraining options is selected, (*-i*, *-s* or *-ob*), all inodes that satisfy one of the options are stored in an internal table. The subject file system has more inodes satisfying these selection criteria than fit in the table.

Try breaking the command up into multiple invocations of *ncheck* so that each invocation does a subset of the selection desired.

9.1.15 Messages of volcopy(1M)

Message
vxfs volcopy: From device not character-special or block-special

vxfs volcopy: To device not character-special

If the output device is not a character special or if the input device is not a character special or block special, one of the preceding messages is displayed and the program exits with a return code of 41. When the input device is a disk device, it is allowed to be a block special for the purpose of easier lookups in */etc/mnttab* when dealing with snapshot file systems.

Check the specified device.

Message
vxfs volcopy: Use dd(1) command to copy tapes

This message appears if both the input and output devices are deemed to be tape devices. Literal copies of tapes using *volcopy* are not supported. The program exits with a return code of 41.

If both devices are tapes use the *dd* command.

Message
vxfs volcopy: The -buf option requires ipc

If a simple *shmat*(2) call fails, *volcopy* is unable to use shared memory for the double buffering scheme and cannot continue. The program exits with a return code of 32.

Run the command again without the *-buf* option.

Message
vxfs volcopy: read error on input

The initial read of the input devices superblock failed. The device may not contain a file system, or cannot support reads of the expected size for a VxFS file system. It is most likely that the wrong device was selected. The program exits with a return code of 41.

Check the input device. If it is a valid VxFS file system, schedule a full file system check at the next administrative period.

Message
vxfs volcopy: Read error *N* on output

Where *N* is the offending errno. If the size of read performed on the input device to read the superblock cannot be repeated on the output device, this error message is displayed. This could occur if no file system exists on the device and an end of media error was returned. If the user is not writing to a tape device or if the user did not use the *-a* option flag, the program prompts the user with this Message and ask if it is allowable to override the error and continue. If the user elects to quit, the program returns a return code of 40.

If the output is an unlabeled tape device, override the error and continue; otherwise quit and check the output device access.

Message
vxfs volcopy: IF REEL 1 HAS NOT BEEN RESTORED STOP NOW AND START OVER

This message occurs when an input tape with a reel sequence number of other than one is loaded as the first tape to restore from and the *-reel* option was used to set the first reel number to be read from. The user is prompted to continue. If the user elects to quit, the program returns a return code of 40.

If reel 1 has been restored, then continue. If reel 1 hasn't been restored, then quit and start restoring the file system from reel 1.

Message
vxfs volcopy: Tape disagrees: Reel *M* of *N* : looking for *O* of *P*

If the expected first tape reel to read from is *O* out of the sequence of *P* tapes and instead, a tape with the sequence number of *M* from a sequence of *N* tapes is discovered, this error is displayed. The user can override the error and continue.

If the tape is an input device, then it is probably the wrong tape to read from. Find the right tape and mount it in place of the current tape. If the right tape can't be located, quit; otherwise continue and override the error.

Message
vxfs volcopy: arg. (*fsname*) doesn
't agree with from fs. (*fsname*)

If the file system (*fsname*) argument does not agree with the name of the file system found in the superblock on the output device, this message appears. If the output device is not a tape, or if the user had not used the *-a* flag on the command line, the user is asked to override or quit from the program. If the user elects to quit, an exit code of 40 is returned.

Check the command line. Make sure the right input and output devices have been specified. If the output device is a tape, make sure it is the right tape. If the devices are correct, override the error and continue; otherwise quit the program.

Message
vxfs volcopy: arg. (*volname*) doesn
't agree with from vol.(*volname*)

The volume name (*volname*) argument does not agree with the volume name. In addition, if a "-" was entered as the command line volume name, this check is skipped.

Check the command line. Make sure the right input and output devices have been specified. If the input device is a tape, make sure it is the right tape. If the devices are correct, override the error and continue; otherwise quit the program.

Message
vxfs volcopy: *destdevice* less than 48 hours older than *srcdevice*

To filesystem dated:

This message, followed by a date and time, is displayed if the age of the file system on the *destdevice* output device is not at least 48 older than the file system from the *srcdevice* input device. If the *-a* option has not been specified, the user is prompted to override the condition or quit from the program. If the user quits, an exit code of 40 is returned.

Check the command line. Make sure the right input and output devices have been specified. If either device is a tape, make sure it is the right tape. If the devices are correct, override the error and continue; otherwise quit the program.

Message
vxfs volcopy: arg. (*out_volname*) doesn't agree with to vol.(*sup_pack*)

If the *out_volname* volume name supplied as a command argument does not agree with the *sup_pack* pack-name found on the output file systems superblock, this message is displayed. The user is prompted to override or quit from the program. Quitting the program returns an exit code of 40. This prompt cannot be avoided when this condition occurs.

Check the command line. Make sure the right input and output devices have been specified. If the output device is a tape, make sure it is the right tape. If the devices are correct, override the error and continue; otherwise quit the program.

Message
vxfs volcopy: from fs larger than to fs

If the size of the output devices existing filesystem - according to its superblock - suggests that the output device may not be large enough to hold the larger file system from the input device, this message is displayed. The user is prompted to override. This prompt cannot be avoided when this condition occurs.

