The New Hitchhiker's Guide to Tru64's AdvFS

Authors:

LMS

JRB

Version 2.2

March 9, 2006

Preface

Introduction

This document describes the inner workings of the Tru64 UNIX Advanced File System. It explains AdvFS concepts and structures, algorithms and design rationales. Diagrams and illustrations are used liberally, and code snippets are avoided.

Use the *Hitchhiker's Guide* as a tool to understand the associated code.

Conventions

This guide uses the following conventions:

bold text	Indicates the introduction of a new term that appears in the glossary.
italic text	Indicates function names.
UPPERCASE TEXT	Indicates error codes and constant variables.
`monospace text`	Indicates commands, utilities, options, attributes, variables, tunables, and structures.
`monospace italics`	Indicates parameters or arguments.

Chapter 1: AdvFS Overview

This chapter provides a high-level overview of the Advanced File System (AdvFS). AdvFS differs from the traditional UNIX File System (UFS). With AdvFS you can modify your system configuration at any time without shutting down the system. AdvFS supports a multi-volume file system, so as your system requirements change you can easily add or remove storage devices such as directly connected disks, software redundant array of independent disks (RAID) volumes, hardware RAID, and storage area networks.

1.1 How AdvFS differs from UFS

In contrast to AdvFS, the UFS model is rigid. A typical UFS maintains a single file hierarchy on a storage device. A UFS file is bound to the device it was created on, so it cannot be moved to another disk (to balance the disk's I/O load, for example).

From a user's perspective, AdvFS looks like any other UNIX file system. AdvFS presents storage to the user as a hierarchy of directories and files. Users can use the `mkdir` command to create new directories, the `cd` command to change directories, and the `ls` command to list directory contents. AdvFS logical structures, quota controls, and backup capabilities are based on traditional file system design. AdvFS replaces or eliminates several standard commands, such as `newfs`, `dump`, `restore` and `fsck`.

AdvFS is the file system for the Tru64 cluster configuration. Cluster operation is transparent. AdvFS running on a cluster, with very few exceptions, looks no different from AdvFS running on a single node.

1.2 Basic AdvFS Concepts

AdvFS presents storage to the user as a hierarchy of directories and files much the same as most other UNIX file systems do. However, the relationship between files and their physical storage in the AdvFS file system is different from other UNIX file systems. AdvFS consists of two distinct layers: the **logical file hierarchy layer** and the **physical storage layer**. The logical file hierarchy layer implements the file-naming scheme and POSIX-compliant functions such as creating and opening files, or reading and writing files. The physical storage layer implements write-ahead logging, caching, file storage allocation, file migration, and physical disk I/O functions. Figure 1-2 shows the relationship between the two layers.

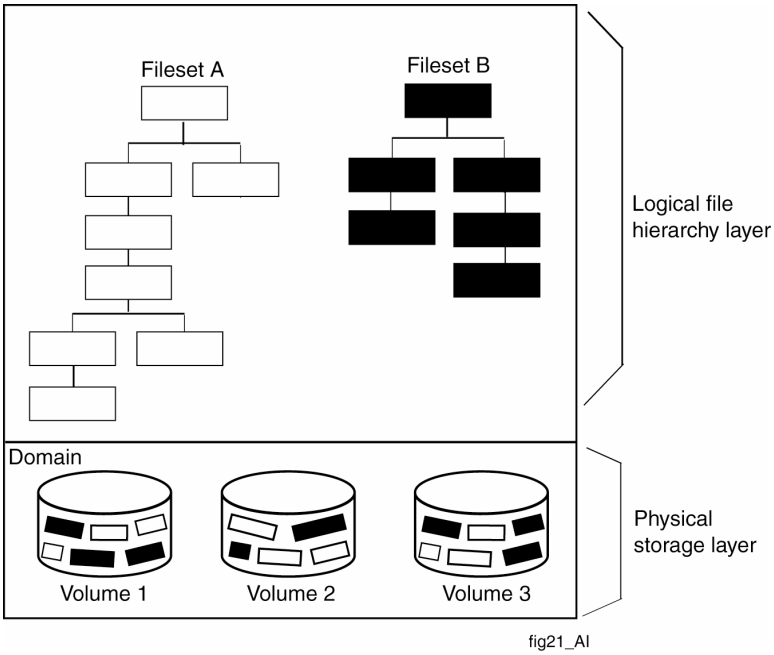


Figure 1: Logical file layer and physical storage layer

The logical file hierarchy layer consists of

- **Files**, which are the entities that store information.
- **Directories**, which are collections of files that are logically related.
- **Filesets**, which are related sets of directories and files that are mounted.

The physical storage layer consists of

- **Domains**, which are collections of volumes.
- **Volumes**, which provide the physical storage.
- **Bitfiles**, which are the physical representation of files.

Files are the user's view of bitfiles. Bitfiles are how files are implemented in AdvFS.

Because AdvFS separates the file hierarchy from the storage, each component can be managed independently. AdvFS files are not bound to specific devices to facilitate migrating files among the storage in a domain. You can move files between volumes without changing path names for your files. Because the path names remain the same, the action on the physical level is completely transparent at the logical file hierarchy level.

AdvFS implements two unique file system concepts: filesets and domains. A fileset (see Chapter 3) follows the logical structure of a traditional UNIX file system. It is a hierarchy of directory names and file names, and it is what you mount on your system. AdvFS goes beyond the traditional file system by allowing you to create multiple filesets that share a common pool of storage called a domain (see Chapter 2). A domain represents the physical storage layer. It is managed separately from the directory structure. You can add or remove volumes within a domain without affecting the directory structure.

A volume (see Chapter 2) is any mechanism that behaves like a UNIX block device. When first created, all domains consist of a single volume. You can transform a single-volume domain into a multi-volume domain by adding one or more volumes to it.

AdvFS has the ability to perform file **striping** (see Section 5.4). File striping allows a file to be spread evenly across several volumes within a domain. As data is appended to the file, the data is spread across the volumes. This increases the sequential read/write performance because I/O requests to the different disk drives can be overlapped.

You can backup your AdvFS file system by using a fileset **clone** (see Section 5.1). An AdvFS fileset clone is a read-only copy of an existing fileset created to capture data at one instant in time. When you clone a fileset (create a fileset clone), the utility copies only the structure of the original fileset, not the actual data. When a file is modified, the file system copies the original, unchanged data to the AdvFS fileset clone. Therefore a copy of the system as it was at the time of creating the clone remains for the life of the clone.

Fast recovery is a distinguishing feature of the AdvFS. AdvFS is a log-based file system that employs **write-ahead logging** to ensure the integrity of the file system. Modifications to the **metadata** are completely written to a transaction log file before the actual changes are written to disk. The content of the transaction log file is written to disk at regular intervals. During crash recovery, AdvFS reads the transaction log file to confirm file system transactions. All completed transactions are committed to disk and uncompleted transactions are undone. The number of uncommitted records in the log, not the amount of data in the file system, dictates the speed of recovery. Recovery usually takes only a few seconds. Traditional UNIX file systems rely on the `fsck` utility to recover from a system failure, which can take hours to check and repair a large file system.

1.3 The FAS and the BAS

AdvFS can also be described in terms of the **File Access Subsystem (FAS)** and the **Bitfile Access Subsystem (BAS)**. The FAS is responsible for managing the logical file hierarchy and the BAS is responsible for managing the hierarchy's physical storage representation. Figure 1-3 shows each functional component's task and how the components interact with each other and with other components (like users and disks) to accomplish their tasks.

Figure 1: Storage Usage and Management

Who?	What?	How?
1. User	2. Manages information	3. Uses files to store information
4. File Access Subsystem	5. Manages files and directories	6. Uses bitfiles to store files and directories
7. Bitfile Access Subsystem	8. Manages bitfiles and their storage on disks	9. Uses disks to store bitfiles
10. Storage Devices	11. Stores bitfiles	12.

The line separating FAS and BAS layer functions has become fuzzy over the years. Although the Tru64 code base still has `msfs/bs` and `msfs/fs` directories, many files and functions no longer map strictly into these layer distinction

Chapter 2: Domain and Volume Concepts

2.1 Basic Concepts

A **fileset** follows the logical structure of a traditional UNIX file system. It is a hierarchy of directory names and file names, and it is what you mount on your system. AdvFS goes beyond the traditional file system by allowing you to create multiple filesets that share common storage called a domain.

A **domain** is the physical storage layer of the AdvFS file system. It is a defined pool of storage that can contain one or more volumes. Because this storage is managed separately from the directory structure, you can expand and contract the size of the domain by adding or removing volumes. The directory structure is not affected.

A **volume** is any mechanism that behaves like a UNIX block device. An AdvFS volume can be a raw disk partition, an entire disk, an aggregate volume provided by Logical Storage Manager (LSM), a storage area network (SAN), or a hardware or software redundant array of independent disks (RAID) storage. When first created, all domains consist of a single volume. You can then transform a single-volume domain into a multi-volume domain by adding one or more volumes to it.

Multi-volume domains increase the storage available for the filesets and allow for preventative disk maintenance. You can add volumes immediately after creating the domain, even before creating and mounting filesets. To perform preventative disk maintenance, you can add a new volume to the domain (see section 2.8), migrate your files to the new volume, then remove the old volume (see section 2.9).

The most significant difference between a typical UNIX filesystem and AdvFS is that AdvFS uses a two-level lookup scheme to locate a file's data. AdvFS looks up the file's name in the directory to get the file's **tag** (which uniquely identifies a file within a fileset). The tag (and the associated fileset ID) is then used to lookup the file's metadata (equivalent to file descriptor). This is done by looking up the tag in the **fileset's tag file** to get the metadata's disk identifier and the location of the metadata in the disk's Bitfile Metadata Table (BMT). The file's metadata entry in the BMT is called a **mcell** (for metadata cell). The mcell is used to locate the file's data and to maintain the file's attributes. The figure below shows the AdvFS two-level lookup scheme.

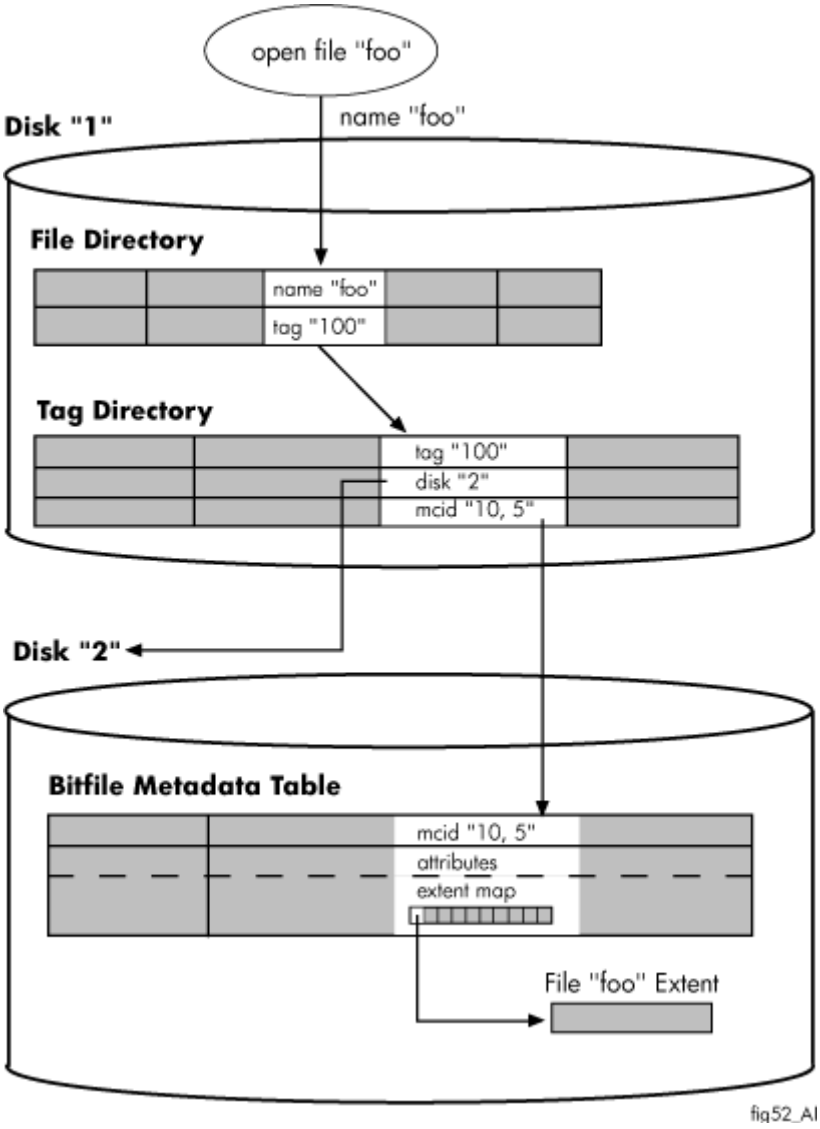


Figure 2: Two-level file lookup

The tag concept is a critical design feature that allows AdvFS to seamlessly move files among the disks in a domain. This is important because it allows a decoupling of a file name from its on-disk structure. A file's directory entry does not need to be updated when a file is moved to a different disk. Only the tag file needs to be updated when a file is moved.

2.2 Physical Volume Layout

As part of domain creation, some volume initialization takes place. Below is an illustration of the physical disk layout for domain version 4. A domain version number (**DVN**) is associated with each domain, and all domains created on Tru64 OS Version 5.0 carry a DVN of 4. Domains created prior to version 5.0 carry a DVN of 3.

The terms in this illustration will be discussed later in this chapter.

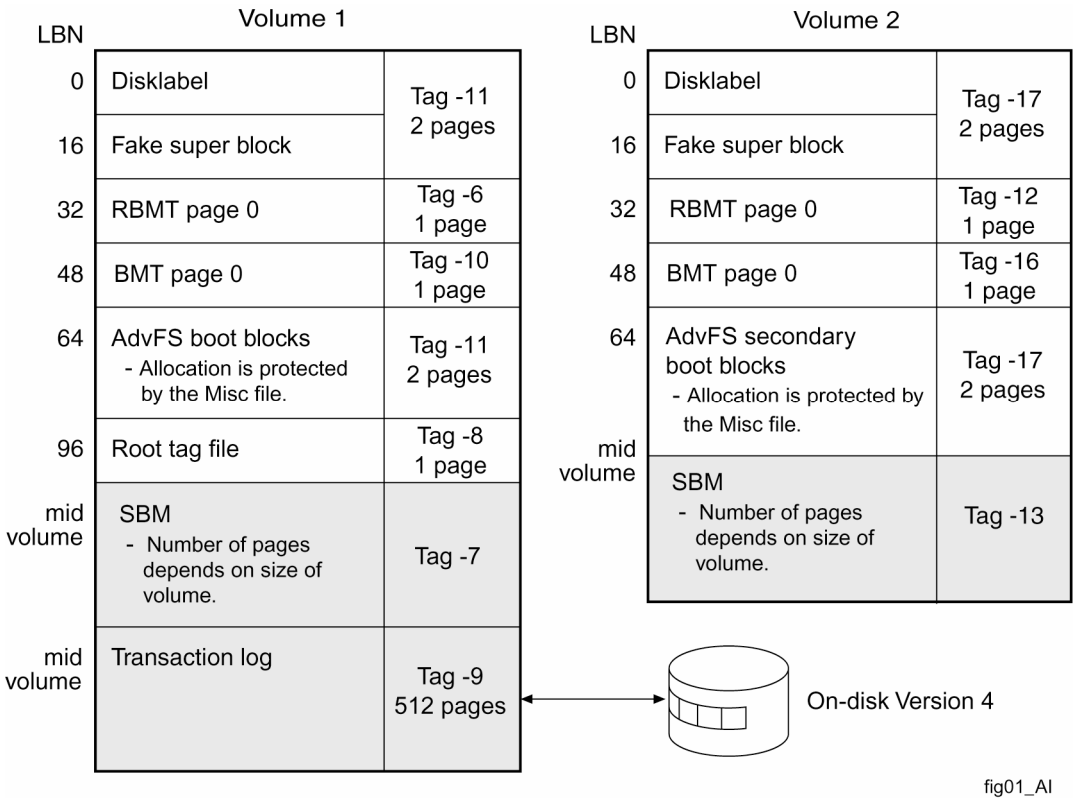


Figure 3: DVN 4 physical disk layout

2.3 On-disk structures

Each volume in a domain consists of the following on-disk structures:

- Reserved bitfile metadata table (RBMT) (see section 2.3.2)
- Bitfile metadata table (BMT) (see section 2.3.1)
- Storage bitmap (SBM) (see section 2.3.3)
- Miscellaneous bitfile (see section 2.3.6)

Each domain has the following on-disk structures. There is only one of each structure per domain, and they can reside on any volume in the domain:

- Root tag file (see section 2.3.5)
- Transaction log (see section (2.3.4)

Each fileset has the following structures, which can reside on any volume in the domain.

- Tag file (see section 3.2.1)
- Fragment file (see section 3.2.2)

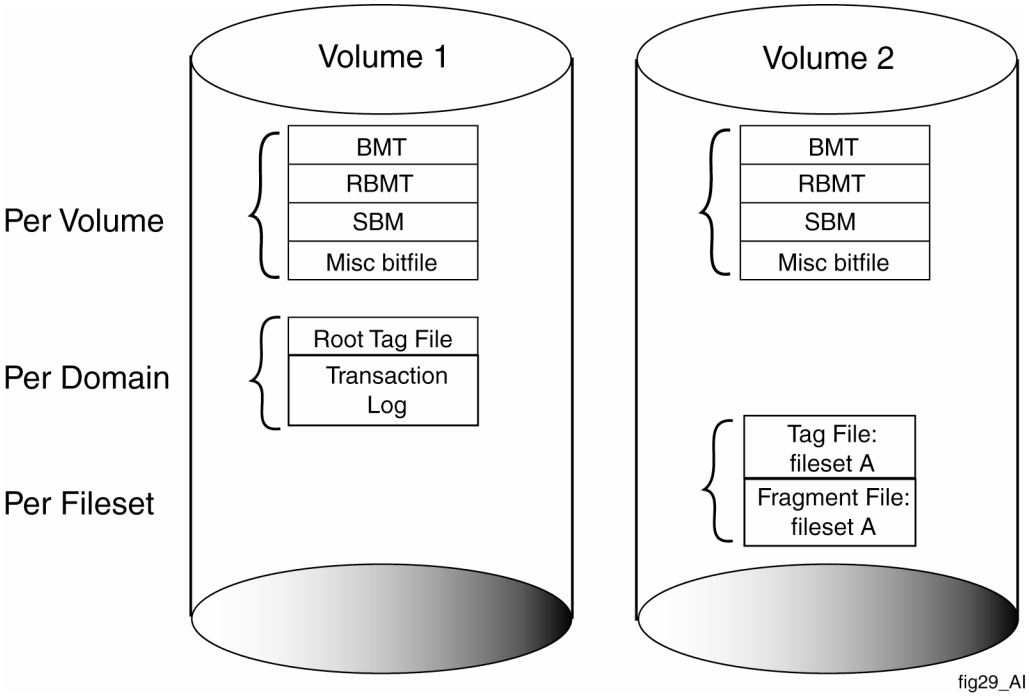


Figure 4: Summary of volume, domain, and fileset specific structures

2.3.1 Bitfile Metadata Table (BMT)

The **BMT** is used to store file metadata for **non-reserved files**. Information stored in the BMT includes file attributes, extent maps, fileset attributes, and POSIX file statistics. Examples of non-reserved files are user files, directories, fileset tag directories, and the frag file. There is one BMT per volume.

The BMT is an array of 8KB **pages.** Each page consists of a header and an array of fixed-size metadata cells (**mcells**), where each mcell contains one or more variable-length **records**. The records are of specific types (bitfile attributes, extent map, etc.) which are defined in `src/kernel/msfs/msfs/bs_ods.h`.

A series of contiguous 8K pages in a file is stored as an **extent**. Extents are groups of on-disk contiguous 8K pages managed by **extent maps**. AdvFS uses the BMT mcells to manage and describe non-reserved files' attributes and storage.

Each file has one **primary mcell** in one of the BMT files of the domain. The primary mcell contains the file attributes, a **primary extent map record**, and a pointer to the next mcell (if there is one). **Secondary mcells** can be used to contain additional mcell records, which can include the POSIX file statistics, fileset attributes attached to tag directories, additional extent records, and POSIX symbolic links. Secondary mcells are linked together such that the head of the linked list is the primary mcell.

An mcell functions similarly to an **inode**. It holds permissions, size, extent information, and link count. Each mcell is 292 bytes, so 28 mcells (plus a 16-byte header) can fit on an 8K page. The following figure compares the user's view of a logical file to the physical on-disk view of the file. The BAS represents the on-disk representation of this file by one or more mcells pointing to extents:

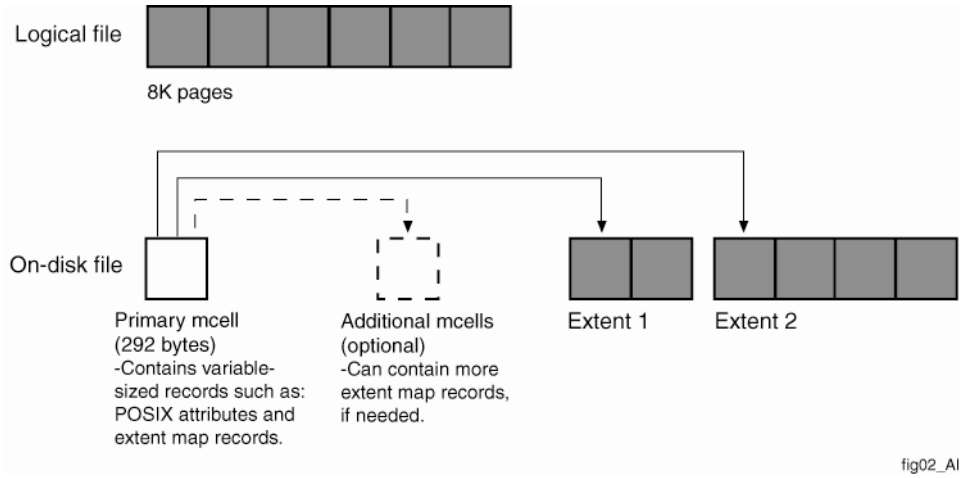


Figure 5: Physical and logical file views

If a BMT page contains at least one free mcell then the page is on a **free page list**. The free page list head is maintained in the first mcell in BMT page 0. Each BMT page also maintains its own **free mcell list**, which consists of mcells within the page that have free records. The free mcell list head is maintained in the header of each BMT page.

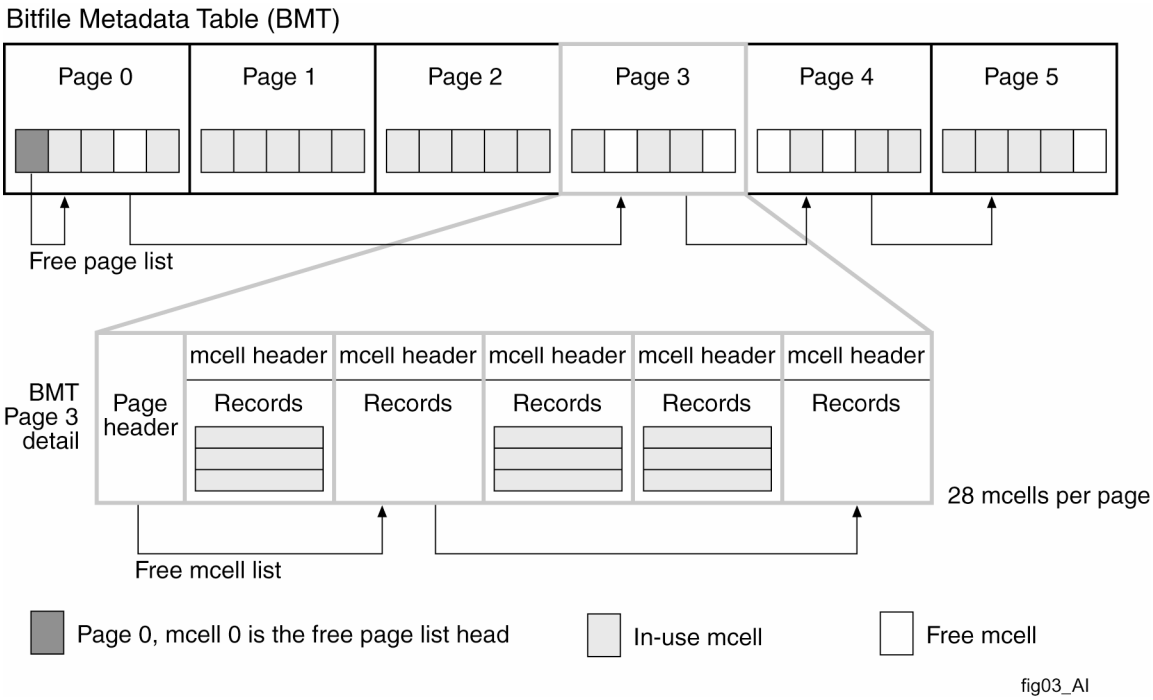


Figure 6: BMT page details and linked lists

The mcells are addressed by the following **mcell address tuple**:

<**volume index**, BMT page number, mcell's index within the BMT page>

Records within mcells are found by a sequential scan within the mcell. This search is very short because most mcells contain only one or two records. AdvFS does not support deletion of mcell records.

2.3.2 Reserved Bitfile Metadata Table (RBMT)

The **RBMT** functions like the BMT, except that it contains all the mcells needed to describe the **reserved metadata files** stored on that volume. Reserved files described by the RBMT include the RBMT itself, the SBM, the root tag directory, the transaction log, the BMT, and the miscellaneous bitfile. There is one RBMT per volume.

AdvFS relies on the RBMT during domain activation and especially during crash recovery. The RBMT always starts at disk sector 32, which allows AdvFS to quickly find the mcells describing all the reserved files on the volume. The information in the RBMT typically fits inside a single 8k page. However, since the BMT is the one reserved file that tends to grow through time, and its extent map records are kept in the RBMT, the RBMT may need to expand into additional pages to accommodate the added extent records for the BMT. This is done by adding additional RBMT pages and linking their location through mcell 27 in the previous RBMT page. This is the mechanism that was added in DVN 4 domains to eliminate the problem seen in previous versions where the BMT could no longer grow because all BMT extent maps were required to fit inside BMT page 0. This resulted in domains in which files could not be added even though there was still storage on the disk. The use of the RBMT as an extensible container for BMT extent records was the solution to this problem.

There are 2 ways to determine the last page of the RBMT. The second to the last page of the RBMT always contains a mcell 27 with a BSR_XTRA_XTNTS record that describes the last RBMT page. The next pointer in mcell 27 on the second to the last page can then be nil (on pre-5.1B domains) or can point to mcell 27 of the last page of the RBMT (5.1B and after).