Check the command line. Make sure the right input and output devices have been specified. If either device is a tape, make sure it is the right tape. If the devices are correct, override the error and continue; otherwise quit the program.

Message

vxfs volcopy: warning! from fs(*input_device*) differs from to fs(*output_device*)

This message is generated if the file system names in the input and output superblocks do not agree and the output device is not a tape device. The user can override the condition, provided the *-a* flag was not used on the command line. Quitting the program returns an exit code of 40.

Check the command line. Make sure the right input and output devices have been specified. If either device is a tape, make sure it is the right tape. If the devices are correct, override the error and continue; otherwise quit the program.

Message
vxfs volcopy: Error *errno* during initialization

If either the input device or output device cannot be stated, the program exits with a return code of 32.

Check the access permissions and pathnames of both devices.

Message
vxfs volcopy: Error while reading label

If this message appears, the *volcopy* label couldn't be read from the input tape device. The program exits with a return code of 40 after listing the number of reels read so far.

Check the tape drive and make sure it is online. Make sure the tape can be read.

Message
vxfs volcopy: Error while writing label

If this message appears, the *volcopy* label couldn't be written to the output tape device. The program exits with a return code of 40 after listing the number of reels written so far.

Check the tape drive and make sure it is online and not write-protected. Make sure the tape isn't write protected or has a writering.

Message
vxfs volcopy: Cannot use -reel with -e when copying to tape

If the *-e* flag was used, the actual length of each reel cannot be predicted and the *-reel* argument cannot be resolved into an input block number. The program exits with a return code of 32.

Restore the file system from the first tape without using the *-reel* option.

Message
vxfs volcopy: Cannot lseek()

This message occurs if a *-reel* argument was given and the non-tape device being read from or written to failed to seek to the offset corresponding to the requested reel. An incorrect or invalid reel number may have been given to the program. The program exits with a return code of 32.

Check the reel number supplied and the device; make sure the reel number is valid for the device.

Message
vxfs volcopy: *special* is not a vxfs filesystem

This message occurs if a *-reel* argument was given and the non-tape device being read from or written to does not contain a valid VxFS file system superblock. The program exits with a return code of 40 after listing the number of reels read or written so far.

Check the input device. If it is a valid VxFS file system, schedule a full file system check at the next administrative period.

Message
vxfs volcopy: Unable to determine machine type

This message occurs if the program was unable to perform a *uname*(2) system call. The program exits with a return code of 32.

Contact a qualified support representative.

Message
vxfs volcopy: Error allocating semaphores: *errno*

If *ipc* is available, semaphores are used to communicate between the parent and child process to synchronize the use of a shared memory buffer. The *semget* call used to set up the semaphores failed and the *errno* is displayed. The program exits with a return code of 32.

See *semget*(2) for a description of possible failures; or contact a qualified support representative.

Message
vxfs volcopy: Error setting semaphores: *errno*

This diagnostic is similar to the previous message but concerns the *semctl*(2) call.

See *semctl*(2) for a description of possible failures; or contact a qualified support representative.

Message
vxfs volcopy: Error allocating shared memory: *errno*

The *shmget*(2) call failed and the *errno* is displayed. The program exits with a return code of 32.

See *shmget*(2) for a description of possible failures; or contact a qualified support representative.

Message
vxfs volcopy: Error attaching shared memory: *errno*

The *shmat*(2) call failed and the *errno* is displayed. The program exits with a return code of 32.

See *shmat*(2) for a description of possible failures; or contact a qualified support representative.

Message
vxfs volcopy: Error locking in shared memory: *errno*

The *shmctl*(2) call used to lock the shared memory into core failed and the *errno* is displayed. The program exits with a return code of 32.

See *semctl*(2) for a description of possible failures; or contact a qualified support representative.

Message
vxfs volcopy: Out of memory

This message occurs if *ipc* is not available on this system and if a buffer of sufficient size buffer couldn't be allocated using the *sbrk*(2) call. The program exits with a return code of 32.

Check the system memory usage. If the system is low on memory, try reducing the memory usage or adding swap space. If the system isn't low on memory, check the process size against the maximum process size and increase the maximum if necessary.

Message
vxfs volcopy: Semaphore operation error *errno*

The *semop*(2) call failed. The range of return codes is not the same with failures in the child process and failures in the parent process:

- failure with the child process: return code 32 ... 33
- failure with the parent process: return code 32 ... 35

The likely cause of such a failure is some external problem with the semaphore mechanism or a spurious signal being received by either the child or parent process. If the failure is in the parent process, the message begins with a capital "S". If the failure is in the child process, the message begins with a lowercase "s".

See *semop*(2) for a description of possible failures.

Message
vxfs volcopy: I/O error *errno* on write

If the child has a write failure trying to write to the output device and the *-e* flag was not used, this message is displayed with the *errno*, and the program exits with a return code of 32. If *-e* was supplied on the command line and the error is an ENOSPC, the prompt for a new reel is given. With other errnos, the message is displayed and the program exits.