The following figure shows the layout of RBMT page 0 in detail:

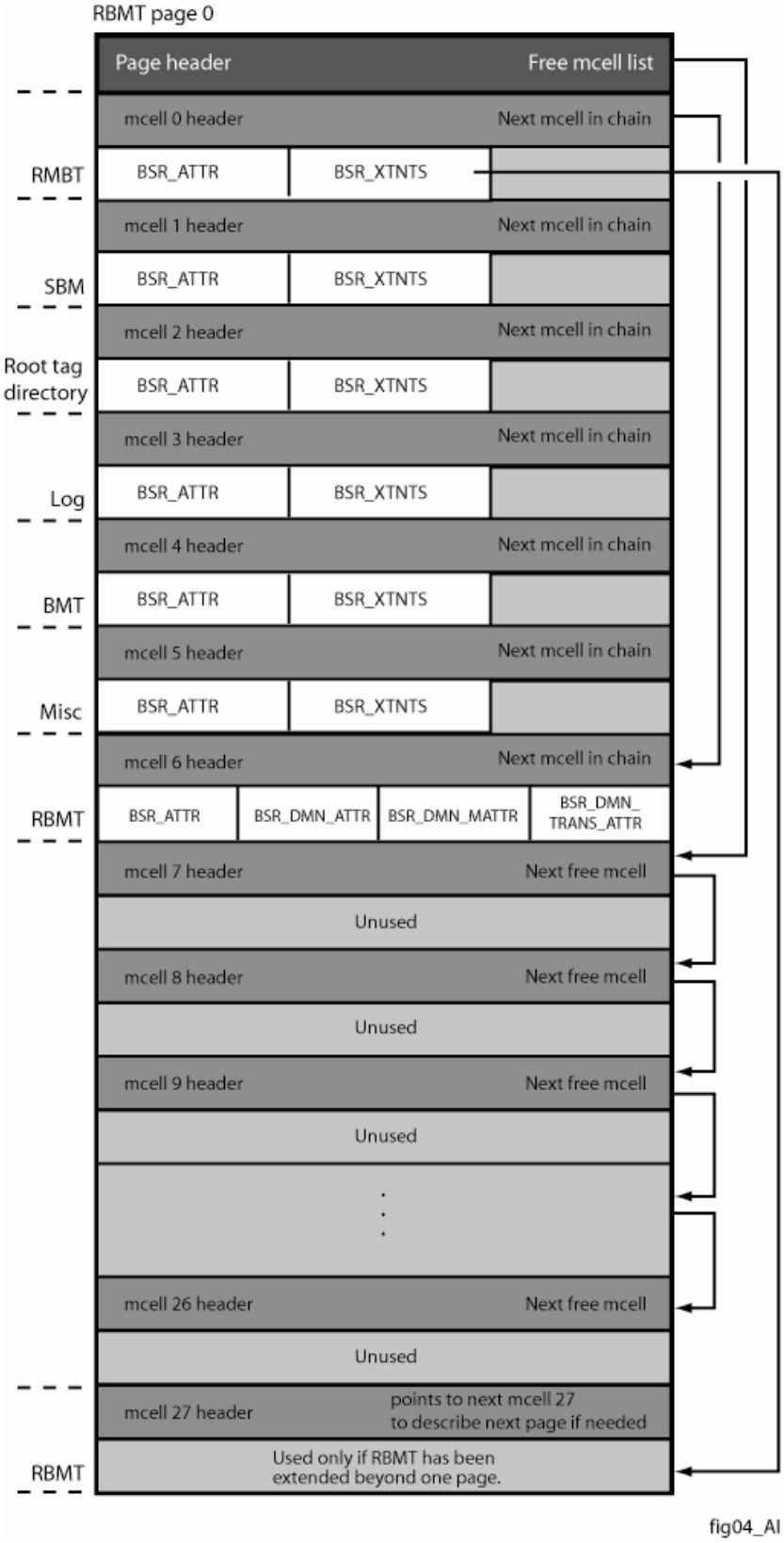


Figure 7: RBMT layout

The RBMT mcells are allocated as shown in the following table:

RBMT mcell number	Use
0	Primary mcell for RBMT
1	Primary mcell for SBM
2	Primary mcell for root tag directory
3	Primary mcell for transaction log
4	Primary mcell for BMT
5	Primary mcell for miscellaneous bitfile
6	Secondary mcell for RBMT. Contains domain and volume attributes for RBMT
7 through 26	Extra mcells used for additional extent maps for the reserved files
27	Used to extend the RBMT to the next page if page 0 becomes full

2.3.3 Storage Bitmap (SBM)

Each AdvFS volume contains a storage bitmap, which keeps track of allocated disk space. In AdvFS terminology, a block is a 512-byte sector, a page is 16 blocks, and one page is 8192 bytes. Each bit in the storage bitmap represents one page. If the bit is set, the page is allocated to a bitfile; if the bit is clear, the page is free (available for allocation).

The storage bitmap is structured as an array of 8KB pages where each page consists of an array of 32-bit integers (each bit represents a page). Each page contains a header (`struct bsStgBm`) with two fields: a 32-bit LSN number and a 32-bit XOR checksum, leaving 65,472 bits per page to map pages on the volume. The actual size of the SBM depends on the size of the volume. The SBM is accessed primarily during file creation, deletion, or migration. It can also be accessed during file extension if the free space cache has been depleted. More on the free space cache shortly (see section 2.4.4).

The primary mcell describing the SBM for a volume is found in mcell 1, page 0 of that volume's RBMT.

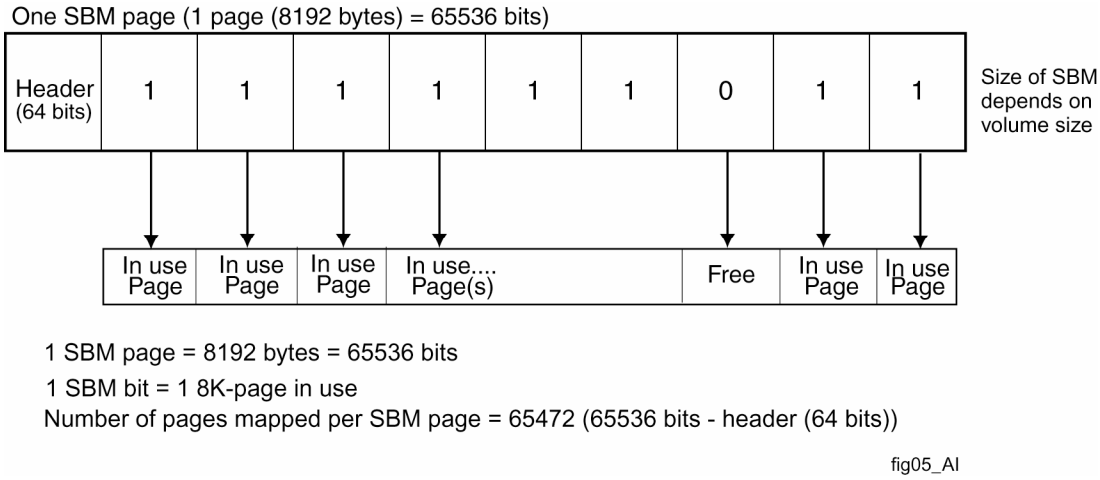


Figure 8: SBM to physical page mapping

2.3.4 Transaction Log

The **transaction log** is the on-disk structure used by the **Flyweight Transaction Manager** and Logging Subsystem (also called the **Logger**) to ensure the integrity of the file system. Modifications to the metadata (file system structures) are written to the transaction log before the metadata changes are written to disk. The log provides recovery capabilities following an error, system crash, or media failure. During crash recovery, AdvFS reads the transaction log to confirm file system transaction completion. All completed transactions are committed to disk and incomplete transactions are undone.

The default size of the log is 512 pages (4 MB), but this can be changed when the domain is created (see Section 2.5) The on-disk format of the log can be viewed in two ways: physically and logically.

The physical view is a series of 512-byte sectors. A sector is the disk unit that can be written atomically, and 512-bytes is the sector-size supported by Tru64 UNIX. The sector-based view of the log is necessary when needing to assure log consistency in the event of a crash. Since a page consists of 16 sectors, then it is possible that less than 16 sectors will be written to the disk during a system failure. The problem is not that all 16 sectors didn't get written, but that the logger must be able to detect that they didn't all get written. To solve this problem the logger puts a unique page ID (an LSN) at the beginning of each sector in the log page (see next figure). Then, when the log page is read during recovery, the logger checks to see if the LSNs in each sector are all the same. If they are not, then the logger knows the page was not completely written. To avoid complex code in the logger to prevent stepping on the portion of each sector that is reserved for the ID, the logger ignores the IDs when writing records to a log page. Then, just before unpinning the log page, the logger copies the bytes from the area where the ID is to be written into a reserved "save area". After this data has been saved, it then writes the ID into the beginning of each sector. That way the logger isn't overwriting (and losing) data from the logged records. Later, when the log page is read, the logger overwrites the IDs with the original values obtained from the page's "save area". This "save area" is actually in the trailer for each page, which is part of the logical view.

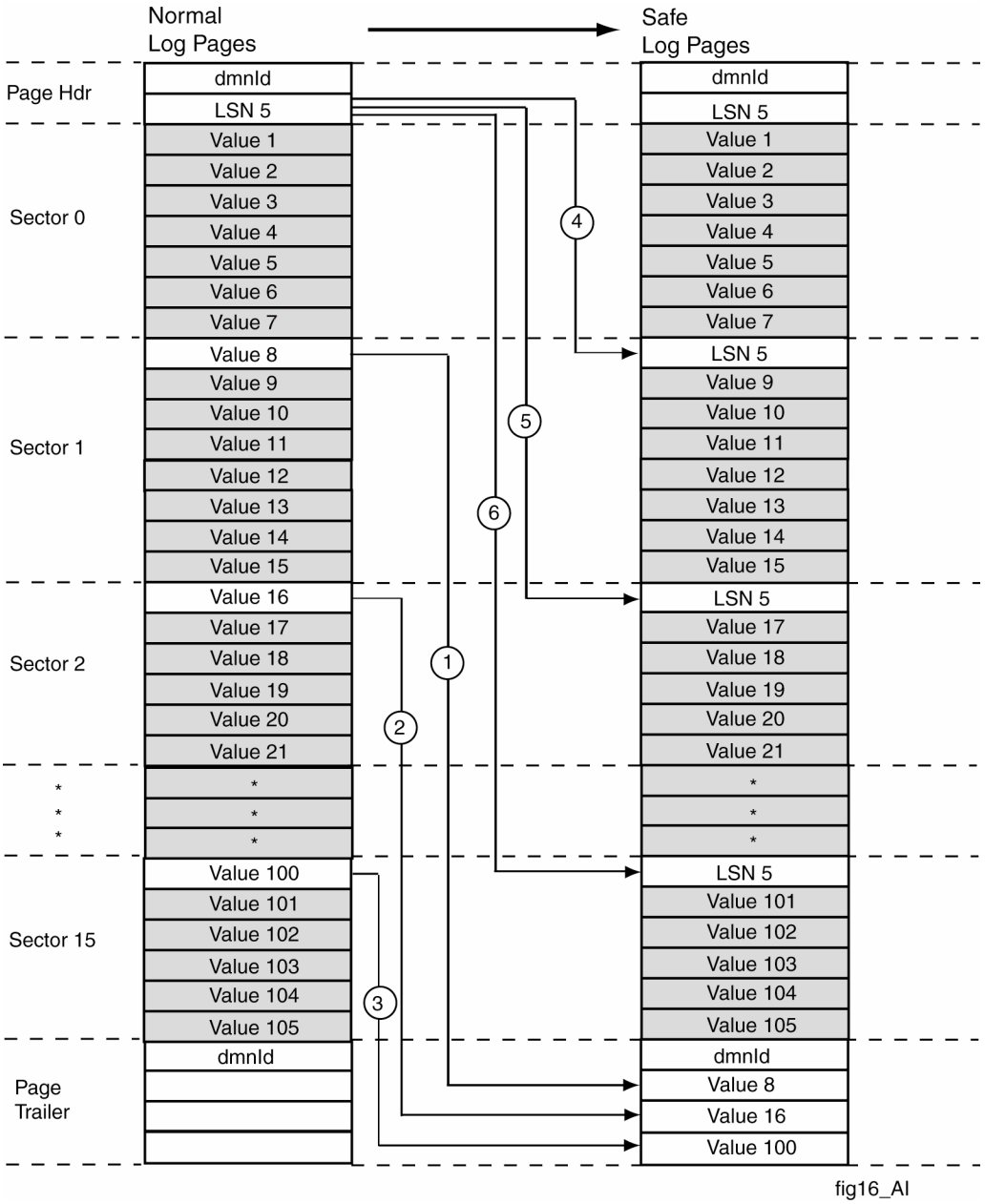


Figure 9: Physical view log page translation used to ensure on-disk log consistency

In the logical view, the log is a circular array of 8k pages that wraps around to page 0 after page 511 has been written. The log has a beginning (tail in circular list terminology) and an end (head). Records are written to the end of the log (head); the beginning of the log (tail) contains the oldest record in the log.

The Logger uses a logical sequence number (**LSN**) to uniquely identify each log record. When the log is first created the log's current LSN is initialized to 2. Each time a record is written it is assigned the current LSN and the current LSN is incremented by 2. LSNs can wrap so there is no limit on LSNs (see Chapter 9 for more information on infinite LSN numbers).

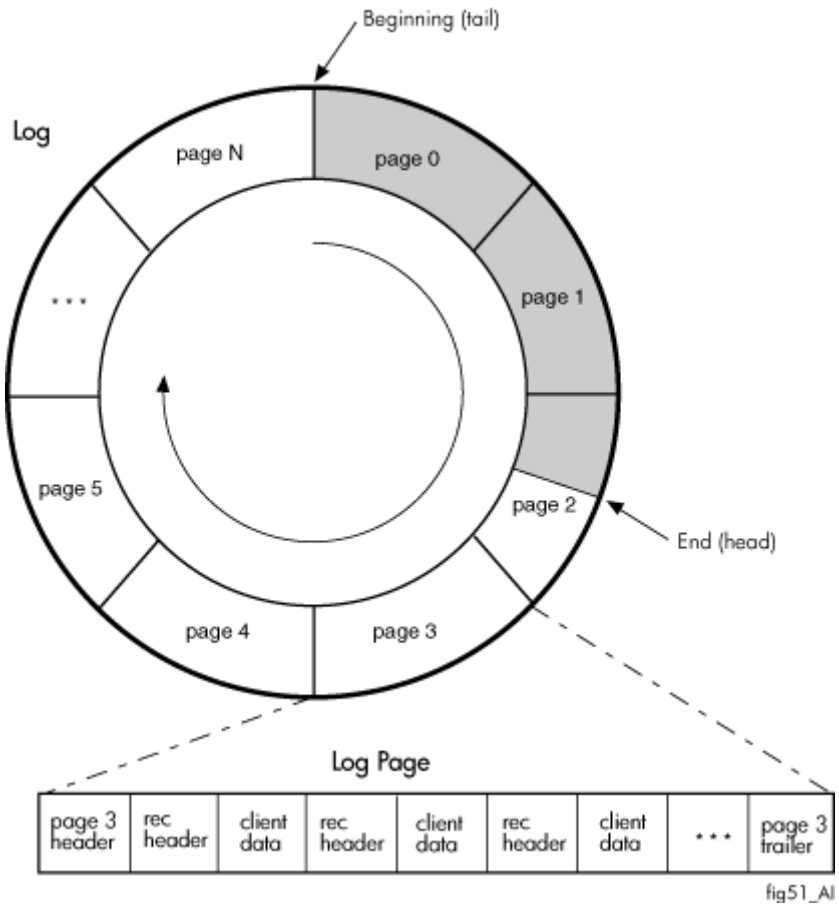


Figure 10: Log as viewed in circular fashion

Each page of the log is composed of a header, some number of variable-sized records, and a trailer:

1. The header consists of the first record's LSN. This is used whenever a domain is activated to locate the end of the log.
2. Some number of variable-sized records. Each record contains a fixed-size header and a variable amount of client data. Each log record is accessed by a unique address. A record address consists of the page number of the page that contains the record, the record's word offset into the page, and the record's LSN (which is the component of a record address that makes the address unique in time).
3. The trailer consists of the “saved area” that was discussed in the physical view of the log; this is for saving data from the beginning of each sector in the log page.

Whenever a record is written to the log, the Logger returns the record's address and its LSN to the client. When reading a log record the client must provide the record's address. The Logger uses the LSN in the record's address to verify that the address is valid (each record header contains the record's LSN). Details on the algorithms used to manage the transaction log are provided in Chapter 9.

2.3.5 Root Tag File

Because there can be many filesets within a domain, there must be a way to locate the tag file that is associated with each fileset. This is accomplished using the **root tag file**. Every domain has a single root tag file that gives the location of the primary mcell associated with the fileset tag file for each fileset in the domain. The root tag file is similar to a normal fileset tag file. It contains an entry for each fileset in the domain, which contains the corresponding primary mcell to locate the fileset's tag file. (see section 3.2.1 for more information on the Fileset Tag File). The fileset number is the index into the root tag file.

The root tag file is at mcell 2 in page 0 of the RBMT (see Section 2.3.2). Although each volume has an RBMT, only one volume in the domain contains a valid root tag file. If the root tag file was ever moved off its original volume, there may be traces of old "invalid" root tag files on other volumes.

To determine which root tag file is currently valid look at the **domain mutable attributes record** on each volume in the domain. This record is found in mcell 6 in page 0 of the RBMT. Any volume that ever housed the log (and therefore root tag file) also has a domain mutable attributes record. The volume containing the domain mutable attributes record with the highest sequence number is the volume containing the latest domain attributes, log, and root tag file. The domain mutable attributes record contains the tag of the current root tag file.

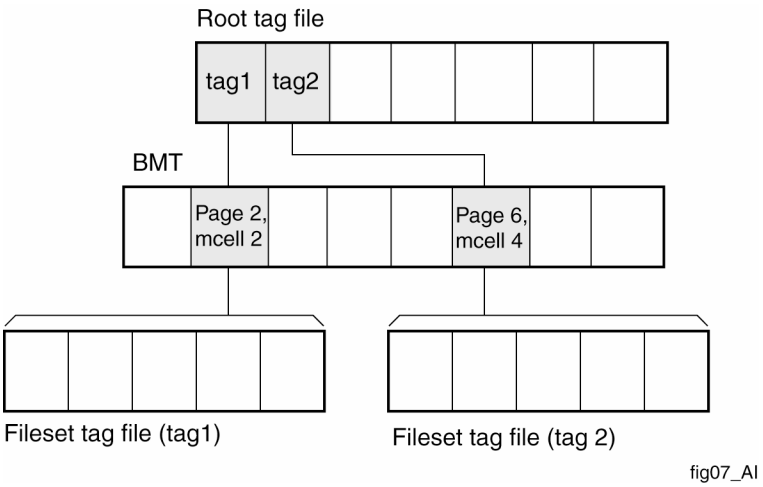


Figure 11: Root tag file to fileset tag file

2.3.6 Miscellaneous Bitfile

The miscellaneous bitfile represents a fake **superblock** (a block used by UFS which contains essential static filesystem information needed at boot time) and other disk overhead structures typically found on a volume. The miscellaneous bitfile is used to reserve sectors on a volume that do not represent actual AdvFS metadata, including the disk label, the UFS boot blocks, the fake superblock (for AdvFS, it contains the **magic number**), and the AdvFS boot blocks. The magic number is 0x11081953 at offset 1372 (0x55c) into the superblock. Each volume has its own miscellaneous bitfile.

2.4 In-memory structures

In general, for every on-disk structure a corresponding in-memory structure exists. The in-memory structures are used to cache parts of their corresponding on-disk structures, to maintain state information,

and to contain related lock structures. Some on-disk structures are represented by several in-memory structures.

Before looking at the specific structures, there is a mechanism that is used in several of the common AdvFS in-memory structures that is useful to understand. The in-memory structures for domains, bitfile sets, fileset nodes, volumes, access structures, and buffers all contain a **magic field**. This field is used for several purposes. First, each has a specific, preset value (e.g. `0xadf003` for a domain) that can be found in `bs_public.h`. Thus, when debugging a crash dump and displaying a structure from a memory pointer, it is immediately obvious if the address is, in fact, a pointer to the type of object specified. If the magic number is displayed correctly, the pointer was valid. If the magic number is not valid, then the address is not a pointer to that type of structure. A second use of the magic field is to understand if a given in-memory structure is legitimately allocated, or if it has been freed. When the structure is allocated, the magic field is given the preset value for the type of structure that it is. This value remains the same while the structure is in use. When the structure is no longer needed, and it is freed, then a special bit (`MAGIC_DEALLOC = 0x00080000`) is set in the magic field. If a crash dump is being investigated and there is code that is manipulating a structure that has this bit set, then it is known that this structure has already been freed, and the use of this structure is no longer valid.

The following illustration shows the relationship between the domain and volume in-memory structures discussed in this section:

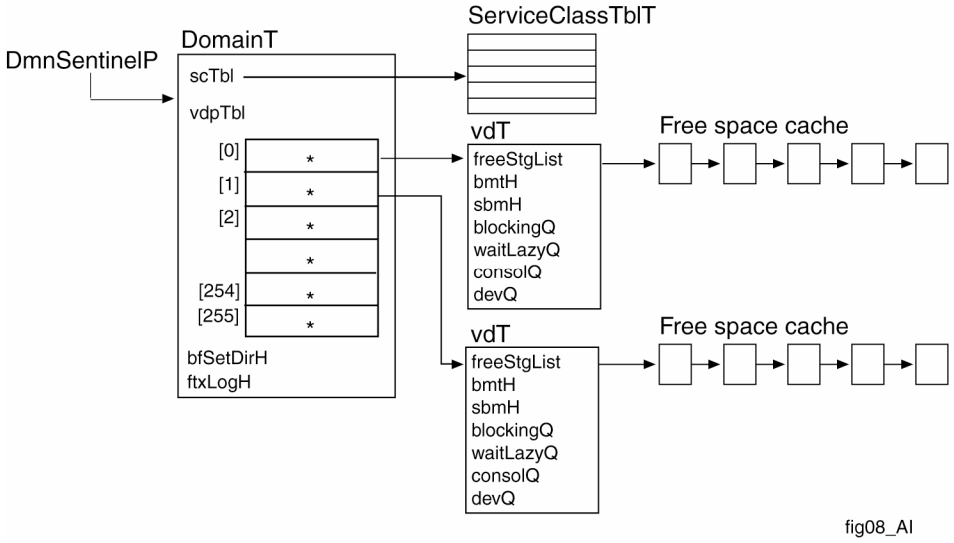


Figure 12: In-memory domain and volume structure relationship

2.4.1 DomainT Structure

Each active domain on a system has an associated `domainT` structure, allocated when its first fileset is mounted. The first domain structure can be found off the `DmnSentinelP` global variable. All active domains are on a circular doubly-linked list using the `dmnFwd` and `dmnBwd` fields within the `domainT` structure. When a domain's last fileset is unmounted, the `domainT` structure is freed (see Section 2.10). Domains that are activated also have their domain IDs hashed and inserted into a global domain hash table (`DomainHashTbl`).

Key items stored in the domain structure include:

- Domain state and ID
- Forward and backward pointers to other domains on this system.
- Volume descriptor array
- Ordered list of dirty metadata buffers that need to be flushed
- Transaction table
- Pointer to the service class table
- Pointer to the transaction log's access structure and state information
- Pointer to the root tag directory's access structure

2.4.2 Volume descriptor array and vd Structure

A volume descriptor array (`domainT.vdpTbl[]`) is used to locate the disks in a domain. This array is a fixed-size, holding 256 (`BS_MAX_VDI`) elements. Each array element is a pointer to an active volume descriptor, or `vd` structure. Note that this array can be sparse; there may be NULL pointers interspersed among the valid pointers.

Each volume descriptor consists of information that describes a disk. The `vd` structure contains the following information as well as other volume-specific fields:

- Volume state and index (`vdIndex`)
- Pointer to the volume's free-space cache
- I/O Queuing information
- Pointers to the volume's RBMT, BMT, and SBM bitfiles
- Pointer to the device vnode
- Physical characteristics of the device

The value of `(vdIndex – 1)` is the offset into the domain's `vdpTbl[]` array in which the pointer to this volume is held.

2.4.3 Service Classes

Each domain has a **service class** associated with it. Each volume added to the domain is added to the service class and removed when the volume is removed from the domain. Service classes are used to determine which volumes within a domain can allocate file storage for new data or data migration. They act as a locking tool when volumes are removed from a domain. Removing the volume from the service class is one of the first steps in the volume removal algorithm. This system prevents any storage from being allocated on the volume being removed, with the exception of a few necessary metadata allocations.

There is one service class table (`ServiceClassTblT`) per domain, and the table contains one service class entry (`scEntry`) per service class in the domain (with today's functionality, that is always one). The entry contains a linked list of the volume indexes in the service class and the volume index of the last

volume used for new storage or mcell allocation in the service class. A volume's corresponding disk structure is found by using the volume index to locate the appropriate volume descriptor pointer in the `vdpTbl` of the domain structure.

The original intent of service classes was quite broad. When an AdvFS disk is initialized, it is assigned to a service class, which defines the storage services the disk will support. AdvFS then allocates storage on a disk that provides the services required by a particular file. Service classes were never expanded beyond use as a way to group volumes into domains, and further expansion of the service class concept would not provide any needed functionality. Many of the service class functions and structures can be removed or simplified in order to clean up the code. For example, we don't need a binary search to find the service class entry (there's only one), and we don't need the disk index list to be a linked list of fixed sized arrays (done for scaling). The key functionality that should be kept is allowing removal of a volume from a service class as a way to avoid new allocations on that volume.

2.4.4 Free Space Cache

To avoid costly I/Os and SBM scanning when searching for free space on a volume, AdvFS uses an in-memory **free-space cache** to keep track of free space on a volume. It is a cache because it has a limited number of entries and does not represent all the free space on a volume.

The cache is a linked list of free space extents. Each extent is represented as a `stgDescT` structure with fields for a starting block in the SBM and the number of contiguous free blocks representing the extent. Note that the `stgDescT` fields are described in terms of clusters, and one cluster is equivalent to one SBM block (or bit), which is one 8K page. The entries in the cache are currently sorted by virtual disk block. ***Potential Enhancement:*** *The free space list could be on two lists: one sorted by disk block number and the other by free space extent size. This allows use of the best-fit algorithm and also allows for space consolidation when new free space is added to the cache.*

When a disk is mounted, its free-space cache is initialized to be empty. The free-space cache is populated whenever it is empty and a request for file storage comes into the storage allocation system. The cache is also repopulated immediately after an explicit invalidation request. The cache is populated by scanning the SBM and creating cache entries for free space extents in the bitmap. When a file requests storage, the cache tries to satisfy the request by delivering part of an entry or several entries until the request is satisfied or the cache is emptied. The cache entries are delivered to files in the order they exist in the cache. The cache is not searched for the closest extent or the best fit for a request.

When the cache cannot satisfy the storage request, all its entries are discarded and it is refilled from the SBM. No entries are retained because there is no provision to merge an entry found by reading the SBM with an entry already existing in the cache.