If a prompt for a new reel is given, change reels and continue. If this is a real error, check the size of the device. If the size is large enough, make sure the device is still enabled for write access.

Message

vxfs volcopy: I/O error *errno* on read

If the parent has a read failure trying to read from the input device and the *-e* flag was not specified or if the error is not an ENOSPC, this message is displayed with the errno, and the program exits with a return code of 32. If *-e* was supplied on the command line and the error was an ENOSPC, a message prompts for a new reel.

If a prompt for a new reel is given, change reels and continue. If this is a real error, check the size of the device. If the size is large enough make sure the device is still enabled for read access.

Message
vxfs volcopy: input tape

vxfs volcopy: output tape

These messages are displayed along with the textual form of the error if a read of the input or output tape fails as part of the search for a tape label.

None.

Message
vxfs volcopy: Input tape is empty

This message is generated if the read of an input tape device results in a bad label header or the header has zero reels listed in the header. This could occur if *volcopy* is used to copy from a labeled tape with no file system contents. The program exits with a return code of 40.

Make sure the right tape is mounted.

Message
vxfs volcopy: Not a labeled tape

If a tape device does not contain a valid *volcopy* header magic number, this Message is generated. If the tape is an output device, the user is prompted to override or quit. If the tape is an input device, the following message is displayed. This error may occur if an incompatible version of *volcopy* was used or if the tape is not in *volcopy* format.

If the tape is an output device, make sure the right tape is mounted. If the right tape is mounted, override and continue.

Message
vxfs volcopy: Input tape not made by volcopy

If the Not a labeled tape message relates to an input tape device, this message is generated and the program exits with a return code of 40 after displaying the total of reels processed so far.

Check the tape. If it is the correct tape, it was probably written with an incompatible version of *volcopy*. Try to find the version of *volcopy* used to write the tape and read it with that version.

Message
vxfs volcopy: Header volume(*label_volname*) does not match (*command_volname*)

If the command line supplied volume name was not entered as a "-", and the name does not agree with the volume name contained in the *volcopy* header, this message is generated and the user is prompted to override or quit.

Check the command line. Make sure the right input and output devices have been specified. If the output device is a tape, make sure it is the right tape. If the devices are correct, override the error and continue, otherwise quit the program.

Message
vxfs volcopy: Size of reel must be > 0, <= 3600

vxfs volcopy: Bpi must be 800, 1600, or 6250

These two messages relate to the value restrictions on the *-feet*, and *-bpi* arguments or on the values entered interactively for these objects. The messages also appears if the command line arguments were out of bounds. The program loops and prompts for these values until valid numbers are entered.

Enter the correct value for bpi or feet.

Message

vxfs volcopy: Cannot read header

This message is generated as part of the check for a header on a tape when the tape reels have just been changed. If the tape is an output device, the user can override the error and continue, or quit. If new tapes are used as output devices, this message and the override prompting always occurs unless all reels of the tapes are labeled before starting the *volcopy*. If the tape is an input tape, the program prompts the user to load the next reel and hit return when done.

If the input device is a tape, find the right tape and mount it. If the output device is a tape, find the right tape and override the error.

Message
vxfs volcopy: Volume is *read_volume*, not *expected_volume*.

This message is generated as part of the check for a header on a tape when the tape reels have just been changed. If the tape is an output device, the user can override the error and continue, or quit. If new tapes are used as output devices, this message and the override prompting always occurs unless all reels of the tapes are labeled before starting the *volcopy*. If the tape is an input tape, the program prompts the user to load the next reel and hit return when done.

If the input device is a tape then find the right tape and mount it. If the output device is a tape, find the right tape and override the error.

Message
vxfs volcopy: cannot access /var/adm/log/filesave.log

If logging is enabled for *volcopy*, at completion of the *volcopy* command this file is maintained to contain the last 201 *volcopy* operations. If the *volcopy* program cannot access the file for writing, then this message is generated and the program exits with a return code of 32. This does not affect the results of the *volcopy* which has already completed.

Check the file */var/adm/log/filesave.log*; if this file doesn't exist, create it.

Message
vxfs volcopy: input ERR *error*

vxfs volcopy: output ERR *error*

One of these messages is displayed along with the failing error (*error*). error refers to the device file and may read for example "no such file or directory". This occurs as part of a reel changing operation when the device is to be reopened for access. It is possible that the wrong drive was selected, or the drive did not come online.

Make sure the tape drive is online and the tape is on the right drive.

Message
vxfs volcopy: Initialization error *errno*

Part of the reel changing operation to reaccess the device failed. If the *volcopy* program cannot access */var/adm/log/filesave.log* for writing, then this message is generated and the program exits with a return code of 32. This does not affect the results of the *volcopy* which has already completed.

Make sure the tape drive is online and the tape is on the right drive.

Message
vxfs volcopy: Need reel *N* label says reel *O*

The reel number contained in the label read from the input tape device for the new reel does not agree with the expected reel number. The program prompts the user to reload the tape.

Find the correct tape and load it.

Message
vxfs volcopy: Cannot re-write header - Try again!