The free-space cache is a one-way pipe; space can be removed from it (allocated) but it cannot be put back. This rule exists in support of **transactionally consistent storage deallocation,** which requires that any storage deallocation transactions must be committed to the on-disk log before the deallocated space can be reallocated to a different file. From the SBM's perspective, storage deallocation modifies the bitmap only (not the free-space cache) and the SBM modifications are logged. To maintain transaction consistency, the deallocated space must not appear in the free-space cache until the associated transaction records are committed to the on-disk log. Therefore, the transaction log must be flushed synchronously before the bitmap is scanned to refresh the free-space cache. Consider the following example if this rule were not followed. Storage is deallocated from File A, and returned to the free space cache. This storage is then reallocated and given to File B. Meanwhile, the storage deallocation transaction is aborted, and the storage is returned to File A. However, this storage has already been reallocated to File B, so we end

up with a situation where the same piece of storage is actually allocated to two different files. This must never happen, so storage is only reallocated after its deallocation transaction has been committed.

The main ramification of this transaction consistency rule is that the free-space cache does not know about deallocated space until the next refresh of the cache. This means that it is not possible to coalesce deallocated space with existing free space extents in the cache. Therefore, the cache does not always contain the largest and possibly the most optimal free space extents. This can cause AdvFS to generate disk fragmentation (refer to Section 4.7.9 for more discussion on fragmentation).

When the file is closed, any preallocated storage at the end of the file is freed. The freed storage is returned to the storage cache the next time the cache is loaded. The freed storage cannot be returned to the storage cache immediately because data corruption could occur if a second file allocated and wrote data to the storage and the system crashed before the log entry for the storage-freeing transaction was written to disk.

If a utility closes a file (after extending it), the next contiguous storage is most likely the storage that is freed on closure. If the utility then reopens the file to append more data, the storage allocated to the file (from the cache) is likely to be non-contiguous with the last page written. This causes fragmentation in files that are repeatedly opened, appended, and closed. This is exactly the model that NFS uses.

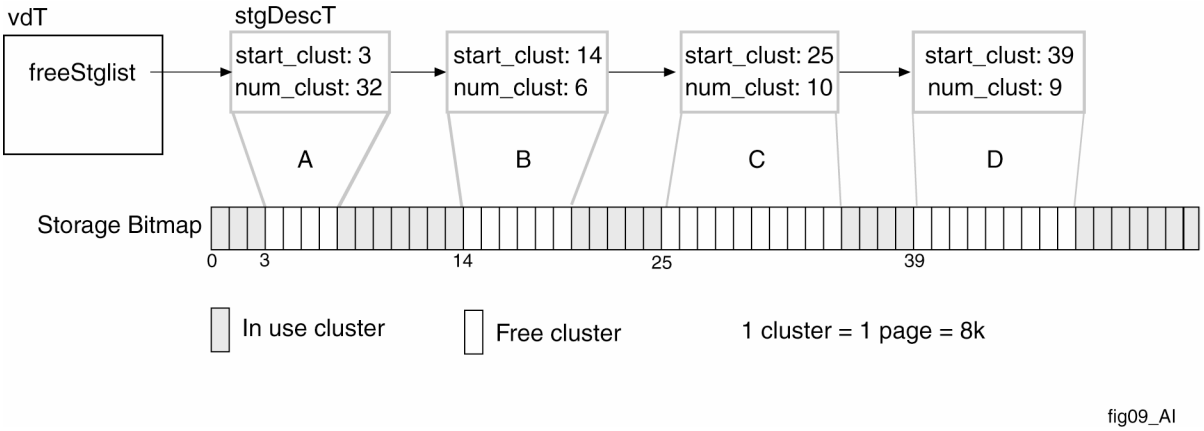


Figure 13: Relationship between the free-space cache and the SBM

2.5 Creating a domain

A domain is created using the `mkfdmn` command, and its invocation is:

```
mkfdmn [options] device domain
```

where `device` is the name of the storage device on which the domain will be built, and `domain` specifies the name of the new AdvFS domain to be created. The device can be a disk or an LSM volume. Only a user with root privileges may use this command. There are several options for this command that can be used to change the default size of the log, to use DVN 3 or 4 format, and to ignore overlapping partition warnings. See the `man` page for more information on these and other options that are available.

Be careful when using this command because it overwrites the specified storage and initializes it to an empty domain.

2.6 Accessing a previously-created domain

Sometimes it is useful to bring the disk(s) from one machine to another, and mount the domain on the second machine. Using `mkfdmn` on this disk will wipe out the information already on the disk. To use this domain on the second system, use the following steps:

1. Boot the system with the new disks in place. Use the list of disks in `/dev/rdsk` or the `hwmgr` command to determine the name of the new disks (the ones with the domain information to be mounted).

2. Go to the `/etc/fdmns` directory, and create a subdirectory with the name of the domain that is being moved onto this system. For example, if the domain name is `oracle_domain`, then create a subdirectory called `/etc/fdmns/oracle_domain`.

3. Change directory into the subdirectory created in step 2. Then create a symbolic link that points to the partition on which this domain information is located, and call it the partition name. For example, if the domain is on `/dev/disk/dsk12c`, use:

   ```
   ln –sf /dev/disk/dsk12c  dsk12c
   ```

   Although it has some limitations, the advscan utility can be used to rebuild the `/etc/fdmns` information from steps 2 and 3.

   ```
   /sbin/advfs/advscan –r <devices>
   ```

   This command will rebuild the `/etc/fdmns` entries automatically, giving the domain the name of the underlying disks (`/etc/fdmns/domain_dsk12c` for the above example). Feel free to rename the domain name in `/etc/fdmns/<domain name>` to one that is more meaningful to you. This is the only place that the domain name actually appears; it is not stored within the domain itself. One problem with using `advscan` is that occasionally it will recover several overlapping domains on partitions of a single disk, where there was really only one most-recent domain. For example, there have been cases where old domains were on the `g` and `h` partitions of a disk, and the `c` partition of the disk contained the most-recent domain, and the one that we wanted to be recovered. `Advscan` actually recovered all 3 domains into `/etc/fdmns`, requiring additional steps to remove the older, unwanted domains. Many times it is simplest to just build the `/etc/fdmns` entries manually, as was discussed in steps 2 and 3.

4. Up to this point, we have manually done the work that `mkfdmn` would have done, except that we have not destroyed the information on the underlying storage. Now we must get the name of the fileset(s) that are contained in this domain (if they are not already known). Use the `showfsets` command:

   ```
   showfsets –b <domain>
   ```

   This will display the names of the filesets contained in this domain.

5. Next, mount the fileset(s) from the domain onto the appropriate directories:

   ```
   mount <domain>#<fileset>  <mount point>
   ```

6. At this point, the information in this domain should be accessible on the new system.

2.7 Removing a domain

A domain is removed using the `rmfdmn` command, and its invocation is:

```
rmfdmn domain
```

where `domain` specifies the name of the existing AdvFS domain to be removed. Only a user with root privileges may use this command, and all filesets and clone filesets must be unmounted.

If you accidentally remove a domain, don't panic! If you have not reused the disks for another domain or raw device, you will be able to reestablish the original domain by doing the following:

1. rebuild the `/etc/fdmns/<domain name>` entry as described in section 2.6
2. fix the disk labels on the appropriate disk partitions. When `rmfdmn` removes a disk partitition from an AdvFS domain, it changes the disk partition's `fstype` field from `'AdvFS'` to `'unused'`. You must use the `disklabel` utility to reset this field back to `'AdvFS'`. Use the command:

   ```
   disklabel -s -F  <dskNp> AdvFS
   ```

   where `<dskNp>` is the disk number and partition (dsk12c for the example in section 2.6). Note that `'AdvFS'` in this command is case-sensitive; specifying `'advfs'` will not work.
3. At this point you should be able to mount your domain and access the data again.

2.8 Adding a volume to a domain

A domain can have up to 256 (BS_MAX_VDI) volumes, but since only one may be specified in the `mkfdmn` command, we need a way to add additional volumes to the domain. For this, we use the `addvol` command, which is invoked similarly to `mkfdmn`:

```
addvol [options] device domain
```

where `device` is the name of the storage device which will be added to the domain, and `domain` specifies the name of the existing AdvFS domain to be expanded. The device can be a disk or an LSM volume. Only a user with root privileges may use this command. See the `man` page for information on the options that are available. There is no need to quiesce the domain while the volume is added.

Internally, the work for this command is very straightforward, with most of the work happening in the kernel routine called `bs_vd_add_active()`. The main steps here are to determine the index in the domain's `vdpTbl[]`, to initialize the disk to look like an empty AdvFS volume with the requisite reserved files, to allocate and initialize all the in-memory structures for this volume, to open all the reserved files on this volume, to write the new volume attributes to disk, to mark the disk as mounted, and to add this volume's storage to the domain.

2.9 Removing a volume from a domain

Removing a volume requires the `rmvol` command, which is invoked similarly to `addvol`:

```
rmvol [options] device domain
```

where `device` is the name of the storage device which will be removed from the domain, and `domain` specifies the name of the existing AdvFS domain to have storage removed. The device can be a disk or an LSM volume. See the `man` page for information on the options that are available. There is no need to

quiesce the domain while the volume is removed, but all filesets in the domain must be mounted. Only a user with root privileges may use this command. Also, this command cannot be run while the `defragment`, `balance`, `rmfset`, or `rmvol` utilities are running on the same domain.

From an internal perspective, removing a volume is more interesting than adding a new volume, primarily because there may be files on the volume to be removed, and these must all be migrated onto other volumes in the domain. If there is not enough free space on other volumes in the domain to accept the offloaded files from the departing volume, the `rmvol` utility moves as many files as possible to free space on other volumes. Then a message is sent to the console indicating that there is not enough space to complete the procedure. The files that were not yet migrated remain on the original volume. You can interrupt the `rmvol` process without damaging your domain. AdvFS will stop removing files from the volume; files already migrated from the volume will remain in their new location.

The volume is actually removed by a series of calls from `rmvol.c` into the kernel. After verifying the volume and domain parameters, the command calls `advfs_remove_vol_from_svc_class()` which dispatches into the kernel to remove this volume from the service class table. There is only one service class implemented in AdvFS, so removing this volume essentially means that no new files can be created on this device once it is removed from the service class. Next `rmvol.c` calls a routine called `move_metadata()` which migrates all user-defined files off the volume being removed. After this, `rmvol.c` calls `advfs_remove_volume()` which eventually dispatches into a kernel routine called `bs_vd_remove_active()`. This routine is responsible for migrating the log and fileset tag file off this volume (if they are on it), and then removing the volume from the domain.

2.10 Domain activation and deactivation

Domain activation is a process of setting up the in-memory data structures for a given named domain so that files, filesets, or volumes inside it can be accessed. Although a domain can be activated for the duration of a single kernel request, domains are typically activated when the first fileset in the domain is mounted, and remains activated while its filesets remain mounted. The activation process is typically handled by two routines that are called sequentially: `bs_bfdmn_tbl_activate()` and then `bs_domain_access()`.

`bs_bfdmn_tbl_activate()` is invoked with a domain name, and uses the global `DmnTblLock` to synchronize access to global domain data such as the chain of in-memory `domainT` structures (see Section 2.4.1). It first calls `domain_name_lookup()` to see if the domain is already on the list of activated domains. If the domain is already activated, then the domain ID is simply returned. Otherwise, `bs_bfdmn_tbl_activate()` must go through the work of activating the domain from data on disk. It does this by first calling `get_domain_disks()` which finds the disks that are associated with this domain by finding the list of volumes in the `/etc/fdmns/<domain>` directory. `bs_bfdmn_tbl_activate()` then loops through each of the volumes, getting the volume attributes from disk, verifying that the information from each volume synchronizes with the others, allocating and initializing the in-memory `vdT` structures, adding the volumes to the service class, and verifying that we found all the disks for this domain. The first time a valid volume is encountered in this loop, a `domainT` structure is allocated, added to the chain of domain structures, and entered into the domain hash table (`DomainHashTbl`). After all the volumes are verified and brought into memory, `bs_bfdmn_activate()` is called. This routine does domain-specific actions including opening the log file, performing recovery on the domain, opening the root tag directory, marking the virtual disk and domain attribute records indicating that this doman has been activated, setting the in-memory domain structure to be BFD_ACTIVATED, and scanning the bitfileset delete-pending list to finish removing any filesets which are on that list.. At this point the work done by `bs_bfdmn_tbl_activate()` is complete and a domain ID is returned to the calling routine.

The domain ID is then passed to `bs_domain_access()` which calls `domain_lookup()` to retrieve a pointer to the in-memory `domainT` structure from the domain hash lookup table. This table contains all the activated domains hashed by domain ID. This `domainT` pointer is passed back to the caller which can now use the `domainT` and `vdT` structures until the domain is deactivated. Each activator increments the `domainT.activateCnt` so these structures will be kept in memory until all threads accessing the domain have exited.

Domain deactivation is handled by calling `bs_domain_close()` and `bs_bfdmn_deactivate()`. The latter routine decrements the `domainT.activateCnt`. If this is still greater than zero, there are other threads that require this domain, so its work is done. Otherwise it goes through the work of deactivating the domain. These steps include checkpointing the log, closing the root tag file, clearing the SBM cache for each of the volumes, closing the log, and dismounting all volumes and removing their in-memory structures. Finally this routine calls `dmn_dealloc()` which will remove the `domainT` structure from the `DmnSentinelP` chain, remove the domain from the domain hash table, remove the root fileset, and remove all memory associated with the `domainT` structure.

Chapter 3: Fileset Concepts

3.1 Basic Concepts

Filesets are defined to be a collection of related files. The relationship between the files is defined by the users and/or applications. The files are organized in a file/directory hierarchy which is functionally equivalent to what is generally called a UNIX file system. Filesets can be mounted and unmounted like file systems, and they are accessed like file systems. The main difference is that there is no direct or simple mapping between a fileset and its storage as illustrated in the figure in section 3.2.1. A fileset's files are stored anywhere in its domain, interspersed with files which belong to other filesets. In short, a fileset is an abstraction that is added on top of a storage domain that allows specific subsets of files to be addressed via the UNIX naming conventions, as well as being backed-up and restored as unique subsets.

One complication in this view is that there are special files, called **reserved files**, that are generally not visible to the user and that do not belong to any of the filesets that the user may create. These files (RBMT, BMT, SBM, LOG, MISC, root tag file, and fileset tag files), may have domain-wide scope (root tag file and LOG), fileset scope (fileset tag files), or volume scope (RBMT, BMT, SBM, and MISC), making the filesets that the user creates inappropriate for the abstraction of these reserved files. Therefore, AdvFS creates a special fileset, called the **root fileset** (or **hidden fileset**) for each domain, and all the reserved files are considered to reside in this special fileset. How to differentiate between these various filesets will be discussed later in this chapter.

3.2 On-disk structures

3.2.1 Fileset Tag File

Each AdvFS file is identified by a **tag** number, similar to an inode number. This tag number is really just an index into a slot in the **Fileset Tag File**. This file was traditionally called a Tag Directory, but since this name is confused with normal directory files, that naming convention is being discouraged. Tag numbers are discussed further in Section 4.3. The figure below depicts a domain with 2 filesets and 5 files. Each fileset can be independently mounted and unmounted. Note that there are two tag files for this domain, and that the tag numbers for the files in each will run sequentially from 1 to n. This means that there may be two files with any given tag number in this domain. Therefore, the unique identifier for any given file within a domain is the combination of its tag number and its fileset tag number. Where does the fileset tag number come from? As explained in Section 2.3.5, there is a **Root Tag File** in each domain that contains the tags for each Fileset Tag File. The index into the root tag file for a particular fileset is that fileset's tag number.

A good way to view the various tag files in AdvFS is to think of the Root Tag File (section 2.3.5) as the directory of all filesets in a domain and the Fileset Tag File as the directory of all files in a fileset. The root tag file is only used when accessing a fileset for the first time. Once a fileset is open, AdvFS uses a pointer to the bfSetT structure for most file operations.

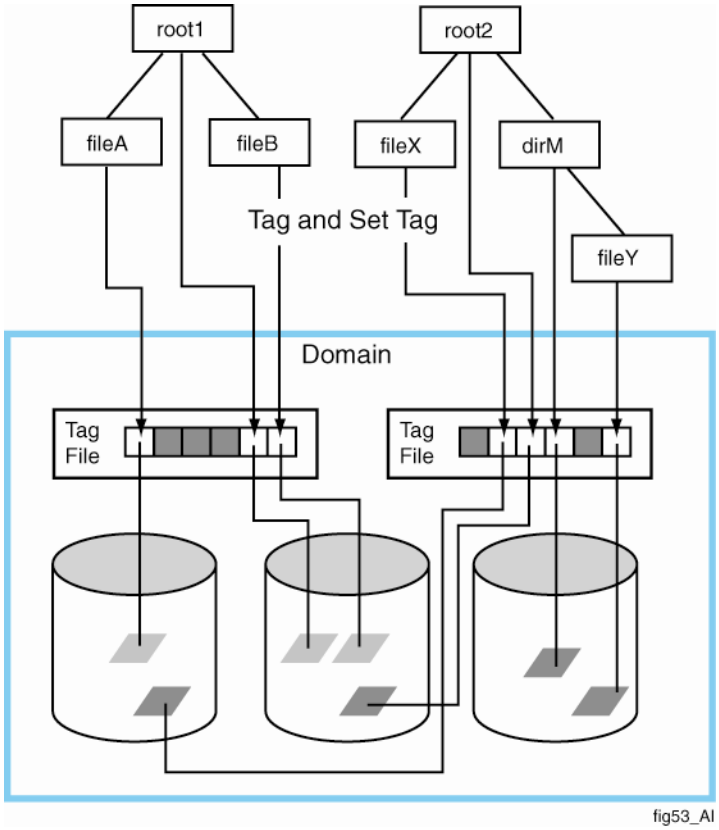


Figure 14: Relationship of files to on-disk data via tags and set tags

The tag file consists of a series of 8K pages (each of type `bsTDirPg`); each page contains a header (bsTDirPgHdr) plus 1022 **tagmap entries.** If these entries are viewed as a linear array that spans the pages, the index into this array is the actual tag number that will appear in the entry. Every tagmap entry is comprised of an 8-byte bsTMap structure that contains the sequence number, volume index, and primary mcell ID for the file. The figure below gives a simplified view of the traversal of information from a tag number to the file on disk.

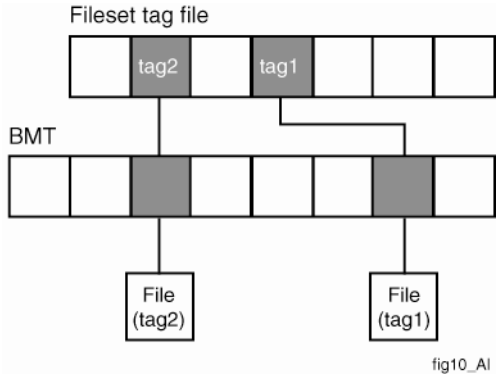


Figure 15: Fileset tag file to on-disk file

3.2.2 Fragment File

Storage allocation in AdvFS is done in 8 KB page units. For files that are not an even multiple of pages in length, only a part of the last page is used to store data. For file systems with many small files, this can result in inefficient storage utilization. To minimize wasting disk storage, AdvFS stores the data that is beyond the last full page of a file in a fragment. This fragment is stored in a special file called the frag file. There is one frag file per fileset, and it always has a tag of 1. The use of fragments and the frag file is explained in more detail in Section 6.2.

3.3 In-Memory structures

3.3.1 bfSet Structure

Filesets are AdvFS's equivalent of UFS's file system structure. The `bfSet` structure is the in-memory representation of a fileset. A `bfSet` can be reached via the `fileSetNode` structure, a file's `bfAccessT`, or by following the doubly-linked list of filesets within the domain structure (`domainT.bfSetHead`). Important information in this structure includes:

- Pointer to the fileset's domain
- Linked list of other `bfSet`s within the domain
- Linked list of access structures associated with this fileset
- Clone and original fileset state and pointers
- Fileset Tag File information
- Fragment file information
- Pointer to the `fileSetNode` structure

A magic number is also contained in the structure, and is used for structure validation and debugging (`bfSetMagic = SETMAGIC = 0xADF00002` for a valid `bfSet` structure).

The global `BfSetHashTbl` structure indirectly points to `bfSet` structures. The macro `BFSET_GET_HASH_KEY(bfSetId)` is used to generate a hash key value for accessing the `BfSetHashTbl`, and is a fast way to get a `bfSet` for a fileset. All associated hash macros can be found in `bs_bitfile_sets.h`

The figure below shows the relationship between the domain, fileset, and tag file structures for a domain where there are two filesets, 'usr' and 'var'. Note that two `bfSetT` structures that describe the mounted filesets have `fileSetNode` pointers and names, while the 'hidden' root fileset has neither. Also note that the tag numbers for the two tag files are ordered sequentially in the context of the root (or hidden) fileset. This is because the tag files are actually members of the root fileset, as evidenced by the facts that their `bfAccess.bfSetp` fields point to the root fileset, the root tag file and the tag files are linked via `bfAccessT.setFwd`, and the file count for the root fileset is 2 (the two tag files). Being in the context of the root fileset is not unique for the tag files, since all metadata files in a particular domain are in the context of the root fileset. One other thing to notice in this figure is that `the bfSetT.dirTag` is -2.0 for the root fileset. This is an arbitrary value assigned to the root fileset when the domain is established; there are no access structures with a tag value of -2. The lowest absolute value for metadata tags is -6, and 1 for normal tags.

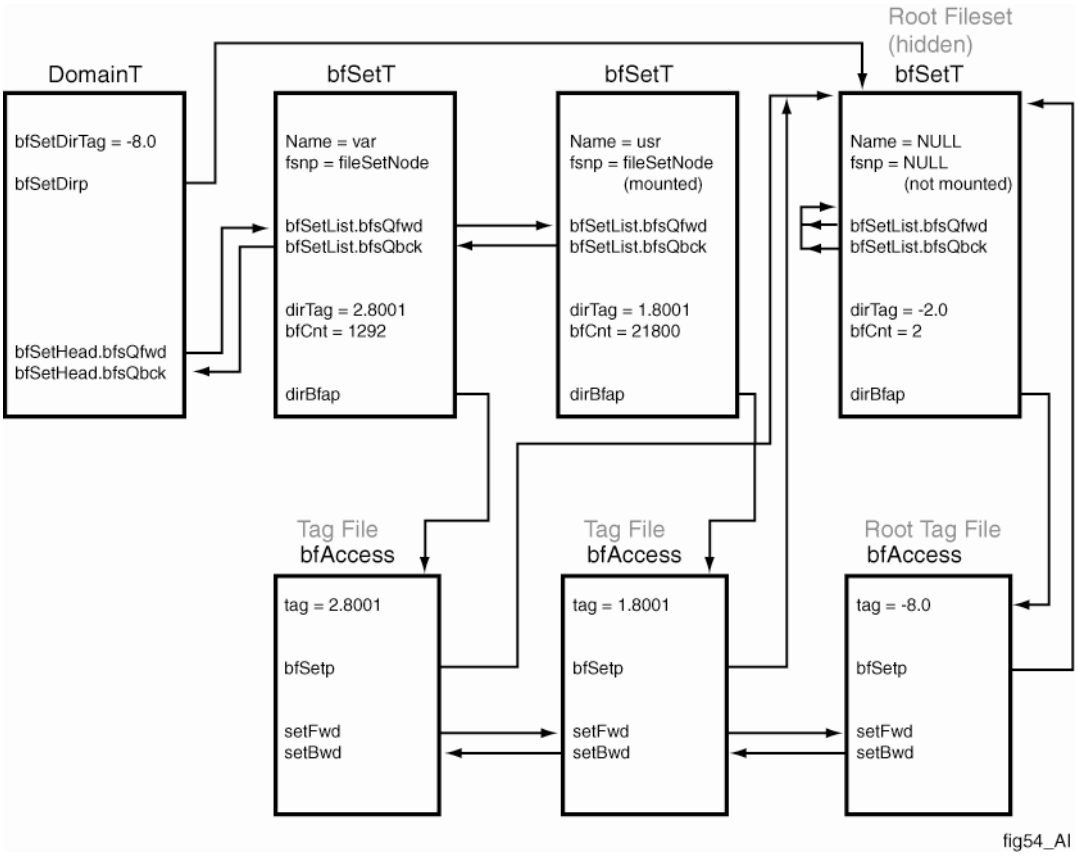


Figure 16: Relationship among in-memory domain, fileset, and tag file structures

In the v4.0 pools, the `bfSet` structures were not chained off the domain structures. All bfSetT structures for all domains on a system were kept in a global vector called `BfSetTbl`[]. The position of any given `bfSetT` in this vector was determined by the value of (fileset handle -1). Thus, if the fileset handle was 12, the `bfSet` address was located at `BfSetTbl`[11]. The field `domainT.bfSetDirH` contained the handle for the root fileset for the domain.

There is one peculiar aspect of the linked list of `bfSet` structures in each `domainT` that bears mention. A typical way to implement this would have been to add two `domainT` fields such as `bfsFwd` and `bfsBwd` which would have been initialized to point back to the `domainT` structure for an empty list. As each fileset was inserted onto the list, the forward pointer would have pointed to the `domainT` structure, and the chain's forward pointer would have pointed to the new `bfSet` structure. This is not how this was implemented, however. A new field called `domainT.bfSetHead` was added to the domain structure. This is a `bfsQueueT` structure that contains `bfsQFwd` and a `bfsQbck` fields. The tricky part is that each of the `bfsQueueT` fields are initialized to point to `bfSetHead` instead of the domain or fileset structures. This necessitates the use of some macros to walk, insert, and delete `bfSet` structures on the domain. When debugging, the key to remember is that: 1) `domainT.bfSetHead.bfsQfwd` points to `bfSetT.bfSetList`, not `bfSetT`, and 2) the end of the chain of `bfSet` structures is reached when `bfSetT.bfSetList.bfsQfwd` points to (`domainT` + `offsetof`(`domainT.bfSetHead`)), not to `domainT`.

3.3.2 fileSetNode Structure

This structure contains information about the mounted AdvFS fileset. The `fileSetNode` can be reached from the `fsContext` structure for each file within the fileset or from the `m_info` field in the `mount` structure for the fileset. Important information in this structure includes:

- Doubly-linked list of `fileSetNodes` for all other mounted filesets within the domain
- Tag structures for the root directory and the ".tags" for the fileset
- Pointer to the fileset's domain
- Pointers to the access structure and vnode for the root directory of the fileset
- Fileset ID and pointer to the `bfSet` structure associated with the fileset
- Pointer to the `mount` structure associated with the fileset
- Fileset quota information
- `fileSetStats` structure containing statistics on fileset use and operations

The tags for the root directory (`rootTag` field) and ".tags" (`tagsTag` field) are 2 and 3 respectively (see section 4.3 for more on special tags). A magic number is also contained in the structure, and is used for structure validation and debugging (`filesetMagic = FSMAGIC = 0xADF00005` for a valid `fileSetNode` structure).