This message occurs if the newly loaded reel cannot be written to for updating the tape label. This can often happen if the drive was left in a read-only mode, or if the tape is not enabled for writing.

Make sure the tape drive is enabled for write access. Make sure the tape is enabled for write access.

Message
vxfs volcopy: Impossible case

This message occurs if the tape being changed is neither an input nor an output device.

Exit the program and try again.

9.1.16 Messages of vxdump(1M)

There are no messages which are specific for *vxdump*(1M). You can find in Messages which could appear when using *vxdump*(1M) Section "Miscellaneous messages".

9.1.17 Messages of vxrestore(1M)

The diagnostic output of *vxrestore* is identical to that for the standard *ufsrestore*. Just those diagnostics that are specific to *vxrestore* are outlined in this section. In particular, *vxrestore* has the added option of selecting the way in which extended attributes of files on the source archive are to be handled (see the information on the *-e* option). This diagnostic section describes messages that may be generated when dealing with extent attributes.

Message
vxfs vxrestore: cannot preserve extent attributes

Attributes for files cannot be preserved. There are two major reasons that such a failure might occur:

- The file system onto which the files are to be restored is not a VxFS file system. Since other file system types do not support the extent attributes of the VxFS file system, the attributes cannot be established in the new files.
- The file system onto which a file is to be copied, moved, or restored from an archive is a VxFS file system, but the block sizes of the source and target file systems make extent attributes of the file impossible to maintain.

If the source files extent attributes must be preserved, the copy must be done onto a VxFS file system of appropriate block size. Either the target file system must be remade as VxFS or a different, compatible file system must be chosen as the target.

Message
vxfs vxrestore: skipping *filename*

This message is issued when the *-e* option argument *force* has been selected and when a files extent attributes could not be preserved in the target file system. The file is skipped and no new file is created.

The file can be restored into the current target file system without its extent attributes by selecting a different *-e* option argument. If extent attributes must be preserved, a different file system will have to be used as a target, or the current one must be defined as a VxFS file system of the same block size as that from which the archive was made.

9.2 Kernel messages

When the file system encounters problems, it can respond in any of the following ways:

- marking an inode bad
- disabling transactions
- disabling the file system

If the file system determines that the information in the inode may be invalid, the file system marks the inode bad. Inodes can be marked bad if an inode update or a directory block update fails. In the case of these types of failures, the file system doesn't know what information is on the disk, and considers invalid the information that it does find. When an inode has been marked bad, the kernel still permits access to the name. Any attempt to access the data in the file or change the inode fails.

If the file system encounters an error while writing the intent log, transactions are disabled. When transactions are disabled, the files in the file system can still be read or written. However, no block or inode allocations or frees, attribute changes, directory entry changes, or any other changes to the file system meta data are allowed.

If an error occurs that compromises the integrity of the file system, the file system disables itself. This type of error is usually caused by errors updating the superblock and inode list during error handling. If the log fails or an inode list error occurs, the superblock must be updated so the next *fsck*(1M) will do a structural check (see Section "fsck(1M)"). If this superblock update fails, any further changes to the file system may cause inconsistencies that aren't detected by the log replay. To avoid this situation, the file system disables itself. When the file system is disabled, no data can be written to the disk. Although some trivial file system operations such as *VOP_SEEK* still work, most operations simply return *EIO* if the file system is disabled. *VOP_PUTPAGE* discards pages without writing them.

The only thing that can be done when a file system is disabled is an *unmount*. The file system is usually disabled because of disk errors. Although a log replay still produces a clean file system, a structural *fsck*(1M) should be done as a safety check. The following command should be used to ensure that a full structural *fsck* is run:

fsck -F *vxfs* -o *full* -y *special*

The disk problems that caused the file system to be disabled should be tracked down and fixed as soon as possible. The file system is usually disabled because the superblock cannot be written with the *VX_FULLFSCK* flag.

Message
NOTICE: vxfs: vx_nospace - *mount_point* file system full (*n* block extent)

The file system is out of space. The free space in the file system should be monitored. If the file system is becoming full, action should be taken to prevent the file system from running out of space again. Often, there is plenty of space and one runaway process uses up all the remaining free space. In other cases, the free space available becomes fragmented and unusable for some files.

To keep the file system usable, create some free space. If a runaway process has used up all the space, then stop that process, find the files created by the process, and remove those files. If the file system is out of space, remove files, defragment, or expand the file system. Expansion is discussed in Section "Resizing the file system" and in the discussion of *fsadm* in Section "Resizing a file system".

To remove files, use the *find*(1) command to locate the files that are to be removed. To get the most space with the least amount of work, remove large files or file trees that are no longer needed. To defragment the file system, use the *fsadm*(1M) command.

Message
WARNING: vxfs: vx_snap_strategy - *mount_point* file system write attempt to read-only file system

WARNING: vxfs: vx_snap_copy - *mount_point* file system write attempt to read-only file system

This message is printed if the kernel attempts to write to a read-only file system. It is unlikely that this error will occur; however, if this situation arises, the file system will be disabled.

The file system was not written, so no action is required to correct the problem. This should be reported as a bug to your customer support organization.