3.4 Creating a fileset

A fileset is created using the `mkfset` command, and its invocation is:

```
mkfset [-o frag | nofrag] <domain> <fileset name>
```

where `domain` specifies the name of an existing AdvFS domain, and `<fileset name>`. is the name of the fileset to be created. The optional `frag/nofrag` argument allows the explicit enabling or disabling the creation of frags for files in this fileset (see Section 6.2). Only a user with root privileges may use this command.

Each domain may have multiple filesets (there is no explicit limit) unless the fileset is enabled for DMAPI, in which case only one fileset per domain is allowed (see Chapter 11). Each fileset in a domain can be independently mounted and unmounted, and have unique fileset quotas assigned.

Once created, a fileset may be renamed using the `renamefset` command:

```
renamefset domain <original fileset name> <new fileset name>
```

This command requires root privileges, and also requires that this fileset (and any clones) be unmounted before it can be renamed.

3.5 Displaying and changing fileset attributes

The attributes of a fileset can be displayed using the `showfsets` command, as follows:

```
showfsets [arguments] domain [fileset]
```

where domain specifies an existing domain, and fileset specifies one or more filesets in that domain. This command has several options that can be specified to limit or customize the output. See the man page for more information.

The attributes of a fileset can be modified using the chfsets command, as follows:

```
chfsets [arguments] domain [fileset]
```

where domain specifies an existing domain, and fileset specifies one or more filesets in that domain. The user must have root privileges to run this command.

This command, if used without options, executes the showfsets command, and displays the existing attributes of the fileset. By specifying options, this command can be used to change file and block quotas and enable or disable object safety, fragging, or DMAPI for the fileset.

See the man page for these commands for details about the use of the arguments.

3.6 Removing a fileset

An existing fileset can be removed by using the following command:

```
rmfset -f domain fileset
```

where domain is the existing AdvFS domain and fileset is the fileset to be removed from that domain. The -f option turns off the message prompt and is useful when running this command from a script. There are several restrictions on the successful completion of this command: the fileset must not be mounted, the user must have root privileges, and the fileset must not have a clone. If there is a clone fileset, remove the clone before removing the original. The clone is removed using the same rmfset command, just specify the name of the clone fileset.

Note that all files in this fileset will be removed when this command is executed, so be careful.

3.7 Cloning a fileset

The clonefset command allows the creation of an on-line backup of the files in a given fileset by making a read-only copy (clone) of the specified fileset. The command is:

```
clonefset domain fileset clone
```

Where domain is the existing AdvFS domain, fileset is the fileset containing the files to be cloned, and clone is the name of the cloned fileset. The user of this command must have root privileges, and there cannot be a DMAPI-enabled fileset in this domain. Only one clone per fileset can be maintained at a time. Files in the clone fileset may be read, but cannot be modified.

Cloning is explained in detail in Section 5.1. A key point to remember about cloned filesets is that the files in the cloned set have extent map information in the original fileset until all the pages of the original are cowed to the clone. Therefore, original filesets cannot be removed until the cloned fileset has first been removed.

3.8 Removing a cloned fileset

Removing a cloned fileset is functionally the same as removing an original fileset, since they are both done using rmfset. There are several differences in internal handling that should be noted. First, an original fileset cannot be deleted while it has a clone fileset. This is because the cloned files may have partial extent maps, and depend on the extent maps in the original files to locate non-cowed pages.

Second, removal of a cloned fileset may actually result in the removal of files within the original fileset. Consider the example of deleting a file in a fileset that has a clone. When the file is removed, it is not actually deleted at that time. Its entry in the directory is removed so that it can't be found, but the underlying bitfile is not deleted because its extent maps may still be needed for accessing data for the clone. In this case, the original file's `bfAccess->deleteWithClone` is set to 1. This saves time when the file is deleted since the pages do not have to be COWed at that time. Eventually, this bitfile is deleted when the clone fileset is removed, as alluded to earlier. The `delete_clone_set_tags()` routine walks through all files in the clone fileset freeing the storage for the cloned files. It also checks to see if the original file's `bfAccess->deleteWithClone` is set; if so, the original file is also deleted.

3.9 Mounting and unmounting a fileset

In AdvFS, filesets are mounted and unmounted to gain access to the domain's storage. Many filesets per domain can be mounted simultaneously. The command for mounting an AdvFS fileset is

```
mount [options] domain#fileset <mount point>
```

where `domain` and `fileset` are the AdvFS domain and fileset to be mounted, and `<mount point>` is the location in the existing directory tree on which this fileset will be mounted. There are many options to the mount command, and they can be viewed in the `man` page. Several AdvFS-specific options are worth mentioning here. The `-o adl` option causes all files in the fileset to use atomic-write data logging for the duration of the mount. See Section 8.2 for information on atomic-write data logging. The `-o dual` option enables the AdvFS fileset to be mounted as a domain volume even though it has the same AdvFS domain ID as a fileset that is already mounted. This is typically used when splitting a mirrored volume, and remounting one of the mirrors for backup. See Section 14.3 for an explanation of mirrored volumes. The `-o noatimes` option modifies the normal updating of access times for files on the fileset. The default behavior (`atimes`) requires AdvFS to flush file access times to disk for each read of regular files. This adds to the I/O load on the system, and some applications do not require a strict time accounting for file accesses. If approximate times of last access are acceptable and I/O performance is critical on a particular fileset, it can be mounted with the `-o noatimes` option. In this case, the file access time updates are not flushed to disk until other file modifications occur. This behavior does not comply with industry standards, but can be used to reduce the number of disk writes for applications with no dependencies on file access times.

The last option to mention is `-o directIO`. This option is used to force all files in the fileset to be opened in direct I/O mode. This option is not supported in the field and does not appear in the `man` page. This option is only used internally for direct I/O testing, and is not to be discussed outside the company. To use this option, a variable called `Advfs_enable_dio_mount` must be set to a non-zero value using a debugger. Then the fileset can be mounted using the `-o directIO` option. To see what mode a fileset is mounted in, use the `'mount -l'` command; if it is mounted in `directIO` mode, the value `'directio'` will appear in the values listed. Since direct I/O and memory-mapping are not supported concurrently, and since all executable files are memory mapped by the loader, any files on a fileset mounted for direct I/O use will not be able to be executed. This is one of the reasons this option is not supported externally. Perhaps the primary reason that this option is not supported is that the Clustered File System (CFS) does not do any locking to prevent simultaneous writes to the same region of a file by different threads (although AdvFS on a standalone system does). This means that unintelligent applications using direct I/O via CFS could end up with mixed data which could be interpreted as data corruption. Since direct I/O was designed for intelligent applications such as Oracle and the other major databases which do their own write synchronization among threads, this is not a big limitation. Not advertising this option prevents an onslaught of customers complaining about data corruption for applications that do not synchronize their own threads.

Section 14.1 discusses how AdvFS and the VFS layers cooperate to mount filesets. That section also shows how to trace the `mount` and `vnode` structures for mounted filesets into the AdvFS `fileSetNode` and `fsContext` structures.

Filesets are unmounted using the `umount` command:

```
umount [options] domain#fileset | <mount point>
```

where `domain#fileset` refers to a mounted fileset within the domain, or <`mount point`> indicates a location in the directory tree where an AdvFS fileset is already mounted. The options are explained in the `man` pages.

Chapter 4: File Concepts

4.1 Files vs Bitfiles

In AdvFS, the term **file** refers to a named storage entity that consists of attributes and a single byte stream as is defined by POSIX; basically it is a UNIX file. A **bitfile** is a generic storage entity that is named with a unique identifier (a tag in AdvFS) and consists of attributes and an array of 8KB pages. Basically, the difference between a bitfile and a file is the level on which you are viewing the file: the physical storage layer manipulates bitfiles on a low level (through mcells, tags, and extents) and the file hierarchy layer manages files on a higher level (through filenames, paths, and directories).

There is a one-to-one relationship between files and bitfiles in AdvFS. Files are instantiated through bitfiles, which means that a file is a bitfile plus some POSIX attributes and POSIX semantics. In fact, everything (file, directory, metadata, tag directory, transaction log, and storage bitmap) in AdvFS is instantiated through a bitfile.

IMPORTANT NOTE: To reduce confusion and maintain word consistency, the word “file” will be used for both “file” and “bitfile” throughout this document. However, where specific differentiation is necessary, “bitfile” will be used.

4.2 Unique File Identification – tags

AdvFS uses tags to decouple a file name from its on-disk structure. Each file within a fileset has its own unique tag, but a file's tag is not unique within a domain. A file's unique identifier in a domain consists of the file's tag and the tag of the fileset’s tag file (see section 2.3.5 Root Tag File). The following figure shows the relationship between the Root Tag File, the Fileset Tag File (see section 3.2.1), and a file:

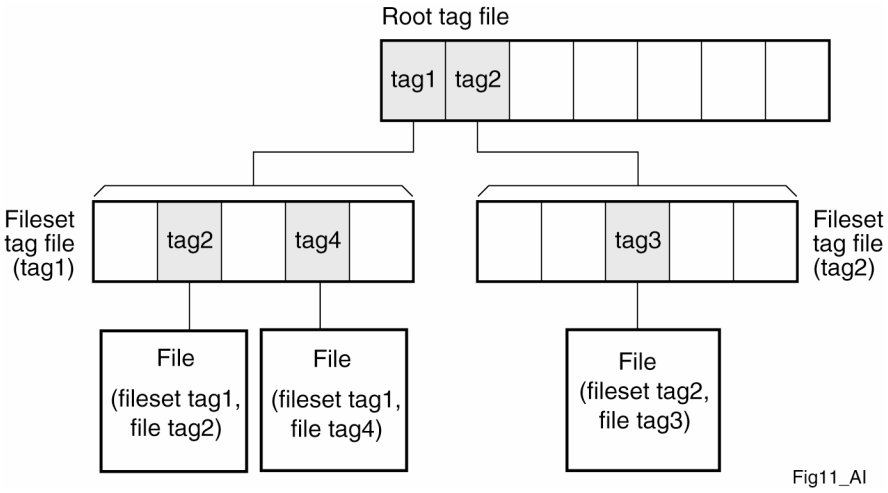


Figure 17: Root tag file to fileset tag file to file

The BMT has been left out to simplify the figure; but as is always the case, a Tag File entry always points to a BMT mcell, which actually defines the file through the file attributes record and one or more extent map records.

A file tag consists of a tag number and a sequence number. The tag number is really just an index into a slot in the fileset tag directory. Whenever a file is deleted, its tag becomes available for reuse. However, for consistency (as for crash recovery), AdvFS cannot reuse the tag unless it is changed from previous uses of that tag. Therefore, each time a tag is reused, its sequence number is incremented to differentiate it from the previous use of the same tag. The sequence number can be used only 32,767 times and then it becomes a dead tag, never to be reused again. This is because the tag sequence is a 16-bit unsigned value on disk, and the high-order bit (0x8000) is used to mark that the slot (tag) is in use. If the sequence number is ever incremented past the value `BS_TD_DEAD_SLOT` (0x7fff), then this slot is dead.

Each fileset has five special tags that are assigned on fileset creation:

File	Tag
Fragment Bitfile	1
Root Directory (/)	2
.tags Directory	3
User Quota File	4
Group Quota File	5

The `/sbin/advfs/tag2name` utility can be used to translate a tag number to a file's full pathname.

AdvFS reserved files (RBMT, BMT, SBM, Tag File, LOG, and MISC) also have special tags. These tags differ from normal tags in that they are negative and do not correspond to slots in a Tag File. In addition, since the reserved files can appear on any volume in a domain, there are different tags for each reserved file depending on which volume they are on. The following explains the calculations used to determine the tag number for each reserved file.

The primary mcell is the initial mcell allocated for each bitfile. Since the reserved files are set up when a domain is created (or when a volume is added to the domain), their primary mcells are known values. Each tag number is calculated using the primary mcell value and the index of the volume on which the file resides. This is shown in the following formula:

Tag = - (primary mcell number + (volume index * 6))

The following table shows the relevant values and the resulting tags for the reserved bitfiles on the first three volumes in a domain:

Reserved File	Primary Mcell	Reserved File Tag Number		
		Volume 1	Volume 2	Volume 3
RBMT	0	-6	-12	-18
SBM	1	-7	-13	-19
Root Tag	2	-8	-14	-20
LOG	3	-9	-15	-21
BMT	4	-10	-16	-22
MISC	5	-11	-17	-23

Remember that there will only be one LOG file and one Root Tag File per domain, but the other reserved files reside on each volume in the domain.

AdvFS reserved files (RBMT, BMT, SBM, Tag File, LOG, and MISC) also belong to a special fileset, the 'hidden fileset' that is associated with each domain (Section 3.3.1). Thus, the value of `bfap->bfSetp` for each of these reserved files will be the value of the hidden fileset. Normally this is not of great significance, except for several issues. First, the hidden fileset cannot be cloned, and therefore these reserved files will never be cloned (see Chapter 5). Also, because the `frag` file has a tag of 1, and the first tag directory in a fileset has a tag of 1, the only way to know which of these files a given access structure represents is to look at its `bfap->bfSetp` value. The frag file is associated with a real fileset, and the tag directory is associated with the hidden fileset. The hidden fileset always has a `bfSetT.dirTag.num` value of 0xfffffe (-2 or `-BFM_BFSDIR`), and can be displayed for each domain using the `domain -fh` command in the `crash` utility..

4.3 In-Memory Structures

Files are represented by VFS `vnodes,` and bitfiles are represented by AdvFS file context (`fsContext`) and access (`bfAccess)` structures. Vnodes contain generic VFS information to describe a file, plus a file system-specific area called `v_data`. There is typically one `vnode` per open file, but the `vnode` may persist on a free list for some time after the file has been closed. AdvFS uses the `vnode.v_data` field to hold links to the `bfAccess` and `fsContext` structures. The following figure shows the relationships among these structures as well as the other volume, domain, and fileset structures discussed in Chapters 2 and 3.

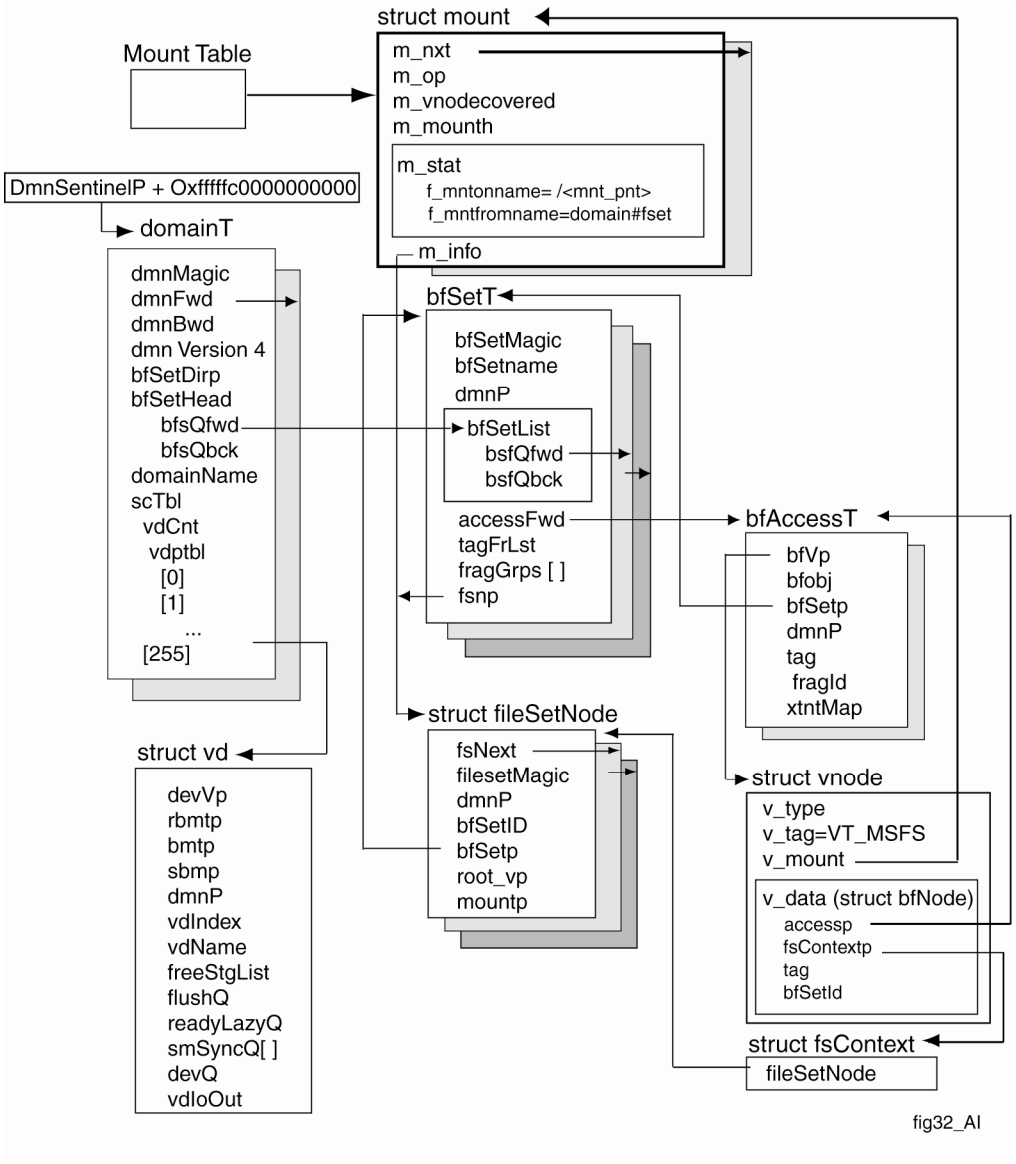


Figure 18: Relationships between file, volume, domain, and fileset in-memory structures

4.3.1 bfAccess structure

Files are managed by AdvFS with access structures, the memory-resident structures that hold most of the information needed to operate on a file. The `bfAccess` structure contains bitfile attributes, the extent map(s), a dirty buffer list, and pointers to the domain and bitfile set descriptors. It also contains pointers to the file's vnode and to the VM object used to interact with UBC. The relationship between a file and its VM object is described in more detail in Chapter 7.

There is one `bfAccess` structure per open AdvFS bitfile. When a file is first opened, an access structure is taken from the free list. When a file is closed, the access structure remains cached and is added to the free list of access structures (global variable `FreeAcc`). Until an access structure is recycled, it retains all the information about a file and can be reassociated with the same file if it is opened again.

Recycling of access structures is done asynchronously as new access structures are needed. The internal algorithms tend to recycle access structures on a least-recently-used basis by removing access structures from the head of the free list, and adding newly-closed access structures to the tail of the list. One exception to this is that `bfAccess` structures marked ACC_INVALID are added to the front of the free list since they have no consistent information for any given file. There is another list on which `bfAccess` structures can be cached when they do not belong to an open file, and this is the closed list, `ClosedAcc`. This list is a holding area for access structures that are not yet capable of being recycled to a new file, but are no longer associated with an open file. When an access structure's reference count goes to zero, the routine *free_acc_struct()* decides whether the structure can go directly onto the free list, or must spend some time on the closed list. Typical reasons for adding the structure to the closed list are that it has dirty pages that must be flushed to disk, or it has dirty stats that must be flushed in a root transaction. In some cases an access structure can be moved directly to the free list when the file is closed.

Access structures on the closed list are processed and moved to the free list by the *fs_cleanup_thread()*. This is a kernel thread that runs in the background and is responsible for several asynchronous activities: 1) cleaning up access structures on the closed list so they can be moved to the free list, 2) deallocating access structures, 3) completing directory truncation operations, and 4) marking frag group headers as bad if requested to do so. This thread is event driven, and receives its events from the message queue called the `CleanupMsgQH`. The *get_free_acc()* routine sends a message to this thread to process access structures on the closed list whenever the maximum number of access structures have been allocated, the free list is empty, and there are access structures available on the closed list. It can also request cleanup if the number of access structures with `saved_stats` is at least `MaxSavedStatsAccessPercent` (15%) of the total number of structures allocated.

Management of access structures can be critical on a system. Because they are fairly large (approx 960 bytes) and a system can have many open files, the amount of memory they consume must be carefully managed. In addition, as the filesystem load on a system changes, it is helpful to be able to allow the access structure pool to change size as needed. When AdvFS is first enabled on a system, a pool of access structures is allocated and placed onto the free list. The number allocated is (2 * `AdvfsMinFreeAccess`), which defaults to (2 * MIN_FREE_VNODES) or 300. This allows AdvFS to coordinate its number of file-specific structures with those allocated for VFS. The *get_free_acc()* routine may send a message to the `bs_access_alloc_thread()` thread to allocate more access structures if there are less than `AdvfsMinFreeAccess` structures on the free list and there are fewer access structures than allowed (`MaxAccess`). The value of `MaxAccess` is scaled to the amount of memory on the system, and is set so that the amount of memory consumed by the access structures is limited to `AdvfsAccessMaxPercent` (defaults 25%) of the amount of memory on the system. Access structures can also be deallocated when they are no longer needed. This is detected in the ADD_ACC_FREELIST() macro that is called to move an access structure onto the free list. If it detects that there are more than `MaxAccess` structures allocated, or that more than `AdvfsAccessMaxPercent` of the entire access structure pool is on the free list, then it will send a message to the cleanup thread to deallocate an access structure. The access structure pool will always contain at least (2 * `AdvfsMinFreeAccess`) structures.

The only access control variable that can be tuned via `sysconfig` (as of Tru64 UNIX Version 5.1B) is `AdvfsAccessMaxPercent`, which can be adjusted from 5% to 95%.

4.3.2 BfNode

The v_data region of the vnode is used to store filesystem-specific data. The `bfNode` is a structure that links the VFS file structures with the AdvFS file structures, and resides in the `vnode.v_data` field.

The bfNode is a container for the file tag, bitfile set ID, and pointers to the fsContext and bfAccess structures.

4.3.3 fsContext structure

The context structure contains file attributes (also called stats) and disk quota information. It also contains an important complex lock, the file_lock. This lock is used in many paths to synchronize access to the file, particularly when the file size is changing. The fsContext structure immediately follows the bfNode structure in the vnode.v_data field. The possibility exists (although it does not as of Tru64 5.1B) that the fsContext structure will be unable to fit within the malloced size of the vnode structure. In this case, memory for the fsContext is obtained from the malloc pool, and the structure can be reached using the pointer in the bfNode.

4.3.4 bsBuf

Each file is logically comprised of a byte-stream of data. AdvFS organizes this byte-stream into a series of 8k pages. When a given page is in the buffer cache, the bsBuf structure is used to represent that page. This means that each file may have many bsBuf structures associated with it. Each bsBuf contains information about the state of the page (dirty, busy, etc.), the location of the UBC page that holds the data for this page, which file and page are represented by this data, and where this data resides on disk. See Chapter 7 for details about the buffer cache and uses of the bsBuf structure.

4.4 On-Disk Structures

On disk, a bitfile consists of metadata (data about the file) and the file's data. As described in section 2.3.2, the metadata consists of a series of mcells in the BMT. An mcell and all the records in the mcell are associated with one bitfile. The each mcell stores the fileset tag number and the file tag number of the bitfile corresponding to that metadata. Each mcell has a pointer to a "next" mcell. This chain of mcells all belong to the same bitfile. This chain of mcells contains all the non-extent metadata for the bitfile.

The primary mcell also contains a primary extent map record. File storage is defined by **extents**, which are contiguous areas of disk storage. This primary extent map record may contain the first extent of data of the file. This record also has a "chain" pointer to a secondary extent record. That secondary extent record will use its "next" pointer to form a linked list of extent records as the file grows and must map more extents.

So the metadata for each bitfile is stored in mcells that may form two singly linked lists of mcells, both headed by the primary mcell. One linked list forms off the "next" pointer in the primary mcell. This list links mcells together by using the "next" pointer in each mcell. This list contains all the metadata for a bitfile except the extents of the file. The other linked list consists of extent mcells. This list starts from the BSR_XTNTS record that always exists in the primary mcell of a bitfile. The BSR_XTNTS record contains a chain pointer to the next extent mcell. Subsequent extent mcells are pointed to by the "next" field in the mcell.

Extent records only describe storage on the disk that they live on and every page in a file is described by an extent except pages in a **hole** at the end of a file. A hole is a portion of a file that does not have storage allocated to it. This can happen if a seek is done on a file which is then followed by a file write at the new location. Holes are described by extent descriptors that do not point to any block on the volume.

No page in a file falls between two extent descriptors or between two mcells. The extent descriptors start by describing page 0 of the bitfile and each extent mcell starts describing storage at the page where the previous extent mcell stopped. See section 4.4 for more information about extent maps and descriptors.

4.4.1 Extent mcell record combinations

The secondary extent records fill the entire mcell so no other records can fit in the extent mcells. There are 3 types of extent records. The primary extent record is always in the primary mcell and is a BSR_XTNTS record. There are two types of secondary extent records: BSR_SHADOW_XTNTS and BSR_XTRA_XTNTS. The BSR_SHADOW_XTNTS record is used by DVN 3 domains as the first secondary extent record and it is used in DVN 3 and 4 domains by striped files as the first extent record in a stripe. The BSR_XTRA_XTNTS record is used for all other secondary extent records on either domain version. When a BSR_SHADOW_XTNTS record exists, there is no extent stored in the primary BSR_XTNTS record. Conversely, if there is an extent in the BSR_XTNTS record it will not point to a BSR_SHADOW_XTNTS record. However, the absence of an extent in the BSR_XTNTS record does not require it to point to a BSR_SHADOW_XTNTS record.

Below are the allowed extent mcell record chains:

1. DVN 3 and 4 reserved bitfiles:

   ```
   BSR_XTNTS -> BSR_XTRA_XTNTS -> BSR_XTRA_XTNTS -> …
   ```

   Most of the reserved files don't need and don't have any BSR_XTRA_XTNTS mcells. The BMT is the exception. It almost always has one or more BSR_XTRA_XTNTS mcells. Remember that all reserved file mcells are in the RBMT in DVN 4 and in BMT page 0 in DVN 3. Reserved files never cross volume boundaries. All the mcells and all the data for a given reserved file will be on one volume.

2. DVN 3 non-reserved non-striped files.

   ```
   BSR_XTNTS -> BSR_SHADOW_XTNTS -> BSR_XTRA_XTNTS -> BSR_XTRA_XTNTS -> …
   ```

   In DVN 3, no extent information was kept in the primary extent record. The primary extent record used its "chain" pointer to point to a BSR_SHADOW_XTNTS record. This record contained the first extent descriptor. The BSR_SHADOW_XTNTS record contains a field that is a count of the BSR_SHADOW_XTNTS mcells and all the BSR_XTRA_XTNTS mcells. This field is short and may roll over for very long lists of extent mcells.

3. DVN 4 non-reserved non-striped files.

   ```
   BSR_XTNTS -> BSR_XTRA_XTNTS -> BSR_XTRA_XTNTS -> …
   ```

   In DVN 4, the BSR_XTNTS record can contain the file's first extent. Since an extent descriptor can not describe storage on another volume, if the file's first extent is on a different volume from the primary mcell, the BSR_XTNTS record will describe a zero length extent. The first BSR_XTRA_XTNTS record will describe the first extent on another volume.

4. DVN 3 and 4 striped (non-reserved) files.

   ```
   BSR_XTNTS -> BSR_SHADOW_XTNTS -> BSR_XTRA_XTNTS -> …
   -> BSR_SHADOW_XTNTS -> BSR_XTRA_XTNTS -> …
   ```

No reserved files are striped. Each stripe starts at page 0 in a BSR_XTRA_XTNTS record. There is no record on the disk that says how many stripes a file has. When opening a file, the extents are read. As each new BSR_SHADOW_XTNTS record is found, an in-memory stripe counter is incremented. Only at the end of the linked list of extent mcells will AdvFS know how many stripes a given file has. The mcell count in each BSR_SHADOW_XTNTS record is the count of mcells in that stripe. Each stripe starts with a BSR_SHADOW_XTNTS record but it may not have a BSR_XTRA_XTNTS record. The BSR_SHADOW_XTNTS record may describe all the extent in that stripe. In that case, a BSR_SHADOW_XTNTS record can point to another BSR_SHADOW_XTNTS record. Two stripes in one file are never created on one volume, but the `rmvol` or `migrate` utilities can form degenerate striped files with more than one stripe of a file on a single volume.

4.4.2 On-disk mcell record types

There are many different record types used to describe information contained in mcells. The following is a chart of the most common record types seen in mcell chains.

Record Type	Description
BSR_ATTR	Record describing file attributes; always in primary mcell
BSR_XTNTS	Record for first extent in primary mcell
BSR_XTRA_XTNTS	Describes additional extents
BSR_SHADOW_XTNTS	Describes the first extent of each stripe in a striped file. Also used as the first extent of a file on V3 domains.
BSR_PROPLIST_DATA	Describes file property lists
BSR_VD_ATTR	Describes volume attributes
BSR_DMN_ATTR	Describes permanent domain attributes
BSR_DMN_MATTR	Describes mutable domain attributes
BSR_MCELL_FREE_LIST	Describes the first page of the mcell free list
BSR_DEF_DEL_MCELL_LIST	Describes storage on the deferred delete list
BSR_DMN_SS_ATTR	Describes domain's vfast attributes
BSR_DMN_FREEZE_ATTR	Describes domain's freeze/thaw state
BSR_BFS_ATTR	Describes the fileset attributes
BSR_BFS_QUOTA_ATTR	Describes fileset quotas

4.5 Extents

Extents are contiguous areas of disk storage. The access structure has a list of extents that make up the file. As a file grows, it must acquire new storage. If the new storage is contiguous to the storage at the current end of the file, the file tends to be composed of a few large extents. If, on the other hand, each new page added to a file is at a random location on the disk, the file will be composed of many small extents. Files with fewer, larger extents have less metadata overhead and show better sequential read performance.

Extents are stored on the disk as a series of **descriptors**. A descriptor is a file page and corresponding starting disk block pair. An extent is the range of pages from one descriptor to the next. All the disk storage necessary to store the pages of the extent is contiguous on the disk starting at the block number specified in the first descriptor in a pair of descriptors. A series of descriptors is ended when the block is -1. There are also descriptor terminators that use the value of –2, and are associated with holes in clone files. See Section 5.1.5 for more information on extent maps in clone files.

For example, the following set of descriptors describes two extents followed by a hole, followed by one more extent.

Page	Block	
0	1024	Start of 15 page extent
15	2048	End of 15 page extent, start of 12 page extent
27	-1	End of 12 page extent, start of 29 page hole
56	64	End of 29 page hole, start of 1 page extent
57	-1	End of 1 page extent

The first extent is 15 pages long and starts at disk block 1024. The next extent is 12 pages long and starts at disk block 2048. There are no disk blocks for file pages 27 through 55. The last page, page 56, is stored starting at block 64. The descriptor for page 57 is needed only so the size of the extent is known.

4.5.1 In-memory and On-disk Extent Maps

An extent map is the set of all the descriptors needed to specify the disk storage for a file. On the disk the extent map for a file is stored in extent records in a chain of mcells. Each record can hold a fixed number of descriptors.

AdvFS keeps an in-memory copy of the extent map for files that have been opened. The in-memory extent map is kept in a `bsInMemXtntMapT` structure. This structure has a variable number of `bsInMemSubXtntMapT` structures which are stored in array format. Each valid `bsInMemSubXtntMapT` structure corresponds to an on-disk mcell extent record. The portion of the extent map covered by the `bsInMemSubXtntMapT` structure is called a subextent map. If the number of subextents grows, a new array (`bsInMemXtntMapT->subXtntMap`) with more entries is allocated. The old array is copied into the new array and the old array is freed.

The `bsInMemSubXtntMapT` structure has an array of page/block descriptors (`bsInMemSubXtntMapT->bsXA`). The number of in-memory descriptors (`maxCnt`) may not be as large as the number of on-disk descriptors in the corresponding mcell (`onDiskMaxCnt`). If the number of in-memory descriptors grows, a new array with more entries is allocated. The old array is copied into the new array and the old array is freed.

The following is the layout of the extent map structure - bsInMemXtntMapT:

origStart	index of first subextent to be replaced in upcoming change
origEnd	index of last subextent to be replaced in upcoming change
updateStart	index of first replacement subextent
updateEnd	index of last replacement subextent
validCnt	number of subextents with valid information
cnt	number of subextents in use, including changes
maxCnt	number of subextents allocated in array (some may not be used)
subXtntMap	array of in-memory subextent maps (bsInMemSubXtntMapT[maxCnt])

The following is the layout of the subextent map structure - bsInMemSubXtntMapT:

pageOffset	first page described in this subextent
pageCnt	number of pages described by this subextent
vdIndex	volume containing the mcell for this subextent record
mcellId	pointer to on-disk mcell corresponding to this subextent
onDiskMaxCnt	number of descriptors allocated in the corresponding mcell
updateStart	index of first descriptor that needs to be updated on the disk
updateEnd	index of last descriptor that needs to be updated
cnt	number of descriptors (bsXA) in use
MaxCnt	number of descriptors allocated in the bsXA array
bsXA	array of extent descriptors (bsXtnt[MaxCnt])

Although both structures have fields named cnt, updateStart, and updateEnd, they have different meanings. In the bsInMemXtntMapT structure, these fields refer to the subextents in the subextent array. In the bsInMemSubXtntMapT structure, these fields refer to the count of descriptors in the descriptor array. More detail on how these fields are used will be described in the following sections.

4.5.2 Modifying In-memory Extent Maps

AdvFS modifies on-disk extents by adding modified subextent maps to the in-memory extent map. The out-of-date on-disk subextent maps are then replaced with the modified in-memory subextent maps. This replacement is done at the same time the on-disk records are updated. The on-disk change is always done under transaction control.

In the bsInMemXtntMapT structure, the range of valid entries in the bsInMemSubXtntMapT array is 0 through (validCnt-1). A subextent map entry is considered "valid" when it has a corresponding on-disk mcell record. When a range of extents in a file changes (pages are added, truncated, or migrated),

AdvFS marks the range of subextent maps that include the modified extents in the `origStart` and `origEnd` fields. The modified subextent maps are replaced by new subextent maps located in `bsInMemSubXtntMapT` array entries `updateStart` through `updateEnd`. The value of `updateStart` is greater than or equal to `validCnt`.

Example #1

The following example illustrates a file with an array of five subextents, so `validCnt` is 5. Subextents 1, 2 & 3 are about to be modified. The first subextent index past the valid ones is 5, so `updateStart` is set to 5. It takes four subextents to affect the change, so `updateEnd` is set to 8. Once the changes have been written into subextents 5 through 8, `cnt` is set to 9 to indicate that 9 subextents are now in use. In this example, the `bsInMemSubXtntMapT` structure has 10 array entries, so a new array does not need to be allocated, and `maxCnt` remains at 10.

Each extent descriptor in the `bsInMemSubXtntMapT->bsXA` array of extents from `updateStart` to `updateEnd` corresponds to an mcell that must be modified. The range of descriptors from 0 to `updateStart` is not changed. The extent descriptor range from `updateStart` through `updateEnd` must be changed on disk.

`bsInMemXtntMapT`

`bsInMemSubXtntMapT[index]`	0	1	2	3	4	5	6	7	8	9
Field values:		origStart		origEnd		updateStart			updateEnd	
Subextent array size: `maxCnt = 10` **Valid subextents:** `validCnt = 5` **Subextents in use:** `cnt = 9`										

As explained previously, the meanings of `updateStart` and `updateEnd` are different in the two extent and subextent structues. In `bsInMemXtntMapT, updateStart` through `updateEnd` specify a source replacement range for subextents. The range does not have to be contiguous with the replacement range or at the end of the current valid subextents. The update range is moved to replace the original range. The number of valid subextents in the updated subextent array is

```
validCnt – ((OrigEnd - origStart) + (updateEnd - updateStart))
```

In the example above, `updateStart` could have been 6. If this were the case, subextent 5 would not be part of the original valid subextents, nor would it be part of the replacement subextents.

In contrast, in `bsInMemSubXtntMapT`, `updateStart` is the beginning of new descriptors in the descriptor array that are already in place. No descriptors in the update range have to move within the descriptor array. The updated descriptor array has `updateEnd+1` (that is, `cnt`) valid descriptors. In the example above, subextent 5, which replaces subextent 1, may be changing only the last half of the extents in the subextent. In this case, if there were six descriptors, `bsInMemSubXtntMapT->updateStart` would be 3. The descriptors from 0 through 2 are the same in subextent 1 and 5. The descriptors from 3 through 5 are different in subextent 1 and 5. When subextent 5 replaces subextent 1, then descriptors 3 through 5 need to be updated on the disk.

Example #2

The following example adds five pages of storage to a sparse file at page 23. The original file is described by two mcells (and two subextents) with extents. Page 23 is described in the first subextent. Therefore, the first subextent is modified.

bsInMemXtntMapT	**Original**	**Intermediate**	**Final**
origStart		0	
origEnd		0	
updateStart		2	
updateEnd		2	
validCnt	2		2
cnt	2	3	2
maxCnt	3		3
subXtntMap[0] ← denoted by (origStart, origEnd) range			
pageOffset	0		0
pageCnt	42		42
updateStart			
updateEnd			
cnt	5		7
maxCnt	10		10
bsXA[0]	0, 1024		0, 1024
[1]	10, 512		10, 512
[2]	12, -1		12, -1
[3]	40, 2048		23, 4096 (added)
[4]	42, -1		28, -1 (added)
[5]			40, 2048
[6]			42 ,-1
subXtntMap[1]			
pageOffset	42		42
pageCnt	3		3
updateStart			
updateEnd			
cnt	2		2
maxCnt	10		10
bsXA[0]	42, 8092		42, 8092
[1]	45, -1		45, -1
subXtntMap[2] ← denoted by (updateStart, updateEnd) range.			
pageOffset		0	
pageCnt		42	
updateStart		3	
updateEnd		6	
cnt		7	
maxCnt		10	
bsXA[0]		0, 1024	
[1]		10, 512	
[2]		12, -1	
[3]		23, 4096 < updateStart	
[4]		28, -1	
[5]		40, 2048	
[6]		42, -1 < updateEnd	

The first subextent map is copied to new subextent map 2. The range from `updateStart` to `updateEnd` (2 to 2) indicates the new subextent that replaces the old subextent in the range `origStart` to `origEnd` (0 to 0). In the new subextent map, `updateStart` through `updateEnd` (3 to 6) indicate the changed descriptors. The last descriptor is `updateEnd`.

After the in-memory extent map is changed, the on-disk mcells are changed to agree with the in-memory extent map. All changes are done under one transaction. Storage can be appended to the end, inserted into the middle, or removed using this mechanism.

4.5.3 Striped Extent Maps

The extent map for a striped file (see Striping, section 5.4) consists of an ordinary extent map for each stripe. Each extent map starts at page 0 and the order of the maps is derived from the order of the mcell chain, which determines which extent map page 0 describes file page 0 and which extent map page 0 describes file page 8, etc.

One extent in a stripe extent map may actually describe two or more discontiguous file page ranges. This is because an extent that describes a set of contiguous disk blocks more than eight pages long is describing two or more stripe segments of no more than eight pages apiece.

When a striped file is cloned, the clone file is striped over the same volumes as the original file. The clone file has the same number of stripes and the same stripe width as the original file.

4.6 Sparse Files

Sparse files are files that do not have disk storage for all their pages. Pages without storage are called **holes**. When a page in a hole is read, AdvFS sees that there is no disk data for that page and creates a page of zeros in memory. The `read()` system call returns the page of zeros.

4.7 File Operations

4.7.1 Create

From the logical file hierarchy perspective, a file is created when its name is inserted into its parent directory. File creation consists of 1) inserting the file's name and tag into the parent directory, 2) creating a primary mcell with metadata for the file in the BMT and 3) adding an entry into the fileset's tag file pointing to the primary mcell.

4.7.2 Open

Opening a file (done via *open()* or *creat()* calls) involves translating a file path name to a file descriptor that can identify the open file (typically a file handle), and preparing the file to be used. Most of the work here is traversing the pathname through the directories, retrieving the tag for the file, looking the tag up in the tag directory, and then setting up in-memory structures (`vnode, bfAccess`) that can be used to make future file references more efficient. Note that there are routines above the AdvFS layer that also allocate structures when a file is opened. For example, there is a `struct file` that is used as a link between open file descriptors for a process and the underlying `vnode` for the open file.

When a file is opened, the application can specify the mode in which the file is opened. Some of these modes are obvious, and some need a bit of explanation. The following is a table of some of the open modes that can be used for AdvFS files and some of the salient points regarding that mode.

Open(2) flag	Explanation
O_RDONLY	Open the file for read access only
O_WRONLY	Open the file for write access only
O_RDWR	Open the file for read and write access
O_CREAT	Create the file if it does not exist
O_APPEND	All writes will be appended to the end of the file. The kernel synchronizes writes among racing threads to assure that no data is interspersed from the different writes.
O_TRUNC	Truncate the file to zero length when it is opened. All previous data is lost.
O_EXCL	If the file already exists and both O_EXCL and O_CREAT have been specified, the open will fail. This prevents accidentally losing the file's data by specifying O_CREAT.
O_DIRECTIO	Open the file for direct I/O. This mode causes the data to be transferred directly from the application buffer to the disk, bypassing the buffer cache. When one process opens a file in this mode, that file is opened for direct I/O for all other processes, whether it was requested or not. Also, once opened for direct I/O, that mode will remain in effect until all processes close that file.
O_CACHE	Open the file for cached I/O. This is the opposite of direct I/O, but this cannot be used to turn direct I/O off for a file. This is the default value, so it is really not necessary to specify this flag.
O_SYNC	Specifies that each write should be synchronous, not returning until the data and updated metadata have been flushed to disk. The configurable variable 'strict_posix_osync' must be set in order for AdvFS to comply fully with POSIX requirements. This will force AdvFS to write the metadata even if the size of the file has not changed. For performance reasons, if O_SYNC is not specified, AdvFS will not update the metadata before returning if the size has not changed.
O_DSYNC	Specifies that each write should be synchronous, not returning until the data (but not the file statistics metadata) have been flushed to disk.
O_RSYNC	Specifies that any pending writes be flushed to disk before a read can be satisfied. This forces AdvFS to flush unwritten data during a read operation. Valid only if O_SYNC or O_DSYNC are also specified.

4.7.3 Read, pread, readv, aio_read

Reading a file can be done via a variety of interfaces, including the *read(), pread(),* and *readv()* routines, which all require a file descriptor obtained from the *open()* call. The file can also be memory-mapped via *mmap(),* and the read will occur when the mapped memory region is accessed.

Because all read operations are inherently synchronous (the read call cannot return until the data is fetched from disk and placed into the application buffer), there is also an *aio_read()* call that can be used by an application to make reads appear to be asynchronous. That is, an application can call *aio_read()* to start a read, and then go do some additional processing. *aio_error()* can be called at a later time to check the status of the read. These routines are part of the **Asynchronous IO** (**AIO**) package. See section 14.2 for more information.

4.7.4 Write, pwrite, writev, aio_write

Writing a file can be done by a variety of interfaces, including the *write(), pwrite(),* and *writev()* routines which all require a file descriptor obtained from the *open()* call. The file can also be memory-mapped via *mmap(),* and the write will occur when the mapped memory is modified (although this does not force the data out to disk).

Writing to a file that is opened for cached I/O is typically an asynchronous operation. This means that the data is written to the buffer cache when the *write()* call returns to the application, but the data will be flushed to disk at some later time. This flushing can occur under application control via calls to *fsync()* or *msync(),* by the operating system which will flush dirty buffers to disk periodically via the *sync()* interface or by the smoothsync algorithms (section 8.9).

When a file is opened for direct I/O, writes bypass the buffer cache and go directly from the application's buffer to the disk. The *write()* routine will not return until the data is on disk. Because direct I/O writes are synchronous, and because the I/O latency is long, many applications that use direct I/O choose to simulate asynchronous behavior by using the **Asynchronous IO** (**AIO**) package. This allows them to call *aio_write(),* do some additional processing, and then check the status of the write at some later time by calling *aio_error()*.

4.7.5 Close

Closing a file is a fairly simple operation. From the application point of view, this really just involves disassociating the file descriptor that was obtained on the *open()* call from the underlying file. There are some underlying operations that AdvFS does at this time, and they are discussed in Section 4.7.6.

4.7.6 Delete

Files are deleted via the `unlink()` system call. This causes the name of the file to be removed from its directory, and it can no longer be located by the logical file hierarchy at that point. However, if processes still have that file open for processing, the file itself does not get removed until the last process closes the file.

Most UNIX flavors do not provide an 'undelete' function, so if a file is removed accidentally, it is usually a cause for concern. AdvFS provides the trashcan concept to help alleviate some of the problems associated with unwanted deletes. This allows files that are deleted to be moved to a trashcan directory instead of being removed. (See Section 6.1.4)

4.7.7 Rename

Renaming a file is conceptually fairly simple: the name in the directory must be changed from the old name to the new name, assuring that the new name is not a duplicate. If the path is modified so that the file moves from one directory to another (within the same domain), the operation becomes a delete of the file directory entry from the old directory and an insert into the new directory. If the rename causes the file to move to a different domain, then the underlying storage will be different, so the whole file must be copied from the old domain to the new one, and then the old storage is deleted.

4.7.8 Fcntl() and ioctl()

These are two interfaces to allow manipulation of open files. *fcntl()* performs controlling operations on a file specified by the file descriptor returned by the *open()* call. A variety of functions can be performed, and several are filesystem-specific. One that is unique to AdvFS is F_GETCACHEPOLICY which will retrieve the cache policy for the open file. This allows an application to determine if a file is opened for direct I/O or for caching. There is also an F_ADVFS_OP function that allows activating atomic-write data logging (ADL) or forcing synchronous I/O, as well as retrieving the current I/O mode for the file (synchronous, asynchronous, ADL). Another interesting *fcntl()* function is F_GETMAP which retrieves the sparseness map for the file. This allows applications such as `vdump` to determine where storage holes exist in a file. The `man` pages have more information on how to use this function.

ioctl() is another useful interface that is typically used to control devices, and is used internally by AdvFS for controlling and retrieving information about disk devices. For example, when a disk is added to a domain, AdvFS uses the *ioctl(GETGEOM)* call to determine many parameters of the disk and to optimize future interactions with that device.

4.7.9 Truncate

Open files can be truncated via the *truncate()* and *ftruncate()* calls; the file can also be extended using this interface. If the new length of the file is less than the previous length, all data beyond the new end-of-file (EOF) is deleted, and the underlying storage is returned to the domain for reuse. If the new length is greater than the previous length, one byte of zero value is written at the offset of the new length. Any complete pages between the previous EOF and the new EOF will not have storage. These pages will be considered part of a sparse hole.

Internally, the VFS routine *vtruncate()* compares the new and old lengths of the file and, if the file is being truncated, dispatches through VOP_SETATTR() and the *msfs_setattr()* routine. However, if the file is being extended, it will set up a VOP_WRITE() call which will go through the *fs_write()* path to write the single zero-value byte at the new EOF.

4.7.10 Hard and symbolic links

A hard link is an additional name for an existing file, and is created using the `link` or `ln` commands. After a link is created, the file has more than one name. For AdvFS, this simply means that there are several unique directory entries, each of which contains the tag number for the same file. Because tags are unique within a fileset context, hard linking across filesets is not permitted.

A symbolic (or soft) link contains the name of the file to which it is linked, and is created using the command: `ln -s sourcename linkname`. These entries in the directory have unique tag numbers, and, in fact, are separate files. What links them together is that the symbolic link file stores the

name of the 'linked-to' file. This name is stored in the metadata (specifically the BMTR_FS_DATA record), provided that the name is less than 261 (BS_USABLE_MCELL_SPACE) bytes long. If it is longer than that, then file extents are allocated and the link is stored in the file itself. The former method is more efficient, so symbolic links with short names tend to perform better than ones with very long names.

4.7.11 Memory Mapping

As mentioned briefly in the read and write sections, a file can be memory-mapped. This is a process whereby a file, or a portion of a file, can be represented by a range of memory. After the file is memory mapped, then simply reading the memory reads the file, and modifying the memory will cause the file to be modified. Mapping is done using the `mmap()` and `munmap()` system calls, and can provide significant performance gains.

This mechanism is largely handled by the Virtual Memory subsystem. If a memory mapped page is touched (either for reading or writing) and this page is not resident in virtual memory, VM will **fault** the page. Faulting is a process whereby the page is brought into virtual memory and made available to the application. For a memory-mapped file, part of this process involves reading the page from disk into memory so that the data can be read and/or modified. When a file page is faulted and brought into memory,VM calls a filesystem-specific 'getpage' routine via the *FSOP_GETPAGE()* macro. For AdvFS, this will invoke *msfs_getpage()*. Further discussion of the *msfs_getpage()* internals is presented in Section 7.6.

4.8 Bitfile Operations

4.8.1 Bitfile States

A bitfile can be in one of the following states:

Invalid	bitfile does not exist
Creating	bitfile is being created
Valid	bitfile exists and is accessible
Deleting	bitfile is being deleted

The bitfile state is maintained in its primary mcell and its corresponding access structure (if one exists), in `bfAccess->bfState`. The figure below shows the possible state transitions:

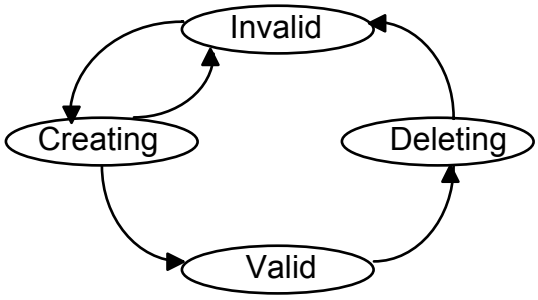


Figure 19: File state transitions

4.8.2 Lookup, open

Most of the work of opening an AdvFS file is done by the *msfs_lookup()* routine via VOP_ACCESS(). The lookup routine is called from *namei()* inside *vn_open()* when the file is opened to convert the filename path to a vnode. The lookup routine is responsible for determining if the file exists, setting up the vnode for the file, and indirectly allocating the `fsContext` and `bfAccess` structures for the file. The *vn_open()* routine will also call VOP_OPEN() which dispatches to *msfs_open()* for an AdvFS filesystem. This routine doesn't do much for opening most files. As of Tru64 UNIX Version 5.1B, *msfs_open()* is used only for checking the cache mode of the file and enabling direct I/O if that was specified in the *open()* call. Other than that, most the work of opening the file is done in the lookup routine.

After a file has been opened, there are several in-memory structures associated with that file, notably the `vnode, fsContext`, and `bfAccess` structures. There is only one copy of each of these structures, no matter how many times the file has been opened. Being able to find these on a running system or in a crash dump is often helpful. The following are some hints about finding and using files that are open.

- To find all the processes that have a particular file open on a running system, use the `fuser <file>` command. This will print a list of process numbers (PIDs) for the processes that have this file open.
- To find the `struct file` structures associated with every file that a process has open, use `kdbx` or `crash` and the command `ofile –pid <pid>`. This will print a series of pointers for that process. This is the process' file descriptor table, and contains an entry for each open file, including multiple opens of the same file. Print the contents of each pointer as a `struct file *`; if the `file.f_type` field has a value of 1 (DTYPE_VNODE), then the pointer at `file.f_data` is the `vnode` for that file. Note that if the file is open for direct I/O, the VDIRECTIO bit will be set in the `vnode.v_flag` field since this is one of the attributes that is file-wide.
- Once you have the `vnode`, you can find the `bfNode` structure that is in the `vnode.v_data` field. Do this by finding the address of `vnode.v_data`, and then casting that address as a `struct bfNode`. Within the `bfNode` are pointers to both the `fsContext` and `bfAccess` structures for this file.

4.8.3 Create

File creation is done via VOP_CREATE() from *vn_open(),* and this dispatches into *msfs_create()* and *fs_create_file()* where most of the file creation logic resides. The work done here is all under transaction control so the various pieces can be rolled back if any succeeding part fails. The primary steps vary slightly depending on whether a regular file, a directory, or a symbolic link is being created, but are essentially:

1. Get a new vnode.
2. Start a transaction of type FTA_FS_CREATE_1.
3. Call *rbf_create()* to get the next tag available for this file
4. Get a new mcell for the file
5. Initialize the bfAccess, bfNode, and fsContext structures

6. Put the file onto the mount queue
7. Write the BMTR_FS_STAT record to the primary mcell
8. Allocate storage, if needed, for the file
9. Insert the new file into the directory
10. Initialize data for directories and symbolic links
11. Write updated stats to disk for the new file and its directory
12. Finish the transaction.

Note that several of the substeps will encompass their own subtransactions. In addition, there is a "root done" transaction that will call *create_rtdn_opx()* to set the on-disk bitfile state to VALID, and wake any threads waiting for this file to become valid. See Chapter 9 for more information about root done transactions.

4.8.4 Read, read-ahead, prefetch

Read operations can enter AdvFS via four different entry points depending on how the application is coded. Reads done via *read()* and its sibling routines will dispatch into AdvFS via the VOP_READ() macro in the VFS layer. This will end up calling *msfs_read()* and then *fs_read()* where most of the read logic resides. Reads done via memory-mapped files are done when the memory page is faulted (see Section 7.6) and *msfs_getpage()* is called to bring the file page into the buffer cache. Reads done via *aio_read()* (see Section 15.2) enter AdvFS via the *msfs_strategy()* routine, which sets up a `struct uio` and calls down into the *vn_pread()* function and ultimately into *fs_read()* like the normal *read()* call. NFS preferentially uses the FSOP_GETPAGE() call for reading a page. However, if a file is opened for directIO, *msfs_getpage()* will return an error to NFS, and it will revert to using VOP_READ() for that file. This is because *msfs_getpage()* does not use the active range locks to synchronize with other threads, and since it is used heavily for memory-mapped files, we did not want to add the active range locking overhead in that path.