Message
WARNING: vxfs: vx_mapbad - *mount_point* file system free extent bitmap in au *aun* marked bad

WARNING: vxfs: vx_mapbad - *mount_point* file system free inode bitmap in au *aun* marked bad

WARNING: vxfs: vx_mapbad - *mount_point* file system inode extended operation bitmap in au *aun* marked bad

If there is an I/O failure while writing a bitmap, the map is marked bad. The kernel no longer trusts bad maps, so no resource allocation can be done from maps that are bad. This situation can cause the file system to report "out of space" or "out of inode" error messages even though *df*(1) may report an adequate amount of free space.

This error may also occur due to bitmap inconsistencies. If a bitmap fails a consistency check, or blocks are freed that are already free in the bitmap, the file system has been corrupted. This may have occurred because someone has written directly to the device or used *fsdb*(1M) to change the file system.

This error sets the *VX_FULLFSCK* flag on the file system. If the map that failed was a free extent bitmap, and the *VX_FULLFSCK* flag can't be set, then the file system is disabled.

Check the console log for I/O errors. If the problem is a disk failure, then the disk should be fixed as soon as possible. If the problem is not related to an I/O failure, find out how the disk got corrupted. If nobody is writing to the device, then the problem should be reported to your customer support organization. In either case, the file system should be unmounted and checked as soon as possible. Make sure the *fsck*(1M) program does a full structural check at the next administrative period.

Message
WARNING: vxfs: vx_sumupd - *mount_point* file system summary update in au *aun* failed

WARNING: vxfs: vx_sumupd - *mount_point* file system summary update in inode au *aun* failed

An I/O error occurred while writing the allocation unit or inode allocation unit bitmap summary information to disk. This error sets the *VX_FULLFSCK* flag on the file system. If the *VX_FULLFSCK* flag can't be set, then the file system is disabled.

Check the console log for I/O errors. If the problem was caused by a disk failure, then the disk should be fixed as soon as possible. If the disk has failed, it should be repaired before the file system is mounted for write access. Make sure the *fsck*(1M) program does a full structural check at the next administrative period.

Message
WARNING: vxfs: vx_direrr - *mount_point* file system inode *inumber* block *blkno* error *errno*

WARNING: vxfs: vx_direrr - *mount_point* file system inode *inumber* immediate directory error *errno*

A directory operation failed in an unexpected manner. The mountpoint, inode, and block number should identify the failing directory. If the inode is an immediate directory, the directory entries are stored in the inode, so no block number is reported. If the error is *ENOENT* or *ENOTDIR*, an inconsistency was detected in the directory block. This inconsistency could be a bad free count, a corrupted hash chain, or any similar directory structure error. If the error is *EIO* or *ENXIO*, an I/O failure occurred while reading or writing the disk block.

When this error occurs, the *VX_FULLFSCK* flag is set in the superblock so *fsck*(1M) should do a full structural check at the next administrative period.

Check the console log for I/O errors. If the problem is a disk failure, the disk should be fixed as soon as possible. If the disk has failed, it should be repaired before the file system is mounted for write access. The file system should be unmounted and checked as soon as possible. Make sure the *fsck*(1M) program does a full structural check.

Message
WARNING: vxfs: vx_ialloc - *mount_point* file system inode *inumber* not free

When the kernel allocates an inode from the free inode bitmap, it checks the mode and link count of the inode. If either is non-zero, the free inode bitmap is corrupted, or the inode list has been corrupted.

When this error occurs, the *VX_FULLFSCK* flag is set in the superblock, so *fsck*(1M) should do a full structural check at the next administrative period.

The file system should be unmounted as soon as possible and a full structural *fsck*(1M) should be run.

Message
NOTICE: vxfs: vx_noinode - *mount_point* file system out of inodes

The file system is out of inodes. The free inodes in the file system should be monitored. If the file system is getting full, action should be taken to prevent the file system from running out of inodes.

To keep the file system usable, free inodes must be created; either remove files or expand the file system. Expansion is discussed in Section "Resizing the file system" and in the discussion of *fsadm* in Section "Resizing a file system".

Message
WARNING: vxfs: vx_iget - *mount_point* file system invalid inode number *inumber*

When the kernel attempts to read an inode, it checks the inode number against the valid range. If the Inode number is out of range, the data structure that referenced the inode number is incorrect and must be fixed.

When this occurs, the *VX_FULLFSCK* flag is set in the superblock, so that *fsck*(1M) should do a full structural check at the next administrative period.

The file system should be unmounted as soon as possible and a full structural *fsck*(1M) should be run.

Message
WARNING: vxfs: vx_iget - inode table overflow

This message is printed when all the in-memory inodes in the system are busy and an attempt is made to use a new inode.

Look at the processes that are running and try to determine which processes are using inodes. If it appears there are runaway processes, they might be tying up the inodes. If the system load appears normal, increasing the *VX_NINODE* parameter in the kernel should solve the problem.

Message
WARNING: vxfs: vx_ibadinactive - *mount_point* file system can't mark inode *inumber* bad

WARNING: vxfs: vx_ilisterr - *mount_point* file system can't mark inode *inumber* bad

This message appears when an attempt to mark an inode bad on disk has failed and then the superblock update to set the *VX_FULLFSCK* flags has also failed. This message indicates that a catastrophic disk error may have occurred since both an inode list block and the superblock have had I/O failures.