AdvFS uses two special algorithms for optimizing read performance:

- **Prefetch** is used when a single read request from an application requires reading more than one AdvFS page.

  When the first page is referenced, AdvFS sets up the rest of the pages to be read and puts the buffers onto the blocking queue. (Actually, if the first page is already in the cache, the remaining pages are put onto the blocking queue; if they all need to be read, they are all put onto the blocking queue at the same time.) Once the data for the first page is transferred to the application buffer, the rest of the pages are already in the buffer cache, eliminating any more waiting for I/O completion. Only copying the data from the buffer cache to the application buffers is required.

  Prefetching is limited to 64 pages by the global variable `max_prefetch_pgs`. This limit is maintained to minimize potential problems if the system is low on memory and the application is trying to read a large number of pages. The variable is not configurable or modifiable by an AdvFS utility. The only way to change it is with a debugger.

- **Read-ahead** is similar to prefetch, but instead of setting up I/O for pages in the current I/O request, read-ahead is used to get sequentially read pages into the cache before the user requests them.

  When a read is sequential to the last one, AdvFS assumes that the file is being read sequentially and starts read-ahead processing. The last page read is recorded in the file's vnode (`vp->v_lastr`). While the page being requested is being referenced, the I/O subsystem sets up two I/Os for the

succeeding pages. Each I/O is of `preferred_transfer_size` to optimize the device's abilities. A trigger page is set to point to the last page of the first I/O. Eventually, the application reads the trigger page. This page is still in the cache, and `preferred_transfer_size` bytes following it are also still in the cache. At that time, AdvFS schedules an additional single I/O worth of pages to be read; these pages follow those that have already been read. Then the trigger page is reset. This continues until all of the data has been read.

For example, assume that the `preferred_transfer_size` for the underlying device is 100 pages and that an application reads a file sequentially in 8K increments starting at page 0. The process is as follows:

1. When the application reads page 0, that page is referenced, read from disk, and loaded into the cache.
2. The data is copied from the cached page into the user's buffer and `vp->v_lastr` is set to page 0.
3. The *read()* call returns the status to the application. When the application requests a read of page 1, AdvFS notices that page 1 immediately follows the last page read (page 0) and goes into read-ahead mode.
4. During the read of page 1, AdvFS schedules I/O for two page ranges, each of size `preferred_transfer_size` (pages 2 through 101 and 102 through 201).
5. AdvFS sets page 101 to be the trigger page and updates `vp->v_lastr` to page 1.
6. Hopefully all subsequent reads of pages 2 through 101 occur without having to wait for any I/O to complete because those pages are already in the cache.
7. When the trigger page 101 is read, AdvFS starts a single I/O for the page range of pages 202 through 301 and sets the new trigger page to 201.
8. This pattern continues until the end of the file is reached or until the application stops reading the file sequentially.

One limitation of the read-ahead algorithm is that there is only one sequential read point maintained for each file. This means that if two applications are reading the same file sequentially, but one is reading at offset page 0 and the other at offset page 10,000, the read-ahead algorithm does not work as efficiently as if each application were reading from a different file. Removing the read-ahead point from the `vnode`, putting it into the `bfAccessT` structure and allowing for a dynamic number of active read offsets could be one way to fix this.

This algorithm can also be defeated if there are few free Unified Buffer Cache (UBC) pages on a busy system. Assume the read-ahead algorithm has retrieved a given range of pages. If the UBC reclaims some or all of these before the original thread can retrieve the data from the cache, then the read-ahead has been defeated, and the I/O must be redone.

Having a single thread monopolize the buffer cache with read-ahead pages could adversely impact other processes. To prevent this, two global variables are used and can be tuned with a debugger if desired. `AdvfsReadAheadNumIOs` determines the number of page ranges that are read ahead. The default for this variable is 2, as shown in the example above. `AdvfsReadAheadMaxBufPercent` limits the number of pages that can be available to a single thread doing read-ahead to a percentage of the pages in the UBC. The default is 5%.

The device-specific variable `preferred_transfer_size` can be determined by looking at the `Rblks` and `Wblks` output of the `showfdmn` command or the `rblks` and `wblks` output of the `chvol` command. The variable can be reset with the `chvol` command and either the `-r blocks` or `-w blocks` options.

4.8.5 Write

Write operations can enter AdvFS via several different entry points depending on how the application is coded. Writes done via *write()* and its sibling routines will dispatch into AdvFS via the VOP_WRITE() macro in the VFS layer. This will end up calling *msfs_write()* and then *fs_write()* where most of the write logic resides. Writing to memory-mapped files is done when the memory page is faulted (see Section 7.6) and the memory is modified. In this case, the pages are not written to disk until the application calls `msync()` or the UBC page is otherwise flushed to disk at the request of the UBC or AdvFS. Writes done via *aio_write()* (see Section 15.2) enter AdvFS via the *msfs_strategy()* routine, which sets up a `struct uio` and calls down into the *vn_pwrite()* function and ultimately into *fs_write()* like the normal *write()* call. NFS appears to use the VOP_WRITE() macro for all its writes.

Most of the logic for writing files is in the *fs_write()* routine. This routine has two basic sections. First this routine checks if the write operation will require storage to be added to the file. If so, it attempts to add the storage by looping through all the pages to be written and calling *fs_write_add_stg()* for any that need to have storage added. If storage allocation is successful, or if part of it fails, but some part of the write can still be processed, then the copying of data from the application's buffers to the cached pages commences in the second part of the routine. If a file has been opened for direct I/O, the write is handled by calling *fs_write_direct()* to setup the I/O. Otherwise, the transfer is done by looping through all pages, pinning them, copying the on-disk data to the cached page, and then unpinning the page. Any transaction management required for files opened for atomic-write data logging is handled in this loop as well. After all the pages are updated, there is some cleanup such as marking the `fsContext` structure if the file statistics need to be updated, updating the new file size, and possibly flushing the cached pages to disk if required.

When storage is allocated in the write path, part of the storage allocation path must zero the newly-allocated pages. This is done in *fs_zero_fill_pages(),* which has a performance optimization to skip the zeroing of any pages that will be completely overwritten in *fs_write()* after the page is allocated. This saves the overhead of pinning, zeroing, and unpinning each of these pages.

Part of the write algorithm is to keep track of the number of sequentially written pages for a performance optimization called 'aggressive write flushing'. This optimization is intended to prevent low-memory situations because most of the UBC pages are dirty and must be flushed to disk before they can be used for other purposes. It works by tracking the number of pages that have been sequentially written for the current file, and if it exceeds the default value of 32 (stored in `bfap->seqWritePgMax`), will check the number of dirty UBC pages. If the number of free VM pages is less than a 'prewrite threshold' indicating that free memory is low, or if the number of UBC pages that are dirty or busy exceeds a predetermined threshold (default is 70% in global variable `aggressiveWritePcnt`), then the aggressive write algorithm causes the pages being written to be flushed to disk before the `write`() call returns to the user. The whole point of this is to prevent a single thread from depleting the UBC available pages by doing a large number of sequential writes. This algorithm actually slows the writer if the amount of memory available is becoming critically low. If the aggressive write algorithm is turned off (by setting the global variable `aggressiveWriteFlush` to 0), then the cached writes are not flushed to disk until smoothsync or the UBC initiates the flush.

4.8.6 Close

When a file is closed for the last time (no other processes have the file open), AdvFS determines if the last page should be moved into the fragment bitfile. See Section 6.2 for a discussion of when this happens. In addition, if the file has pre-allocated pages at the end of the file, they are truncated. This is done by calling *bf_setup_truncation()*. If necessary, the file stats are updated under transaction control by calling *fs_update_stats()*. Statistics that can be updated include creation/modification/access times and file size. If a file is created or deleted, then its parent directory modification time should also be updated when the file is closed.

4.8.7 Delete

File deletion is a complex set of steps that comes through VOP_REMOVE() into *msfs_remove()*. Because AdvFS supports trashcans (see Section 6.1.4), the delete code first determines if there is a trashcan directory for this file, and if so, calls *msfs_rename()* to move the file to the trashcan. If the file is actually going to be deleted, then the following primary steps are followed:

1. Seize the file_lock for the file and its parent directory
2. Start an FTA_OSF_REMOVE_V1 transaction.
3. Decrement the file's link count (`fscontext->dir_stats.st_nlink`)
4. If the link count is not zero:
    a. Call *remove_dir_ent()* to remove the file from the parent directory
    b. Flush the modification time to disk
5. Else the link count is zero, so actually remove the file:
    a. Call *cache_purge()* to remove the file's name from the name cache.
    b. Call *rbf_delete()* to put the file into the BSRA_DELETING state and put the file's primary mcell onto the Deferred Delete List (DDL). If this is an original file and the file has a clone, the file is simply marked as being deleted, but is not actually deleted at this point.
    c. Call *remove_dir_ent()* to remove the file from the parent directory.
6. Commit the FTA_OSF_REMOVE_V1 transaction.
7. Unlock the file and parent directory.

Some of the work of the deletion is done by root done transactions for the FTA_OSF_REMOVE_V1 transaction. If the delete is being finished (not undone), then this invokes *bs_delete_rtdn_opx()*. This routine makes sure the bitfile set is still open, and then calls *tagdir_remove_tag()* to remove the file's tag from the bitfile set's tag directory.

If a file is deleted and it has a clone, then the clone still has a dependency on the original file. The original file is removed from the directory so it is not available under its old name, and then it is marked as 'delete with clone' in its BSR_ATTR record on disk. When the clone is removed, this file will then be deleted.

After a file's primary mcell is moved to the DDL, its storage will be returned to the domain for later reuse. See Section 4.7.10.2 for information about storage deletion and the use of the DDL.

4.8.8 Truncate

File truncation (new file size is smaller than original file size) is dispatched through VOP_SETATTR() and *msfs_setattr()*. The truncation process requires the following steps:

1. Get bfAccess and fsContext structures
2. Seize the `fsContext.file_lock` and an active range lock
3. Update the new file size and statistics
4. Start an FTA_OSF_SETATTR_1 transaction.
5. Call *fs_delete_frag()* if the file is being truncated such that the frag must go away.
6. Call *fs_trunc_test()* to see if the file can be truncated.
7. If it can be, call *bf_setup_truncation()* to do the first phase of storage deallocaton.
8. If truncating to an offset within a page, pin the trailing page, zero out the range beyond new EOF, and unpin the page.
9. Call *fs_update_stats()* to flush the new stats (size and times) to disk.
10. Commit the FTA_OSF_SETATTR_1 transaction.
11. Call *stg_remove_stg_finish()* to finish the storage deallocation.
12. Release the active range lock and the file_lock.

The active range lock is taken in this path to coordinate with direct I/O threads to prevent I/O in a range of the file that is being truncated.

4.9 Copy on Write (COW)

Copy-on-write is relevant when a file has a clone in a cloned fileset. Before a page in an original file can be modified, that page must be copied to the clone. This happens only on the first modification of that page after the clone has been created. When a given page is COWed, up to 31 successive pages may be copied to the clone at the same time. COWing more than one page is done anticipating that sequential pages in the original file will be modified. With this procedure, the clone file is less fragmented than if only one page at a time were allocated and copied. Note that COWing multiple pages is limited to the active range when a file is opened for direct I/O. This is done so that only pages within the protection of the active range lock will be modified (this is only true in later baselevels of 5.1B - see Section 8.12.3 for further details). Cloning is discussed more fully in Section 5.1.

4.10 Storage Allocation and Deallocation

4.10.1 Allocation

A main goal of storage allocation is to allocate disk space to a file as contiguously as possible. This enables the buffer cache and I/O scheduler to perform large I/O requests (prefetching and consolidated writes) and to minimize the number of I/O's required to read and write to a file. So, even when a write operation from an application is for just one page, AdvFS may preallocate more than that page to ensure that when the write request for the next page arrives, that page will be allocated adjacent (on disk) with the previous page. (See section 4.5 for more information about extents).

A typical storage allocation has the following steps:

1. An application program calls the `write(2)` system call to write a data buffer to a file. When writing sequentially, the data is appended to the file.

2. The `write(2)` system call causes the AdvFS `fs_write()` routine to be called. If this routine determines that there is no disk space allocated for the write request, it calls `rbf_add_overlapping_stg()` to allocate one or more pages to the file.

3. `rbf_add_overlapping_stg()` calls the storage bitmap routines to find one or more free space extents on disk that satisfy the storage request. It marks the bits in the bitmap as 'allocated' and it updates the file's extent map to refer to the newly allocated storage.

4. `fs_write()` then transfers the data buffer to the newly allocated space in the file.

The preallocation of pages is controlled by `fs_write()`. It uses a storage allocation algorithm that increases the number pages per allocation as the file increases in size. The basic assumption is that as a file gets bigger it is more likely that it will get even bigger, so it is okay to preallocate more space to the file. The preallocation is currently limited to 16 pages. Preallocated space at the end of the file that is not used (end-of-file is before the preallocated pages) will be truncated when the file is closed (last close if multiple applications have the file open).

Disk storage is preallocated when a file is extended. If a file is growing at two or more sites (the end of the file and/or some sparse holes are being filled), each site gets its own preallocated storage. In some cases, this storage is not removed from the file when the file is closed. If a file is written sequentially and a hole is created by seeking forward past the end-of-file, the preallocated storage in the newly created hole is truncated before the new end-of-file is extended. However, preallocated storage added in an existing hole remains with the file when the file is closed.

To avoid costly I/Os and scanning of the SBM when searching for the free space on a volume, AdvFS uses a free-space cache. This cache is used to keep some information about the space still available on each volume. For more information on the free-space cache, see Section 1.4.4.

4.10.2 Object Reuse

After allocating disk space, `fs_write()` zero-fills all the allocated pages. However, it does not flush the zeros to disk synchronously, so it is possible that after a system failure the file's EOF is beyond the last written page and the pages between the last written page and the EOF may not have been zero-filled on disk. That is, the zero-filled buffers did not get written to disk. This is both data corruption and a security issue because now it is possible for the owner of this file to see old data that was on the disk in this page range. This ability to see data that is not part of the current file is called **object reuse**.

Preventing object reuse is called **object safety**. Activating object safety forces zero-filled pages to be written to disk before data is written to disk. This is done using the `chfsets` utility:

To enable object safety for all files in a given fileset:

```
chfsets –o objectsafety <domain> <fileset>
```

To disable object safety for all files in a given fileset:

```
chfsets –o noobjectsafety <domain> <fileset>
```

Because the flushing of the zero-filled pages to disk is extra overhead in the storage allocation path, the use of the object safety attribute will cause some performance degradation during storage allocation.

4.10.3 Deallocation

Storage deallocation is responsible for returning unused storage to the volume's free space cache following a file truncation or deletion. There are two main issues that must be dealt with when deallocating storage:

1. Storage deallocation is a potentially unbounded transaction because a file can consist of a large number of extents. In addition, large extents may represent many SBM pages which may cause a problem since there is a limit to the number of pages that can be pinned in a single transaction. So, it is difficult to deallocate storage and return it to the SBM in a single, bounded transaction.

2. Releasing resources must be done carefully so that there is no chance that the storage could be reallocated to another file before the transaction deallocating the storage has committed.

To deal with these issues, AdvFS deallocates file storage in two steps:

1. The storage to be removed is gathered as an extent map in a chain of mcells and put onto the **deferred delete list** (DDL). There is one DDL per volume, and is essentially a chain of mcells containing extent maps of storage to be returned to the volume's SBM. Once the extent maps are placed onto the DDL, the deallocation of this storage from its file is logically committed. If the system crashes after the storage has been moved onto the DDL, AdvFS continues the storage deletion process after the domain is reactivated. Even a very long list of mcells with huge extents can be cut from the file's mcell list with one cut. Very few on-disk changes are needed.

2. The second step in the storage deletion is to return the storage delimited by the extent maps on the DDL back to the volume's SBM. This is accomplished via a **continuation transaction** that performs as much work as possible given the constraints of the transaction system. When the continuation transaction commits, it also starts another continuation transaction if there is more work to be done. This takes care of the unbounded nature of storage deallocation.

4.11 Property Lists

A property list is a file attribute (or series of attributes) that can be set persistently, and later retrieved. Typically this data is a <name, value> pair, where the name is used to differentiate between the various attributes that can be stored for the same file. For instance, property lists are used to track managed

regions for files under DMAPI control (see Section 11.4), audit properties, manage **Access Control Lists (ACLs)**, and control other application-specific functions and attributes.

Property lists are sometimes referred to as "VFS+" because of their generic container properties; they are not unique to AdvFS. They can be set by the application using the routines in the `proplist.so` library; these include `setproplist`() and `getproplist`(). Ultimately, these library routines call the system vnode operation, VOP_SETPROPLIST() and VOP_GETPROPLIST(). For AdvFS, these dispatch to *msfs_setproplist()* and *msfs_getproplist()*.

Internally, the property lists are stored in a file's metadata, which is inside the mcell chain for that file in the BMT. The writing and updating of property lists is done under transaction control. Even though a property list can span mcells, the size of property lists is restricted by the BMT architecture and the transaction subsystem. Because updating an mcell requires the byte range (record) to be pinned, and there is a maximum of 7 record pins per page in the transaction subsystem, the property list may not span more than 7 mcells. Since each mcell is 292 bytes, and there is some overhead in the structures, the property list has a maximum length of 1548 bytes. In addition, the larger the property's name, the less room there is to store the data associated with that property (the name can be up to 255 characters).

The physical layout of property lists always contains 3 parts: the header, the name, and the value. The name and value contents are determined by the application. The header is determined by AdvFS, and has evolved somewhat through time. For DVN 3 domains, the header is a `bsPropListHeadT` structure that contains the length of the name data, the length of the value data, and some flags. So, if the property list data all fit inside one mcell, there would be the property list header, then the name, and then the value. Note that the name and value part of the data will be padded on disk so that they will be long-word aligned. For instance, if the name occupies 10 bytes, the value part of the data will start at byte 16 in the data list, since that is the next closest long-word (8-byte) boundary.

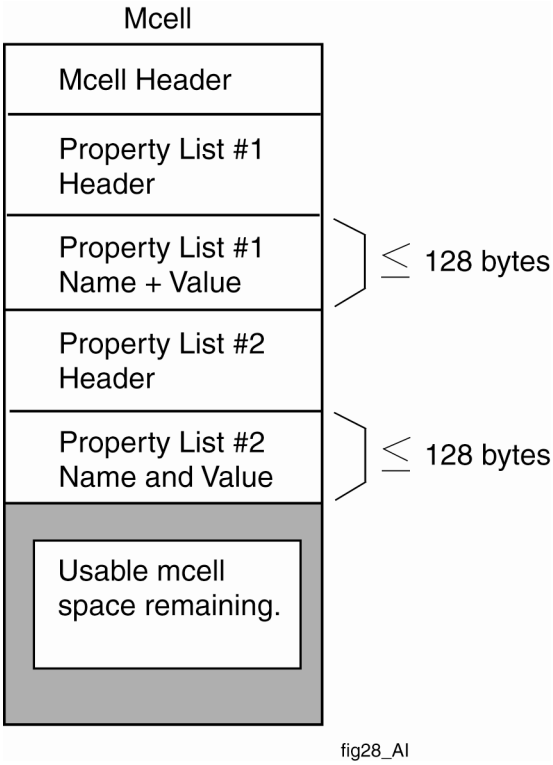


Figure 20: Small property list

A property list can be put into the mcells in two different fashions, depending on the size of the property list data. A 'small' property list entry is one in which the size of the name rounded up to the nearest long-word boundary plus the length of the value is less than 129 bytes. There can be several small property list entries per mcell. Since there is no facility for deleting mcell records, when a small property list entry is deleted, it is merely marked as not-in-use (the BSR_PL_DELETED bit is set in the flags field of the record) and can be overwritten at a later time.

If the property list entry is slightly larger, say 132 bytes, the property list entry would occupy an entire mcell. This is a 'large' property list entry and is designated by having the BSR_PL_LARGE bit set in the flags field of the property list header. A large property list entry has the advantage that, since the header and data records consume one entire mcell, this list entry can be deleted by unlinking the mcell from the primary mcell chain. This mcell is then available for general use and will appear on the free mcell list.

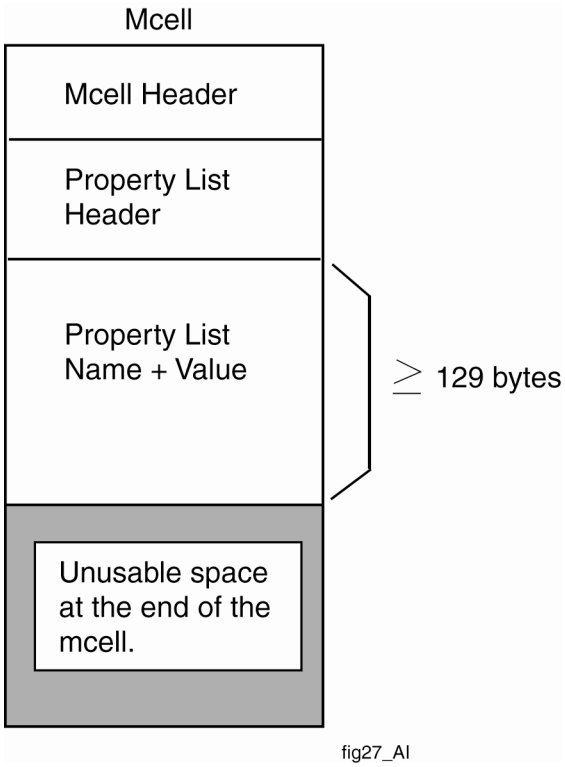


Figure 21: Large property list

If the property list entry is even larger such that it takes up room in two mcells, then the header, name, and as much of the value as possible are placed in the first mcell, and the rest of the data is placed in the following mcells (up to a maximum of 7 mcells). The structure used to describe the data in the second through sixth mcells is a `bsPropListPageT` structure, but that is really nothing more than a container for a buffer in DVN 3 domains.

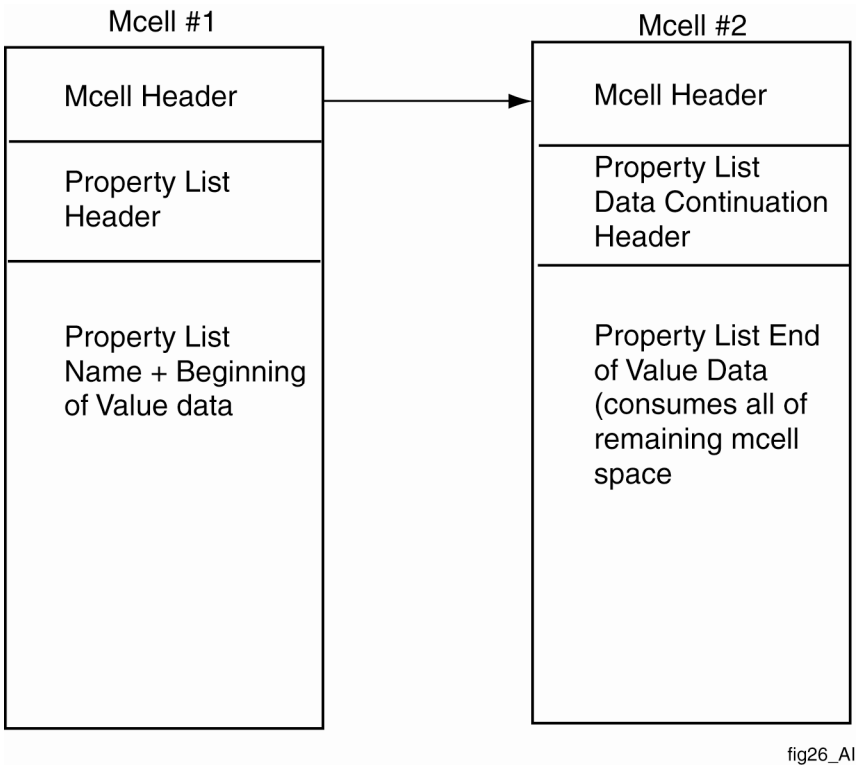


Figure 22: Property list spanning 2 mcells

In DVN 4, the property lists changed slightly to allow for detection and possible fixing of corrupt property list chains. A field called `pl_num` was added to the `bsPropListHeadT` structure to allow internal numbering of the property list chains. Then, two fields were added to each `bsPropListDataT` structure. One is `pl_num` and should match the `pl_num` in its header record. The second field is `pl_seq`; this is a sequence number and allows the ordering of the mcells within the property list chain.

Be aware that the `chfile` command uses an *fcntl()* call to invoke the VOP_GETPROPLIST() and VOP_SETPROPLIST() dispatch routines to set the persistent atomic-write data logging and the synchronous-write attributes for a file. However, these attributes are not actually stored in a property list. This entry point was chosen because AdvFS needed a mechanism that would allow an NFS client to turn these attributes on and off at the server. The property list interface was easy to use for this purpose. Inside *msfs_setproplist_int()* and *msfs_getproplist(),* however, this call is intercepted and the appropriate data are sent to *msfs_set_bf_params()* and *msfs_get_bf_params()* to handle these attributes, and no property lists are built or read.

Chapter 5: Unique AdvFS Operations

5.1 Cloning

Cloning can be thought of as creating a new fileset that is a snapshot of an existing fileset. The cloned fileset is in the same domain as the original fileset. This snapshot does not change, even though the data in the files of the original fileset continue to be modified. The following sections discuss some relevant aspects of cloning, particularly from the aspect of storage allocation.

5.1.1 Creating a clone

Clones are created using the following command:

```
clonefset <domain> <fileset> <clone>
```

where `<fileset>` is the fileset that will be cloned, and `<clone>` is the name of the new (clone) fileset. This clone fileset may be read, but none of its files may be modified. Thus, this fileset is essentially read-only to applications.

When a fileset is cloned, the original fileset's tag file is copied into the clone fileset. This ensures that both sets contain the same files. Both tag files originally point to the same metadata in the BMT; this may change later as the original files are modified. Also, the original fileset's `bfSetp->cloneCnt` value is incremented, and the clone's `bfSetp->cloneId` is set to this newly-incremented value. The original fileset always has a `bfSetp->cloneId` value of zero.

When a clone is first created, the cloned files have no metadata or disk space of their own, but the clone fileset itself gets metadata of its own. The tag file of the clone fileset is created as a copy of the tag file of the original fileset. Therefore, the primary mcell for each entry in the tag file is a pointer to the original file's primary mcell. As pages in the original file are modified, the differences must be maintained, and this is discussed in the next section.

First, however, consider that some files and filesets are never cloned. The 'hidden fileset' (Sections 3.3.1 and 4.2) associated with each domain is never cloned because it cannot be specified to the `clonefset` utility. In addition, the following files are never cloned: RBMT, BMT, SBM, LOG, MISC, and all tag files. These files are not cloned because they belong to the hidden fileset in each domain. There are no explicit checks in the clone or cow paths that specifically exclude these files. The fact that `(bfap->bfSetp->cloneSetp == 0)` is true for these files is adequate to keep the cloning/cowing code from being invoked for these reserved files.

5.1.2 Writing to a cloned original (COW)

The first time after cloning that a file is modified, the clone must have its own metadata allocated. If the file's `bfap->cloneCnt` value is less than the fileset's `bfSetp->cloneCnt` value, then the clone needs to have its own metadata allocated and then have its `bfap->cloneCnt` incremented. After that, the page(s) being modified (in the original file) must be copied to the clone file before any modifications are allowed to the original file. This procedure of copying the original data to the clone is called *copy on write* or COW. A given page must be COWed only the first time that page is modified after the fileset has been cloned.

As an optimization, when a page is COWed, that page plus up to 31 subsequent pages are copied to the clone. This is called **COW-ahead**, and is done anticipating that sequential pages in the original will be modified. This algorithm makes the clone file less fragmented than if only one page at a time were allocated and copied. Fewer than 32 pages may be COWed if: 1) some of the pages are already mapped in the clone, 2) the end of the original file (`bfap->nextPage`) or the end of the original file at the time the clone was created (`cloneap->maxClonePgs`) is reached, 3) this file is open for directIO, or 4) there is a hole within 31 pages of the page being modified.

If a file is open for directIO, COW-ahead is disabled in most baselevels. This was done to ensure that no pages outside the active range are touched. Starting in 5.1b BL25 and 5.1a BL26, COW-ahead for directIO files has been re-enabled, but it restricts the pages being COWed to those within the current active range. This compromise protects the data integrity while minimizing the number of extents generated in the clone file.

5.1.3 Reading from a Clone

When a file is read, AdvFS needs to access the extent maps to find the data on disk. In the case of a clone file, it first looks at the extents of the clone. If the requested page exists in the clone extents, then the clone page is a copy of the original, and the clone extents can be used to locate the data. If the requested page does not exist in the clone extents, the original file's extents are checked. If the page exists there, then the page has not been modified since the clone was created, and the requested page is read using the original file's extents. If the page is not in the original file's extents, then this page is in a hole, and zeros are returned for the read operation.

5.1.4 Sparse files and permanent holes

A cloned file maintains the sparseness of the original file at the time the fileset was cloned. A hole in the original file is recorded in the clone file extents as a **permanent hole**. They are the result of COWing a hole in the original file into the clone's extent maps. There is no mechanism that can create a permanent hole in an original file.

A permanent hole is denoted by a value of –2 in the `vdBlock` entry of a clone file's extent map. This indicates that a new extent is starting, but that there are no disk blocks for this extent within the clone file or the original file. A -2 descriptor must be followed by another descriptor to terminate the permanent hole extent. A -1 in a clone can be used for two purposes: 1) it terminates an extent and can be the last descriptor in a series of descriptors, or 2) it represents a normal hole in the clone which means that the page has not been COWed to the clone and that the original file extents should be referenced for the page.

When a page in an original file is modified and it has a clone, we check the clone's descriptor for that page. If the clone's page is in a normal hole (-1 descriptor), the page is COWed to the clone. If the clone's page is in a permanent hole (-2 descriptor), the page is not COWed to the clone. The clone's -2 extent represents a real hole in the original file and behaves as if storage were already allocated for those pages. Since clone files are never truncated or deleted (except when the entire clone is deleted), a permanent hole is never filled in.

When a page in a hole in the original file is written, the AdvFS code sees that the original file had a hole surrounding the page. A permanent hole the size of the original's hole is inserted into the clone extents. Any further pages written to the original file that fill in the hole do not change the clone file, because the clone file already has the pages mapped as a permanent hole in its extents. Even though the pages are mapped in an extent that doesn't have any storage, they are still considered mapped and are not filled in. A clone extent that starts with a -1 can be filled in. The pages within an extent that start with a -1 are considered not mapped. Rather, they signify that those pages have not yet been COWed to the clone.

Finding a hole within 31 pages of the page to be modified truncates the COW-ahead. All the pages from the selected page to the hole are copied to the clone. The hole is copied as a permanent hole in the clone. The size of the permanent hole is the size of the hole in the original file at the time the clone was created.

After the descriptor of the original file's hole has been transferred to the clone, the original file's on-disk extent maps are modified within a transaction. The original file's in-memory extent map is merged so that the updated subextent maps replace the original subextent maps. The in-memory extent map is now ready for the next modification under a separate transaction.

Adding storage to a clone file is done under one transaction because, if not, the second part (adding the hole) could be lost if the system crashed right after committing the first part (adding the storage). This transaction is accomplished by performing two sequential modifications to the in-memory extent map without changing the on-disk mcells. Then the on-disk mcells are changed to agree with the twice-modified in-memory extent maps. The second modification may have to create a new update range of subextent maps by copying and modifying the update range produced by adding storage. If that happens, `updateStart` may not be contiguous with the old valid subextent maps.

5.1.5 Deleting a file that has a clone

When an original file that has a clone is deleted, it is not actually deleted at that time. Its entry in the directory is removed so that it can't be found, but the underlying bitfile is not deleted. Its extent maps may still be needed for accessing data for the clone. A field in the `bfAccess` structure for the original file (`bfAccess->deleteWithClone`), is set to 1. This saves time when the file is deleted since the pages that have not been COWed to the clone do not have to be COWed at this time. The original bitfile is deleted when the clone fileset is removed. The `bs_bfs_delete()` routine, which is responsible for deleting bitfile sets, will call `delete_clone_set_tags()` if the fileset being removed is a clone. This routine walks through all files in the clone fileset freeing the storage for the cloned files. If the original file's `bfAccess` has its `deleteWithClone` flag set, it is deleted at that time.

5.2 File Migration

File migration takes advantage of the deferred delete list (DDL). The new storage is allocated and immediately put on the DDL so it will be cleaned up in the case of a crash before the migrate is complete. Then migrate sets up an in-memory extent map (called the `copyXtntMap`) to describe the new storage and in which to copy the all the data being migrated. This `copyXtntMap` is associated with the file's access structure and eventually becomes the file's `xtntMap` when the migration is complete.

The data is copied to the new storage location and the `copyXtntMap` is updated appropriately. Next, the pointers to `xtntMap` and `copyXtntMap` are swapped so that the new storage and extent maps describe the file. When the copy is transactionally complete, the old storage is put on the DDL.

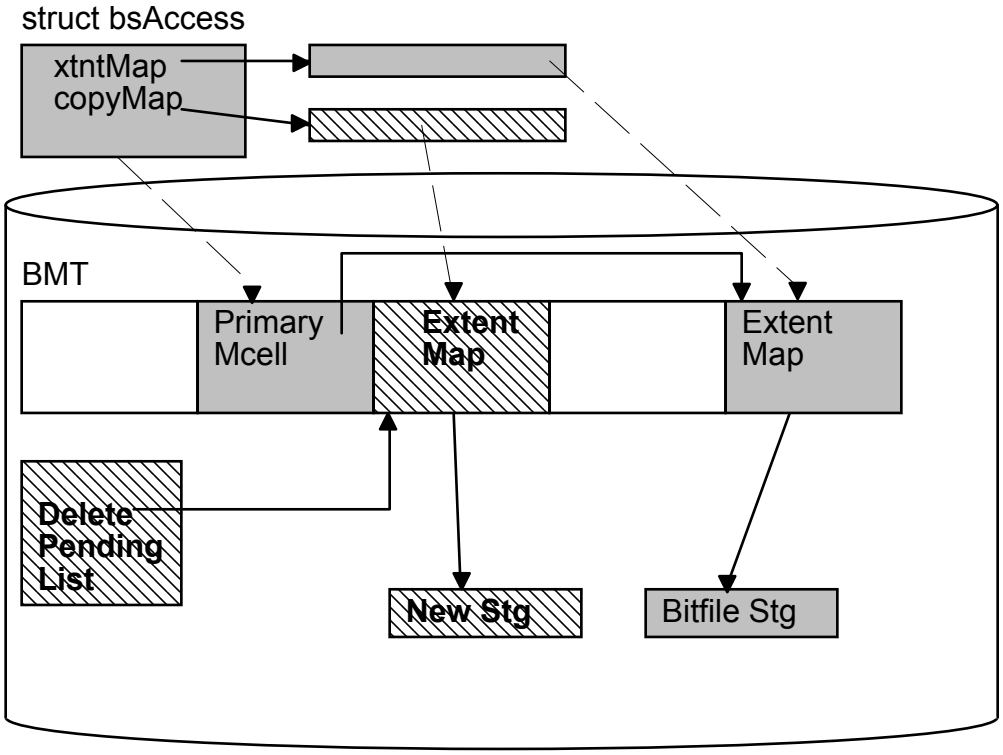
In the normal file migration case, we allocate new storage, migrate the file, and then close the file. On file close, the old storage is taken off the DDL and deallocated.

In the case of a crash while migrating a file, the DDL may contain the old storage or the new storage. When the fileset is mounted on reboot, the DDL is processed and any storage on the DDL is considered safe to deallocate, so we do just that. However, if we are in a cluster and we are performing failover, we do not want to process the DDL on remount. First we need to reestablish connections with any clients, since it is possible that files are still open on the client side (and the client does not know that the server is being relocated).

After we have reestablished necessary connections, we want to process the DDL. However, we don't want to remove storage for files that are still open on the client. The client will remove the storage as

appropriate on the last close of the file. We check for a bfAccess structure for each file on the DDL, and if found, that storage is not removed from the DDL.

The exception to this rule is with special files with tags less than six, which are not normal user files. These files (frag file, root directory, quota files, .tags) are opened on mount and are not closed until the filesystem is unmounted. For files in this case, we remove any old storage that exists on the DDL after reboot.



5.3 Defragmentation

For a variety of reasons, a file's extent map may become large and cumbersome because the storage for the file is discontiguous in the domain. When this happens, the file is said to be fragmented. Note that this is not related to a frag in the fragment file. A fragmented file is inefficient because its extent maps require more disk space and memory to store and it takes longer to walk through the extent maps to find any given page. For this reason, maintaining files with minimal extent maps is preferred from both disk utilization and performance perspectives. A utility called `defragment` is provided that will make the storage for files in a given domain more contiguous. Before you can defragment a domain, all filesets in the domain must be mounted. There are some additional recommendations for cleaning up a domain before running `defragement` to allow the best chance for getting the most contiguous files and for quantifying the efficacy of a `defragment` run. See the AdvFS Administration Guide for these recommendations.

In version 5.1B, the vfast utility was introduced. Vfast is a background defragmenter. It runs in the kernel and defragments the worst fragmented files as they are discovered. Vfast has largely eliminated the need to run defragment.

5.4 Striping

AdvFS can stripe individual files over more than one volume in a multi-volume domain. A striped file distributes its storage more or less evenly over the volumes that it is striped over. The stripe width is the number of sequential file pages allocated to one volume before the next volume is used for the next stripe.

The stripe utility always sets the stripe width to 8 pages. Therefore, AdvFS allocates the first 8 pages on the first volume, the next 8 pages on the next volume, and so on until all the volumes specified are used. Then, the next 8 pages are allocated from the first volume again.

Since striped file storage is allocated in a round-robin fashion over several volumes, if AdvFS runs out of storage on any of those volumes, the file can not grow.

A striped file's extent map is really composed of several extent maps, one for each volume's stripe. Each extent map starts at page zero, so the page numbers in a striped file's extent maps are relative to the volume's stripe. Page 0 in stripe 0 is page 0 in the file. Page 0 in stripe 1 is page 8 in the file. Most of the routines that work with extent maps work with only one stripe extent map at a time and work with extent map relative pages verses file relative pages.

See Section 4.4.4 for a discussion of striped extent maps.

The following table shows how a striped file's pages are distributed across the volumes. (This information is generated automatically for striped files using the `bfaccess -x <bfap>` command in the `crash` utility). This file is 112 pages in length, but has a hole spanning pages 80 through 87. These pages would have been on stripe #2, but they are missing, and the normal round-robin assignment of pages is resumed on stripe #3.

Stripe Number	vdIndex	bsPage	vdBlk	Pages Mapped
1	2	0	66784	0 - 7
				24 - 31
				48 - 55
				72 - 79
				96 - 103
		40	-1	
2	3	0	66784	8 - 15
				32 - 39
				56 - 63
		24	-1	
		32	67168	104 - 111
		40	-1	
3	1	0	66992	16 - 23
				40 - 47
				64 - 71
		24	66928	88 - 91
		28	1047552	92 - 95
		32	-1	

5.4.1 Sparse Striped Files and Clones

When a page is COWed, `bs_cow_pg()` tries to COW up to 31 pages after the page to be modified. These pages may span several stripes. If a hole in the original file is found while looking 31 pages ahead of the page to be modified, no more pages are COWed during this call to `bs_cow_pg()`. The hole discovered in one stripe causes a permanent hole to be created in the clone in that one stripe. Even if the hole in the original spans more than one stripe, only one permanent hole is added to the clone file's extent map in this COW operation.

The permanent hole is as long in the one stripe as the hole in the original file was in the one stripe. Subsequently, if a page originally in the hole of another stripe is added, the clone gets a permanent hole in the other stripe. This retains the sparseness in the original file at the time the clone was created.

The following example illustrates the page layout (see Section 4.4) of a 90 page file with three stripes and a hole from page 35 through the end of page 75. In the illustration, a '–' means the page exists and a '.' means the storage for that page has not been allocated (it is part of a hole).

Descriptor Page	**File Page Volume 1**		**File Page Volume 2**		**File Page Volume 3**	
0	0	– – – – – – – –	8	– – – – – – – –	16	– – – – – – – –
8	24	– – – – – – – –	32	– – –	40	. . . . .
16	48	. . . . .	56	. . . . .	64	. . . . .
24	72	. . . . – – – –	80	– – – – – – – –	88	– – . . . . .

The following example illustrates the descriptors for each volume. The disk block numbers are not shown, but are represented by 'xxx' values.

Stripe #1		**Stripe #2**		**Stripe #3**	
Descriptor	**Pages Mapped**	**Descriptor**	**Pages Mapped**	**Descriptor**	**Pages Mapped**
0, xxx	0 – 7, 24 – 31	0, xxx	8 – 15, 32 – 34	0, xxx	16 – 23
16, -1		11, -1		8, -1	
28, xxx	76 – 79	24, xxx	80 – 87	24, xxx	88 – 89
32 -1		32, -1		26, -1	

If page 17 is modified in the original file, pages 17 - 34 are COWed to the clone. The hole in the original file at page 35 stops the further COWing of pages even if the next non-hole page in the file is within 32 pages of the modified page (see section 5.1.4). The hole discovered in the second stripe at descriptor pages 11 though 23 causes a permanent hole (represented as a -2) to be added to the clone's second stripe. The descriptors for the clone now appear as follows:

Stripe #1		Stripe #2		Stripe #3	
Descriptor	**Pages Mapped**	**Descriptor**	**Pages Mapped**	**Descriptor**	**Pages Mapped**
0, -1		0, -1		0, -1	
				1, xxx	17 - 23
8, xxx	24 - 31	8, xxx	32 - 34		
		11, -2		8, -1	
16, -1					
		24, -1			

5.5 Sync (update daemon), time of day stamping

To allow applications the ability to flush dirty file buffers to disk, UNIX supplies two system calls: `sync()` and `sync2()`. The first is used to synch each mounted filesystem, and the second is used to synch those filesystems that are not under smooth-sync control. (See Section 8.9 for a discussion of smooth-synch mechanism). In addition, to prevent too many buffers from being dirty, UNIX has traditionally supplied an `update` daemon that calls `sync2()` every 30 seconds.

The `sync()` and `sync2()` system calls utilize the VFS_SYNC() and VFS_SMOOTHSYNC() VFS operations. For AdvFS filesets, these are dispatched into `msfs_sync()` and `msfs_smoothsync()`.

There is a mechanism associated with the synchronization of disks that writes a timestamp to disks periodically as a sanity-check to the battery-backed clock. In AdvFS, the routine that handles this is `msfs_sync_todr()`. This is called from `msfs_smoothsync()` at `smsync_age` intervals, and from `msfs_sync()` during `sync()` and `sync2()` system calls. This routine will write a BMTR_FS_TIME record to a disk when: 1) it is unmounted; 2) it is the root fileset and the system is shutting down; or 3) it is the root fileset and an hour has expired since the last update.

5.6 System Boot

The boot process is a series of discrete steps. When the system is first turned on, the firmware is loaded and run. At this point, the system is in 'console' mode, and the administrator enters the 'boot' command to boot from the desired device. For a disk to be bootable, it must contain the appropriate information: the **bootblock**, the **primary bootstrap program**, and the **secondary bootstrap program**.

The first step of the boot process is to read the bootblock off the device; this is always at block 0 for Tru64 systems. The bootblock is really just a data record that contains the location of the primary bootstrap program on that device. This location is at logical block number (LBN) 64 on the device for AdvFS, and at LBN 1 for UFS and CDFS. The code for the primary bootstrap is then loaded into memory and executed. The purpose of this piece of code is to locate the secondary bootstrap program (`/osf_boot`) and to load it into memory. The `osf_boot` code is responsible for prompting for the kernel to be booted, then locating that kernel and loading it into memory. Once the kernel is loaded, it is executed, the various subsystems are loaded and initialized, and the system is brought up.

Several aspects of the boot process need further explanation. First, the primary bootstrap program is at a known location on the disk, so there is no additional intelligence required to find the appropriate disk block. This makes the initial code required to load the primary bootstrap fairly small. The secondary bootstrap, however, can be located anywhere on the root filesystem, and locating it requires knowledge

about the underlying filesystem. Therefore, both the primary and secondary bootstrap programs must contain some code that understands how to traverse the filesystem and locate a file. Second, the system has no way of knowing what type of filesystem will be on the device that is being booted. Determining what type of filesystem is on that disk is one of the responsibilities of the bootstrap code. Third, until the kernel is running and the filesystem code is initialized, kernel resources such as vnodes, file descriptors, the buffer cache, and typical filesystem routines, are not available for use to read the filesystem. Special routines (a mini-filesystem) are used to do this in the primary and secondary bootstrap programs, and in the early stages of kernel initialization. A bare-bones subset of filesystem routines is linked into the primary and secondary bootstrap programs to keep them as small as possible. More on these routines shortly.

Let's take a step back and see where the various pieces of the boot code come from. The bootblock and primary bootstrap program are stored in the root partition in the `/mdec` directory. The bootblock is named `xxboot.<filesystem>`, so that the bootblock record for AdvFS would be named `xxboot.advfs`. (UFS is considered a default in the naming convention, so `xxboot`, not `xxboot.ufs` is the UFS bootblock file). The primary bootstrap program is named `bootxx.<filesystem>`, with the same naming convention caveats mentioned for the bootblock. At one time a different primary bootstrap program was required for each disk type, necessitating files such as `bootrz.advfs` and `bootra.advfs`. This is no longer true, but the old file names are still maintained in the `/mdec` directory with symbolic links to the `bootxx.<filesystem>` file. The `bootxx.<filesystem>` file can be used on any type of disk to make it bootable.

How is a disk made bootable? Basically, it only takes the presence of the bootblock and the primary bootstrap program. The `/usr/sbin/disklabel` or `/usr/sbin/newfs` utilities will write the bootblock and primary bootstrap progarm to the appropriate locations on a disk to make the disk bootable. However, these utilities do nothing magical, they just copy the appropriate bootblock to LBN 0, and the appropriate primary bootstrap program to the correct LBN based on which type of filesystem is going to be supported. It is possible to make a non-bootable AdvFS disk bootable by using the `dd` utility to copy the bootblock to LBN 0, and the `bootxx.advfs` file to LBN 64.

Now, let's go back and look at how the primary bootstrap program works and how it uses the special routines that understand the AdvFS filesystem. A global structure is used throughout the primary bootstrap code to pass information between routines and to retain state information. This structure is `struct iob` for UFS and CDFS, and `struct iob2` for AdvFS. The layout of these structures is interdependent, so don't change either one without knowing what you are doing. The bootstrap code basically sets up the `iob` structure, opens the device, and then reads the superblock into memory. It then checks to see if the UFS filesystem magic number is in the superblock. If not, it calls `open_advfs()` to see if this is an AdvFS filesystem. The `open_advfs()` routine and the rest of the mini-filesystem routines used for booting are in the file `kernel/dec/sas/sys_advfs.c`. `open_advfs()` follows the following sequence of steps to find the file `/osf_boot`:

1. Open the Root tag file.
2. Find the entry for the root fileset tag file.
3. Open the root fileset tag file.
4. Find the entry for the root directory.
5. Read the primary mcell for the root directory.
6. Read and search the root directory for the `osf_boot` entry. If the path name were more complicated, the `search_dir()` routine would be called for each path component until the entire path is resolved.

7. If a match is found, read and save the contents of the first extent mcell into the iob2 structure. This will be used later when the file is read. Return success. If no match is found, return an error.

At this point, /osf_boot has been opened, but it has not been read. Later read_advfs() is called to read the file into memory. It does this by using the extent map information loaded in open_advfs() to locate the data on disk, and read the data into memory.

Once osf_boot is running, it prompts for the kernel to be loaded. This name is passed to open_advfs() and read_advfs() as before to load the kernel into memory so that it can be executed.

If you need to modify any of the code in sys_advfs.c, please note that this code can be tricky. First, many of the subroutines are recursive, so that takes special care. Second, the iob2 structure contains only 2 buffers that can be used to hold data, and each of these buffers is used for multiple types of data. For instance, the i_io_un buffer is a union that can be used for an I/O buffer, a BMT page, or a directory entry. The i_in union can be used for a tag directory page or an extent mcell. This makes the sequence in which these buffers are used extremely critical so that the data is not overlaid prematurely. Increasing the size of this structure is fraught with peril. Theoretically it is possible, but has not been done so successfully. The stack space available in the bootstrap programs is extremely limited, so you must be careful. You can not add much stack space to any of the subroutines. For instance, adding an 8K buffer to read data into will generate stack errors in the bootstrap programs.

Debugging code in the primary or secondary bootstrap programs is almost impossible. For this reason, there is ancillary code and some #defines in sys_advfs.c that allows that module to be built as a standalone utility for testing. See the instructions at the top of the file for doing this. Depending on which values are #defined, this utility will open the specified file (using open_advfs()), and then read the file. If 'CATIT' is defined, then the file is read (using read_advfs()) and the contents are dumped to stdout. If 'SHOWDIR' is defined, then the file is assumed to be a directory and the directory entries are read and displayed.

5.7 System Shutdown

When the system is shutting down, it calls a routine named, oddly enough, boot(). Boot() does most of the work responsible for cleaning up before the system goes away. One of the things done in this routine is to set the global variable 'advfs_shutting_down' which is done specifically for AdvFS. There is also a 'shutting_down' global variable used by the device drivers, and AdvFS used to check that, but it was slated for removal at one time, so AdvFS added its own file system-specific variable. 'Shutting_down' was never removed after all.

In boot(), mntflushbuf() is called for all mount points. In this routine, if the fileset is AdvFS, msfs_mntflushbuf() is called to flush any outstanding dirty buffers to disk asynchronously. After mntflushbuf() has been called on all mount point from the boot() routine, mntbusybuf() is called on each mount point to count the number of outstanding dirty buffers that we are waiting to finish flushing to disk. We loop in boot(), calling mntbusybuf() until the number of unflushed buffers reaches zero.

When shutting down, the current timestamp is also written to the BMTR_FS_TIME record on the root filesystem's disk. See Section 5.5 for a discussion of writing timestamp records to disk.

Chapter 6: Special Files

Directories and the fragment bitfile can be thought of as 'special' files because of the way they are handled. Directories are special because, although they are not metadata files, changes to their layout must be logged to ensure atomicity of file creation and deletion. Similarly, the fragment bitfile is a reserved file that is not visible to the logical file hierarchy, and also has its changes logged. However, it contains mostly user data that is typically not logged. The details of these files are explored in this chapter.

6.1 Directories

AdvFS directories are interesting because: 1) they are user-visible files but are not manipulated directly by applications; 2) they provide the interface between the logical file hierarchy layer and the physical storage layer; 3) they must be protected by the logging subsystem to ensure that file creation and deletion are atomic operations, and 4) they may or may not be indexed.

Historically, AdvFS directories are similar to UFS directories. Directories are a series of records that contain a file name plus some information to associate (link) the name with the location of its data. In UFS, that information link is an **inode** number, but in AdvFS, the link is a tag. The relationship between the file name and tag is what provides the interface between the logical file hierarchy layer (name) and the physical storage layer (tag) (see section 1.2). One of the drawbacks to the historical directory layout was that as the directory got larger, the time to search the directory increased as well. To solve this problem, indexed directories were introduced into AdvFS in Tru64 UNIX Version 5.0. If a directory is indexed, its data is actually spread across two distinct files: the traditional directory file and the index. This allows the flexibility to use directories either with or without the index file. It also allows an older kernel that does not know about indexed directories to read a directory with an index (although the benefits of the index are not realized).

6.1.1 Non-indexed Directories

In this section we will explore the traditional part of the directory from an AdvFS perspective. When the term 'directory' is used in this section, it refers to the file only, not the index, unless explicitly stated otherwise.

AdvFS directories can be viewed as a series of 512-byte logical blocks. This size is rooted in the fact that the underlying sector size for disk devices supported by Tru64 UNIX is 512 bytes. This means that the 512-byte blocks will be written atomically: either completely updated or not updated at all. To take advantage of this, the AdvFS designers laid out the directory as a series of 512-byte sections. No data written within any of these sections can span into another section.

Superimposed on top of this 512-byte scheme is the logical layout of each directory page. Each 8K page contains a sequence of variable-length directory entry records, one for each file in the directory. Each entry is made up of a fixed-length header (`struct dir_header`), a variable-length name field (`char fs_dir_name_string[]`), and a fixed-length trailer (`struct dir_ent_end`). (See Table 6.1 for a sample directory layout). The entry header contains the tag for the file (`fs_dir_bs_tag_num`), the size of the directory entry record (`fs_dir_size`), and the length in bytes of the filename (`fs_dir_namecount`). The size of the directory entry is useful for allowing advancement to the next entry in the page. The trailer is tacked on at the end of the filename, and is included in the size of

fs_dir_size. The variable-length name field is always padded so that the trailer will begin on the first 4-byte word boundary immediately following the filename. If the variable-length name field is already 4-byte word aligned, a word (4 bytes) of nulls is added; the name field must be terminated by at least one null. The trailer contains the full tag (struct bfTag) for the file.

If a file is deleted, the fs_dir_bs_tag_num field in the dir_header contains a zero value. It appears that the tag.num value is repeated in the header and trailer because the original designers wanted the AdvFS entry header to have the same structure and size as the UFS entry header.

At the end of each directory page is a fixed-length trailer structure (struct dir_rec) that contains three values: the offset to the start of the last directory entry on the page, the offset of the largest free space in the page, and the page type. The values in the page trailer appear to be of limited usefulness. The value for largestFreeSpace is always set to zero, and the value for pageType is always one. The value for lastEntry_offset contains a pointer to the last entry in the page, whether it is a valid entry or a free slot.