If this situation occurs, the file system will be disabled to preserve file system integrity.

The file system should be unmounted and a structural *fsck*(1M) performed as soon as possible. Check the console log for I/O errors. If the disk has failed, it should be repaired before the file system is mounted for write access.

Message
WARNING: vxfs: vx_ilisterr - *mount_point* file system error reading inode *inumber*

An I/O error occurred while reading the inode list. The *VX_FULLFSCK* flag is set in the file system.

Check the console log for I/O errors. If the problem was caused by a disk failure, then the disk should be fixed as soon as possible. If the disk has failed, it should be repaired before the file system is mounted for write access. The file system should be unmounted and checked as soon as possible. Make sure the *fsck*(1M) program does a structural check.

Message
WARNING: vxfs: ...

vx_bmap - *mount_point* file system inode *inumber* marked bad

vx_bmap_indirect - *mount_point* file system inode *inumber* marked bad

vx_delbuf_flush - *mount_point* file system inode *inumber* marked bad

vx_dirbread - *mount_point* file system inode *inumber* marked bad

vx_dircreate - *mount_point* file system inode *inumber* marked bad

vx_dirlook - *mount_point* file system inode *inumber* marked bad vx_do_getpage - *mount_point* file system inode *inumber* marked bad

vx_do_extrealloc - *mount_point* file system inode *inumber* marked bad

vx_exttrunc - *mount_point* file system inode *inumber* marked bad

vx_get_alloc - *mount_point* file system inode *inumber* marked bad

vx_iglock - *mount_point* file system inode *inumber* marked bad

vx_ilisterr - *mount_point* file system inode *inumber* marked bad

vx_ilock - *mount_point* file system inode *inumber* marked bad

vx_indtrunc - *mount_point* file system inode *inumber* marked bad

vx_iread - *mount_point* file system inode *inumber* marked bad

vx_iremove - *mount_point* file system inode *inumber* marked bad

vx_irwlock - *mount_point* file system inode *inumber* marked bad

vx_logwrite_flush - *mount_point* file system inode *inumber* marked bad

vx_readnomap - *mount_point* file system inode *inumber* marked bad

vx_trancommit - *mount_point* file system inode *inumber* marked bad

vx_trunc - *mount_point* file system inode *inumber* marked bad

vx_undiradd - *mount_point* file system inode *inumber* marked bad

vx_undirrem - *mount_point* file system inode *inumber* marked bad

vx_unindtrunc - *mount_point* file system inode *inumber* marked bad

vx_untatran - *mount_point* file system inode *inumber* marked bad

vx_zero_alloc - *mount_point* file system inode *inumber* marked bad

When the kernel decides it can't trust an inode it marks the inode bad on disk. The most common reason for marking an inode bad is a disk I/O failure. If there is an I/O failure in the inode list or on a directory block or an indirect address extent, the kernel will decide it can't trust the data in the inode or it can't trust the data it tried to write to the inode list. In these cases, an error message should have been printed from the disk driver followed by one or more inodes being marked bad.

Failures in validation checks can also mark an inode bad. If the kernel finds a bad extent address, invalid inode fields or corruption in directory data blocks it will mark the inode bad. A validation check failure indicates the file system has been corrupted. This probably happened because someone has written directly to the device or used *fsdb*(1M) to change the file system.

When this error occurs, the *VX_FULLFSCK* flag is set in the superblock so *fsck*(1M) should do a full structural check at the next administrative period.

Check the console log for I/O errors. If the problem is a disk failure then the disk should be fixed as soon as possible. If the disk has failed, it should be repaired before the file system is mounted for write access.

If the problem is not related to an I/O failure, find out how the disk got corrupted. If nobody is writing to the device, then the problem should be reported to your customer support organization. In either case, the file system should be unmounted and checked as soon as possible. Make sure the *fsck*(1M) program does a structural check.

Message
WARNING: vxfs: vx_idelxwri_done - *mount_point* file system inode *inum* had a write error at offset *off*

This message prints when the file system encounters an error while flushing non-synchronous extending write data. Since the data for the file couldn't be written, the file probably contains stale data. If the file system was mounted with the *blkclear* mount option, then the *VX_AF_NOGROW* flag is set on the file. When this flag is set, further attempts fail to extend the file. The flag is cleared when the file is truncated.

Check the console log for I/O errors. If the problem was caused by a disk failure, then the disk should be fixed as soon as possible. If the disk has failed, it should be repaired before the file system is mounted for write access. Make sure the *fsck*(1M) program does a structural check at the next administrative period.

The *ff*(1M) utility should be used to determine the name of the file that failed. The contents of the file should be checked, and if necessary restored from backup or recreated.

Message
WARNING: vxfs: vx_log_add - *mount_point* file system log overflow

This message appears when the log id overflows. When the log id reaches *VX_MAXLOGID* (approximately 1 billion by default) a flag is set so the file system will reset the log id at the next opportunity. If the log id has not been reset, when the log id reaches *VX_DISLOGID* (approximately *VX_MAXLOGID* plus 500 million by default), the file system is disabled to preserve its integrity. Since log reset will occur at the next 60-second sync interval, this should never happen.

The file system should be unmounted and a structural *fsck*(1M) performed as soon as possible.

Message
WARNING: vxfs: vx_logerr - *mount_point* file system log error *errno*

This message is printed when a write to the intent log fails. The kernel will attempt to set the *VX_FULLFSCK* and *VX_LOGBAD* flags in the superblock to prevent log replay from being run. If the superblock can't be updated, the file system will be disabled to preserve its integrity.

The file system should be unmounted and a structural *fsck*(1M) performed as soon as possible. Check the console log for I/O errors. If the disk has failed, it should be repaired before the file system is mounted for write access.

Message
WARNING: vxfs: vx_ag_init - *mount_point* file system validation failure

When a vxfs file system is mounted, the file system structure is read from disk. If the file system is marked clean, then the structure should be correct. Also, the first block of the intent log is cleared. If any of the I/O fails, or the structure is not consistent, then the *VX_FULLFSCK* flag will be set and the mount will fail. If the error isn't related to an I/O failure, this may have occurred because someone has written directly to the device or used fsdb to change the file system.

Check the console log for I/O errors. If the problem is a disk failure, then the disk should be fixed as soon as possible. Run a full structural fsck on the file system and try to mount it again. If the problem is not related to an I/O failure, find out how the disk got corrupted.

If nobody is writing to the device, then the problem should be reported to your customer support organization.

Message
WARNING: vxfs: vx_wsuper - *mount_point* file system superblock update failed

An I/O error occurred while writing the superblock during a resize operation. The file system is disabled.

The file system should be unmounted and checked as soon as possible. Make sure a full structural fsck is performed. Check the console log for I/O errors. If the disk has a hard failure, it should be repaired before the file system is mounted for write access.

Message
WARNING: vxfs: vx_snap_copyblk - *mount_point* primary file system read error

This error occurs when there is a snapshot file system. When the primary file system is written, copies of the original data must be written to the snapshot file system. If a read error occurs on a primary file system during this copy, any snapshot file system that doesn't already have a copy of the data is out of date and must be disabled.

An error message should have been printed for the primary file system. Resolve the error on the primary file system and rerun any backups or other applications that were using the snapshot that failed when the error occurred.

Message
WARNING: vxfs: vx_snap_bpcopy - mount_point snapshot file system write error

This message is printed when there is a snapshot file system and when a write to that snapshot file system fails. As the primary file system is updated, copies of the original data are read from the primary file system and written to the snapshot file system. If one of these writes fails, then the snapshot file system is disabled.

Check the console log for I/O errors. If the disk has failed, it should be repaired before it is used again. Resolve the error on the disk and rerun any backups or other applications that were using the snapshot that failed when the error occurred.

Message
WARNING: vxfs: vx_snap_alloc - *mount_point* snapshot file system out of space

During a snapshot backup, as the primary file system is modified, the original data is copied to the snapshot file system. This message is printed if the snapshot file system runs out of space to store the changes.

This can occur if the snapshot file system is left mounted by mistake. It can also occur if the snapshot file system was given too little disk space to work with or the primary file system had an unexpected burst of activity.

The shapshot file system is disabled.

Make sure the snapshot file system was given the correct amount of space. If it was, then try to determine the activity level on the primary file system. If the primary file system was unusually busy, then try rerunning the backup. If the primary file system is no busier than normal, then try moving the backup to a time when the primary file system is relatively idle or increase the amount of disk space allocated to the snapshot file system.

Rerun any backups that failed when the error occurred.

Message
WARNING: vxfs: vx_snap_getbp - *mount_point* snapshot file system block map write error

WARNING: vxfs: vx_snap_getbp - *mount_point* snapshot file system block map read error

During a snapshot backup, each snapshot file system maintains a block map on disk. The block map tells the snapshot file system where data from the primary file system is stored in the snapshot file system. If an I/O operation to the block map fails, then the snapshot file system is disabled.

Check the console log for I/O errors. If the disk has failed, it should be repaired before it is used again. Resolve the error on the disk and rerun any backups that failed when the error occurred.

Message
WARNING: vxfs: vx_disable - *mount_point* file system disabled

This message is printed when a file system is disabled. It is preceded by a message that specifies the reason for disabling.

This usually indicates a serious problem with the disk.

The file system should be unmounted and a full structural *fsck*(1M) performed as soon as possible. If the disk has failed, it should be repaired before the file system is mounted for write access.

Message
WARNING: vxfs: vx_disable - *mount_point* snapshot file system disabled

This message is printed when a snapshot file system is disabled. It is preceded by a message that specifies the reason for disabling.

The snapshot file system should be unmounted. The previous message will identify the reason for disabling the snapshot file system. After the problem is corrected, rerun any backups that failed due to the error.

Message
WARNING: vxfs: vx_check_badblock - *mount_point* file system had an I/O error, setting *VX_FULLFSCK*

When the disk driver encounters an I/O error, it causes a flag to be set in the vfs structure. If this flag is set, then the kernel will set the *VX_FULLFSCK* flag as a precautionary measure. Since no other error has set the *VX_FULLFSCK* flag, the failure probably occurred on a data block.

Make sure a full structural *fsck* is performed at the next administrative period. Check the console log for I/O errors. If the disk has a hard failure it should be repaired as soon as possible.

Message
WARNING: vxfs: vx_resetlog - *mount_point* file system can't reset log

This message is printed when the kernel encounters an error while resetting the log id on the file system. This should only happen if the superblock update or log write encountered a device failure.

If this happens the file system is disabled to preserve its integrity.

The file system should be unmounted and a full structural *fsck*(1M) performed as soon as possible. Check the console log for I/O errors. If the disk has failed, it should be repaired before the file system is mounted for write access.

Message
WARNING: vxfs: vx_inactive - *mount_point* file system inactive of locked inode *inumber*

This message is printed if the *VOP_INACTIVE* is called for an inode while the inode is being used. This should never happen; however, if this situation arises, the file system will be disabled.

The file system should be unmounted and a structural *fsck*(1M) performed as soon as possible. This should be reported as a bug to your customer support organization.

Message
WARNING: vxfs: vx_metaioerr - *mount_point* file system meta data *mount_point* error

A read or a write error occurred while accessing file system meta data. The full *fsck* flag on the file system was set. The message will specify whether the disk I/O that failed was a read or a write. File system meta data includes inodes, directory blocks, the file system log, etc. If the error was a write error, it is likely that some data was lost. This message should be accompanied by another VxFS message describing the particular file system meta data affected, as well as a message from the disk driver containing information about the disk I/O error.

The condition causing the disk error must be resolved. If the error was the result of a temporary condition (such as accidentally turning off a disk or bumping a cable), correct the condition. Check for loose cables, etc. At the next administrative period, run *fsck* on the file system, which will perform a full *fsck* to repair any damage that occurred (possibly with loss of data). In the event of an actual disk error, if it was a read error and the disk driver remaps bad sectors on write, then it may be fixed when *fsck* is run, since *fsck* is likely to rewrite the sector with the read error. In other cases, you will need to repair or reformat the disk drive and restore the file system from backups. Consult the documentation specific to your system for information on how to recover from disk errors. The diskdriver should have printed a message that may serve to discover more information.

Message
WARNING: vxfs: vx_dataioerr - *mount_point* file system data *function* error

A read or a write error occurred while accessing file data.The message will specify whether the disk I/O that failed was a read or a write. File data includes data currently in files and free blocks. If the message is printed because of a read or write error to a file, then another message that includes the inode number of the file should be printed. The message may be printed as a result of a read or write error to a free block, since some operations allocate an extent and immediately perform I/O to it. If the I/O fails, the extent is freed and the operation fails. This message should be accompanied by a message from the disk driver containing information about the disk I/O error.

The condition causing the disk error must be resolved. If the error was the result of a temporary condition (such asaccidentally turning off a disk or bumping a cable), correct the condition. Check for loose cables, etc. If any file data was lost, restore the files from backups. (The file name can be determined from the inode number. See the description of the *ncheck*(1M) command in Chapter "ncheck(1M)" for more information.) If an actual disk error occurred, you will need to make a backup of the file system, repair or reformat the disk drive, and restore the file system from backups. Consult the documentation specific to your system for information on how to recover from disk errors. The disk driver should have printed a message that may serve to discover more information.

Message
WARNING: vxfs: vx_writesuper - *mount_point* file system superblock write error

An attempt to write the file system superblock failed due to a disk I/O error. If the file system was being mounted at the time, then the mount will fail. If the file system was mounted at the time and the full *fsck* flag was being set, the file system will probably be disabled and an adequate message will be printed. Otherwise, if the superblock was being written as a result of a sync operation, then no other action is taken.

The condition causing the disk error must be resolved. If the error was the result of a temporary condition (such as accidentally turning off a disk or bumping a cable), correct the condition. Check for loose cables, etc. At the next administrative period, perform a full *fsck* on the file system to repair any damage that occurred. If an actual disk error occurred, you will need to repair or reformat the disk drive and restore the file system from backups. Consult the documentation specific to your system for information on how to recover from disk errors. The disk driver should have printed a message that may serve to discover more information.

Message
WARNING: vxfs: vx_snap_getbitbp - *mount_point* snapshot file system bitmap write error

An I/O error occurred while writing a snapshot file system bitmap. The snapshot file system is disabled.

The snapshot file system should be unmounted. After the I/O problem is corrected, the snapshot file system can be remounted.

Message
WARNING: vxfs: vx_snap_getbitbp - *mount_point* snapshot file system bitmap read error

An I/O error occurred while reading a snapshot file system bitmap. The snapshot file system is disabled.

The snapshot file system should be unmounted. After the I/O problem is corrected, the snapshot file system can be remounted.

Message
PANIC: vxfs: vx_flush_procs - return from *mount_point*

During startup, the kernel created the *vxflush* daemon, but it unexpectedly died.

Record the messages and the stack trace. Reboot the machine and save a dump.