Table 6.1 illustrates the data in a simple subdirectory page. The first two entries are for the '.' and '..' files which are in all directories. Three files, file1, bigfilename, and bigfile2 were created, and then the file bigfilename was deleted. We can tell that this file was deleted because its tag number in the entry header has the value zero, even though all the other information for the entry remains intact. If another file is created in this directory with a name the same length or shorter than bigfilename, it will be written over this data. Note that the entire page after the valid filenames (at offset 0x78) contains a series of segments all with tag values of zero. These are empty slots and will be used as needed. The first slot contains 392 free bytes of free space, while all others are empty 512-byte segments.

Table 6.1 Layout of Entries in a Sample AdvFS Subdirectory

Offset into the page	Tag #	Entry Length	Name Length	Filename	Tag.Sequence
0	7	0x14=20	1	.	7.8001
0x14	2	0x14=20	2	..	2.8001
0x28	8	0x18=22	5	file1	8.8001
0x40	0	0x1c=28	11	bigfilename	9.8001
0x5c	10	0x1c=28	8	bigfile2	10.8001
0x78	0	0x188=392	0	0	0
0x200	0	0x200=512	0	0	0
0x400	0	0x200=512	0	0	0
…					
0x1c00	0	0x200=512	0	0	0
0x1e00	0	0x200=512	0	0	0
dir_rec.lastEntry_offset = 0x1e00					
dir_rec.largestFreeSpace = 0					
dir_rec.pageType = 1					

If a file is deleted and there is an adjacent free entry within its 512-byte block, the free areas are coalesced. In the above example, if file1 is deleted, then the tag number at offset 0x28 is set to zero, and the entry length for that entry is set to 0x34 because its entry of length 0x18 is combined with the following entry of length 0x1c. If there had been a free entry at offset 0x14, then that entry would have

been coalesced as well. Coalescing (also called **glomming**), however, will not occur across 512-byte block boundaries.

There are three principle operations that can occur to a directory: create, delete, and lookup. Secondarily, storage can be added or removed from a directory. When creating a new entry, the inserting thread must first verify that it is not inserting a duplicate entry into the directory. This means that after some synchronization mechanism is seized, it must either search the entire directory (in the case of non-indexed directories) or search the b-tree (in the case of indexed directories) to ensure that the entry does not already exist. Timestamps can be used to avoid unnecessarily searching the directory, and this will be explained shortly. Directory entry removal is somewhat simpler than insertion because there is no need to check all entries. The only verification required is that the entry to be removed is actually associated with the file being deleted. Since entries are never moved in the directory, once it is located, its position is known until it is removed. Because lookup and removal may occur on different calls to the kernel, timestamps may again be used to avoid unnecessary re-searching of the directory when removal starts. The only synchronization required in this path is to ensure that one thread is not trying to remove an entry that has already been removed by another thread, and to guarantee that the glomming of entries cannot cause another thread to lose its way through the directory.

All directory operations must be synchronized to assure proper operation when multiple threads are manipulating the same directory concurrently. The directory's file lock (`fsContext.file_lock`) is the primary synchronization mechanism among these operations. During lookup operations, the file lock is seized for shared access. This allows several threads to search the directory concurrently, while blocking modifications to the directory during the lookup process. The real concern for lookup is that no entry lengths are being modified in a non-atomic fashion by an insert or delete operation such that the searching thread may not find the next entry correctly. Both insert and delete operations seize the file lock for exclusive access, preventing all racing operations. Note that this lock is file wide, so insertions and deletions effectively single-thread operations for a given directory while that modification is in flight. A file rename is essentially a deletion followed by an insertion, so it also seizes the file lock for exclusive access.

Storage addition and deletion to all AdvFS files are done with the file lock held for exclusive access, and this convention is also followed for directories.

Timestamps can be used to avoid unnecessary directory searches. There are two timestamps that are maintained, one in the directory's `fsContext` structure, and one in each thread's `nameidata` structure. The `nameidata` is local to the current thread, and is not modified by the actions of other processes or threads. During lookup (`msfs_lookup`), the `nameidata` timestamp is set to the timestamp value in the file's context structure. When a new directory entry is inserted (`fs_create_file`, `msfs_link`, `msfs_rename`), the timestamp in the file's context structure is incremented to indicate that the directory has been modified. This field is also incremented in the entry removal path (`remove_dir_ent`), to indicate that the directory has been modified.

Now, let's see how this all goes together in actual system operations. During file creation, an application calls `open()`, which eventually calls `vn_open()`, `namei()` and `msfs_lookup()`. During lookup, the fact that the name does not already exist is verified, and a location where the entry can be inserted is saved in the `nameidata` structure. If the lookup finds no existing entry, then `vn_open()` calls `msfs_create()` and `fs_create_file()` where the directory entry is created. The file lock is not held between the `msfs_lookup()` and `msfs_create()` calls, so it is possible for a file of the same name to be created between the lookup and the create. Since a file create must be absolutely certain that the file name doesn't already exist, it could search the directory again, but this would be inefficient. To avoid this double search, the timestamps are compared in

fs_create_file(). Remember that the nameidata timestamp was set to the file's timestamp during the lookup. If, after seizing the file lock in the create path, the timestamps are still the same, then the thread knows that there was no intervening directory modification between the lookup and the create call, so it is safe to go ahead and insert the entry. If the timestamp has changed, however, the thread doing the create must re-search the directory to: 1) verify that the file hasn't already been inserted, and 2) find a location at which the entry can be inserted.

The removal path is similar in that msfs_lookup() is called to ensure that the file exists. If it does, it saves the location of the directory entry in the thread's local nameidata structure. When msfs_remove() or msfs_rmdir() is called, the location of the entry is known unless a racing thread already removed it. This means that the remove code must, if the timestamp has changed since the lookup, verify that the entry pointed to by the nameidata structure belongs to the file intending to be removed. If so, it is removed. If not, it returns an error indicating that the file no longer exists.

6.1.2 Indexed Directories

Indexed directories were introduced into AdvFS in Tru64 UNIX Version 5.0 with the goals of: 1) improving lookup performance, 2) scaling lookup performance with the number of entries in the directory, 3) being invisible to applications, and 4) maintaining the old directory format for backward compatibility. If a directory is indexed, its data is actually spread across two distinct files: the traditional directory file and the index. This allows the flexibility to use directories with or without the index file. It also allows an older kernel that does not know about indexed directories to read a directory with an index (although the benefits of the index are not realized).

As names of files are added to a directory, more than one 8k page will eventually be required to hold all the entries. An index is automatically built for a directory when it expands from one to two pages in size. The reason for waiting until a directory reaches 2 pages in length is that the search time for a one-page directory is minimal, and the added overhead of maintaining the index for a small directory was determined to be excessive. Once created, however, the index remains with the directory until the directory is deleted, even if the directory is truncated to only one page.

You can tell if a directory has an index by running:

```
showfile <directory>
```

If this is a domain that supports directory indexes (DVN 4 or later) and the directory has an index, then the word '(index)' will follow the directory name under the heading 'File'. If you would like to display the statistics for the index instead of the traditional portion of the directory, enter:

```
showfile -i <directory>
```

In this case, the word 'index' is displayed under the File heading, and the name of the directory is appended in parentheses.

The index file is organized as a B+ Tree. This structure allows a quick search mechanism using a tree structure where each node has many entries, and the tree can grow and shrink as needed using generalized splitting and pruning routines. In a B+ tree each non-leaf node contains a series of entries that each point to a child node whose elements are less than or equal to its own value. The leaf nodes contain the actual data elements.

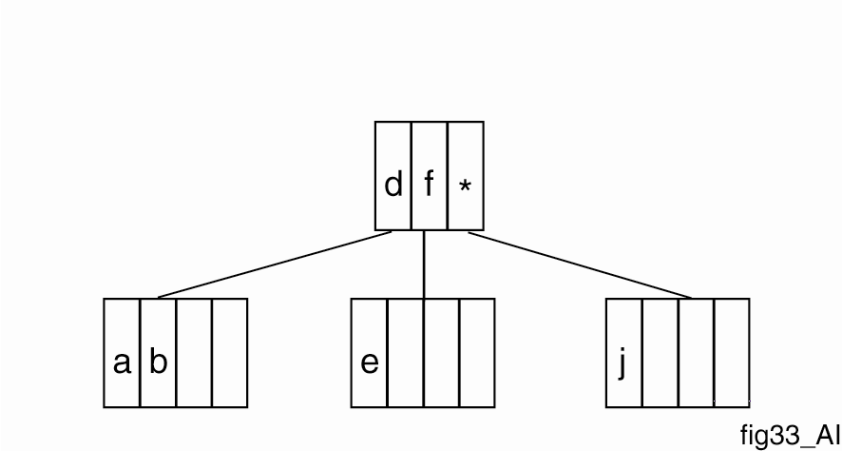


Figure 23: B+ Tree Nodes

Each index actually contains two B+ Tree structures: one for the filenames and one for the free space within the directory. The index file metadata has a mcell record that contains the page numbers to locate the root nodes for each of the two B+ trees plus the depth of each of the trees (# of levels). The filename tree maps the location of the filenames in the directory. Each filename is hashed into a 56-bit value, and each hash value is paired with the offset in the directory where the filename entry resides. Collisions in the hashed values can occur and are handled in the code. The free-space B+ tree exists to facilitate retrieval of free space in the directory by mapping deleted entries. These locations are sorted and stored in the tree by file offset within the directory. Unlike the directory file, which coalesces unused areas within 512-byte regions, the free space within the free space tree is coalesced into the largest possible chunks. The free space is given out by first fit in the directory, which encourages directory truncation.

Each index node is an 8K page containing a header and up to 510 (IDX_MAX_ELEMENTS) entries. Each node can be described by the `struct idxNode` which contains the number of elements currently in the node, the page number of the right sibling, and an array of `struct idxNodeEntry` structures to hold all of the data elements. Each of these elements contains a search key plus a union of offsets, page numbers, or free space size depending on the type of node.

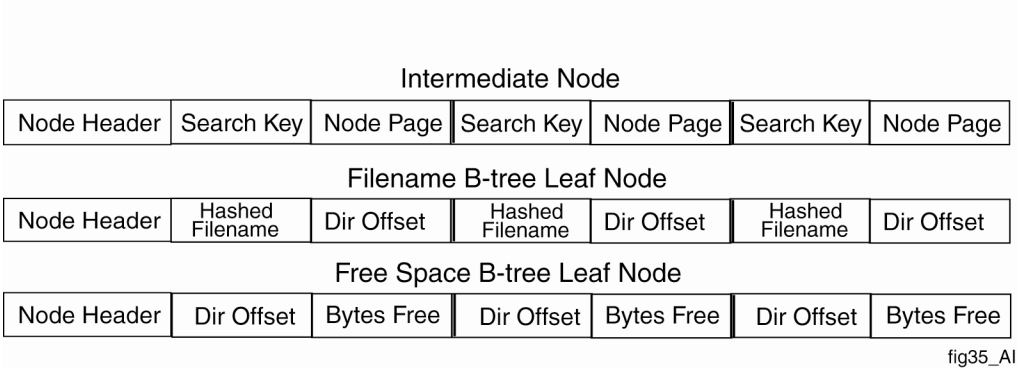


Figure 24: B+ Tree Index Node Comparison

Insertion of entries into a B+ tree once a node is full can cause the node to be split. When the node is split, half the entries are placed into each of the 2 new nodes. The addition of a new node will force a new entry to be inserted into the parent node that points to the newly-split nodes. If that parent node is already full, it will also split. By this mechanism, it is possible that the split can propagate up the tree. A new level can be produced if the root splits and a new root node is added. See the figure below.

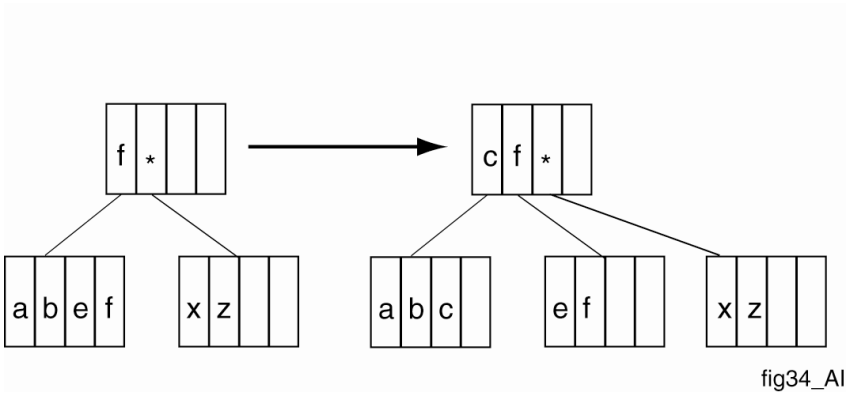


Figure 25: Splitting a B+ Tree Node

Pruning of a B+ Tree takes place when a node becomes ¾ empty (less than 128 entries in one node). Nodes on the same level are merged by moving the elements from the right sibling into the left. If the two nodes cannot be compressed into one node, their elements will be evenly distributed between the two nodes. This pruning can occur at any level of the tree. Levels are collapsed when only one node exists at that level.

Based on all this, we can see that file creation causes a new entry to be inserted into the directory. This will cause AdvFS to obtain the space from the free space B+ tree, insert the hash value and offset into the filename B+ tree, insert the entry into the directory, and initiate file truncation if the truncation flag is set (see section 6.1.3 on Directory Truncation). Similarly, removing a file will cause the directory entry to be deleted; this involves removing the hash value from the filename B+ tree, inserting the space into the free space B+ tree, zeroing the tag in the directory entry, and setting a truncation flag if the last page in the directory file is now all free space.

As mentioned previously, indexed directories are actually treated as two files internally: the directory file and the index file. When looking at the `bfAccess` structures for directories, you can tell if a directory is indexed if the `bfAccessT.idx_params` field is neither `NULL` nor `-1`; it should look like a valid pointer to a structure of type `bsIdxRecT`. If the `bsIdxRecT.flags` field has the IDX_INDEX_FILE bit set, then this is the `bfAccess` structure for the index file. Otherwise, this is the `bfAccess` structure for the directory file itself.

A note on the values of `bfAccessT.idx_params`. This field is initialized to NULL when the `bfAccess` structure is initialized or recycled. This value indicates that the file's `mcell` should be read to see if an index file is associated with this directory. A value of `-1` indicates that the directory does not have an index and its `mcell` does not need to be checked. This saves a read every time the directory is opened.

To decipher the contents of the structure at `bfAccessT.idx_params`, cast it as a `(bsDirIdxRecT *)` if the `bfAccess` is for the directory file, but cast it as a `(bsIdxRecT *)` if the `bfAccess` is for the index file. The following are examples using the `crash` utility:

Given the following data:

```
directory bfap =                0xffffc00c3816c08
directory bfap->idx_params =    0xfffffc000a2737e8
index bfap =                    0xfffffc00c3816008
index bfap->idx_params =        0xfffffc0056abc948
```

Then:

```
crash> * bsDirIdxRecT 0xfffffc000a2737e8

struct {
   flags = 0x0                  <== IDX_INDEX_FILE is not set
   idx_bfap = (nil)
   index_tag = struct {
      num = 0x3007              <==  tag.num for index file
      seq = 0x8001              <==  tag.seq for index file
   }
}
```

This is the data for the directory file since the `IDX_INDEX_FILE` bit is clear. It contains the tag for the index file, and, if the index file were open, a pointer to the index file's `bfAccess` structure. The value of `bfap->idx_params->idx_bfap` is NULL in this case because the index has been closed. This indicates to future threads that the index must be re-opened before use. When the index file is opened, the value of `idx_bfap` is set to point to the `bfAccess` structure for the index.

The following is the data for the index file, as indicated by the `IDX_INDEX_FILE` bit in the flags field. This contains information for finding data inside the index file, as well as a pointer back to the directory's `bfAccess` structure.

```
crash> * bsIdxRecT 0xfffffc0056abc948

struct {
   flags = 0x2                  <== IDX_INDEX_FILE is set
   bmt = struct {
      fname_page = 0x3          <== page with root node for name tree
      ffree_page = 0x1          <== page with root node for free space tree
      fname_levels = 0x1        <== # of levels in name tree
      ffree_levels = 0x0        <== # of levels in free space tree
   }
   dir_bfap = 0xfffffc00c3816c08 <== bfap for directory.
}
```

Here are some debugging hints if you need to wander into this code. There are always at least two pages in an index, the two B+ tree root nodes. Partial dumps will not contain the contents of the index nodes since these are in UBC pages. If you need access to the index file contents, use `showfile` and `disphex` to dump the index pages.

Consider the following example in which we want to see the directory and index information for a subdirectory called `/fset1/test40` which contains 511 files.

Step 1: Use `showfile` to get information about the directory file.

```
prompt> showfile /fset1/test40

        Id  Vol  PgSz  Pages  XtntType  Segs  SegSz  I/O   Perf  File
2e.8001    2    16     2      simple     **    **      ftx   33%  test40 (index)
```

This shows us that there is an index associated with this directory. Before looking at the index data, let's see the information inside the directory itself.

Step 2: To dump the directory pages for this file, use the `disphex` utility. This utility and its source code can be found at `soak2:/usr/specs/filesystem/tests/general`.

```
prompt> disphex -D /fset1/test40
```

This will read the directory one page at a time starting at page 0, interpret the directory entries, and dump them to the screen in an abbreviated format:

```
Pg Offset  Tag   Seq   Entry Len      Name
---------- ----- -----  ---------  ------------------
   0        21  8001      20     .
  20         2  8001      20     ..
  40        97  8001      28     Test_file_0
  68       165  8001      28     Test_file_1
  96       238  8001      28     Test_file_2
 124       288  8001      28     Test_file_3
...
8032     12398  8001      32     Test_file_251
8064     12445  8001      32     Test_file_252
8096     12496  8001      32     Test_file_253
8128     12538  8001      32     Test_file_254
8160                       32     --- Empty slot ---
Block = 0
Enter: block #, [c #], d, q, [o #], L, l, n, N, p, P, ?
```

These are the entries for page 0. Hitting return will display the entries for page 1. At the prompt you can enter the next page of data that you want to display. For instance, typing ‘3’ will display the data for page 3 in the file. Type 'q' to quit.

Step 3: Using the `-i` option of `showfile`, see the information for the index.

```
prompt> showfile -i /fset1/test40

        Id  Vol  PgSz  Pages  XtntType  Segs  SegSz  I/O   Perf  File
3007.8001    2    16     4      simple     **    **      ftx   33%  index (test40)
```

From this we get the tag (0x3007) and size (4 pages) of the index file.

Step 4: We can use `disphex` and the `.tags` interface to dump the pages of the index. We need the decimal value of the tag, which is 12295 (0x3007).

```
prompt> disphex /fset1/.tags/12295

    0  ff000000 00000000 01000000 02000000   |................|
   10  00000000 00000000 00000000 00000000   |................|
   20  002e0000 00000000 00000000 00000000   |................|
   30  000e0300 00000000 14000000 00000000   |................|
   40  00202bd0 a5ca5a00 28000000 00000000   |. +...Z.(.......|
   50  00212bd0 a5ca5a00 44000000 00000000   |.!+...Z.D.......|
   60  00222bd0 a5ca5a00 60000000 00000000   |."+...Z.`.......|
   70  00232bd0 a5ca5a00 7c000000 00000000   |.#+...Z.|.......|
   80  00242bd0 a5ca5a00 98000000 00000000   |.$+...Z.........|
   90  00252bd0 a5ca5a00 b4000000 00000000   |.%+...Z.........|
   a0  00262bd0 a5ca5a00 d0000000 00000000   |.&+...Z.........|
   b0  00272bd0 a5ca5a00 ec000000 00000000   |.'+...Z.........|
   c0  00282bd0 a5ca5a00 08010000 00000000   |.(+...Z.........|
   d0  00292bd0 a5ca5a00 24010000 00000000   |.)+...Z.$.......|
   e0  0040b202 5daaac05 40010000 00000000   |.@..]...@.......|
   f0  0041b202 5daaac05 60010000 00000000   |.A..]...`.......|
   Block = 0
   Enter: block #, [c #], d, q, [o #], L, l, n, N, p, P, ?
```

This is the raw data for page 0 of the index file, but we really don't know what kind of information this represents (file name tree or free space tree; root node or leaf node).

Step 5: Use the `crash` utility to search for the access structure for this directory. If you have used the `showfile` commands above, this subdirectory has been accessed recently, so its `bfAccess` structure should still be around on the free list.

```
crash> bfaccess -s 12295

LIST      bfAccessT           bfSetT           Tag
------  ----------------  ----------------  -------------
Free   fffffc00c3816008  fffffc00e3e4e788      3007.8001

crash> bfaccess fffffc00c3816008
    struct {
       ...
       idx_params = 0xfffffc0056abc948
    }

crash> * bsIdxRecT 0xfffffc0056abc948
    struct {
       flags = 0x2
       bmt = struct {
            fname_page = 0x3
            ffree_page = 0x1
            fname_levels = 0x1
            ffree_levels = 0x0
       }
       dir_bfap = 0xfffffc00c3816c08
    }
```

From this data we can see that the file name btree root node is page 3 of the file, and the free space root node is page 1. Since there are 4 pages in this file (from the `showfile` in step 3), we can infer that pages 0 and 2 contain the file name leaf nodes. This agrees with the fact that the filename tree has one level (root level plus one level for leaf nodes).

Step 6: Now use `disphex` with the `-I` option to format the data on the page as if it were an index. Currently this utility does not know the difference between leaf and non-leaf nodes, nor between the file name and free space tree, so all the data is presented in raw form. But not quite so raw as the unformatted data.

The following example dumps the root node for the file name tree in which there are 2 leaf nodes, at pages 0 and 2.

```
prompt> disphex -I /fset1/.tags/12295
Page in file:     3
Elements in page: 2
Left page:        1
Right page:      -1
Max elements/node:510

non-leaf nodes:        Key = Search Key  Location = Node Page
file name leaf nodes:  Key = Hashed Name  Location = Dir Offset
free space leaf nodes: Key = Dir Offset  Location = Bytes Free

        Key        Location
---------------    ---------
     723878400            0
            12            2
```

6.1.3 Directory Truncation

Directories can be truncated whenever all the entries are removed from a trailing page, but never from an interior page. This means that if a directory has 5 pages of entries, and all of the entries are removed from page 4, page 4 is eligible to be truncated and have its storage returned to the domain (page 0 is the first page). If all the entries are removed from page 2, that page is not eligible to be truncated since there are still entries in page 3. If the entries from pages 3 and 4 are subsequently removed, then pages 2, 3 and 4 will be truncated.

Curiously, directories are not truncated on entry deletion, but on entry insertion. This is because of the amount of overhead required to determine the truncation point. Since the directory is scanned and the last known entry is determined on insertion, but not during removal, insertion was chosen as a more-efficient operation to trigger storage truncation. Another reason that truncation is done at insert time is to avoid trying to truncate a page that has pinned records. As in all storage deallocation, the truncation is done in two phases. The first phase is done in the context of the thread doing the directory insertion. This thread then passes a message to the `fs_cleanup_thread()` which completes the storage deallocation.

An indexed directory does not have its index file removed even when the directory itself is truncated down to a single page. Once the directory has been indexed (as it expands beyond the first page), it remains an indexed directory forever.

6.1.4 Trashcans

Trashcans are a mechanism that system administrators can use to prevent unwanted deletions of files in the AdvFS environment. In a typical UFS filesystem, and in AdvFS directories without trashcans, there is no way to retrieve a file (undelete it) after it has been removed. With trashcans, however, the file is never deleted, it is simply moved to the trashcan directory. The deletion code checks for a trashcan directory, and if one exists, it merely renames the file instead of a deleting it. If the trashcan is on a different domain, the file is copied to the trashcan directory and then the old file is removed.

Trashcans are established using the `mktrashcan <trashcan> <directory>` command. See the man page for more information about adding, removing, and displaying trashcans.

6.2 Fragment Bitfile

6.2.1 File Fragments and Fragging Concepts

Storage allocation in AdvFS is done in 8 KB page units. For files that are not an even multiple of 8 KB in length, only part of the last page is used to store data. For file systems with many small files, this can result in inefficient storage utilization.

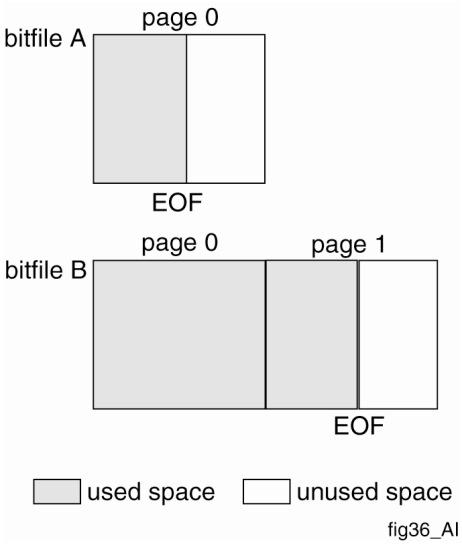


Figure 26: Wasted space in a non-fragged file

To minimize wasting of disk storage, AdvFS stores the data that is beyond the last full page of a file in a **fragment**. (Storing this remainder of data in a fragment is typically called **fragging** the file). Fragments, by definition, are sized in 1k increments from 1K to 7K. Thus, if a file is 13.5K in length, the 5.5K in page 1 will be stored in a 6K fragment. This scheme guarantees that the wasted space at the end of a file is always less than 1K. This is illustrated in the figure below:

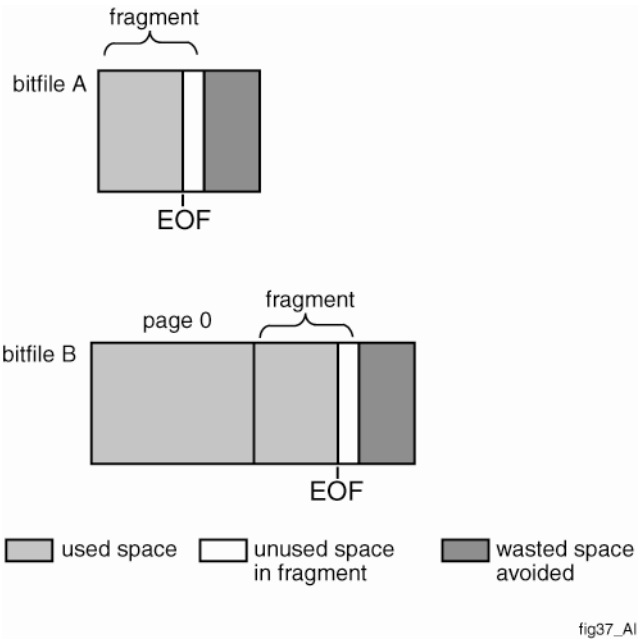


Figure 27: Wasted space avoided with fragging

In bitfile A in this figure, assume that the file size is 5632 bytes, or 5.5K. This data will all fit inside a 6K fragment with only 512 wasted (unused) bytes. If this file were not fragged, then the wasted space would be 2.5K of a full 8K page. Therefore, AdvFS realizes a savings of 2K, and the wasted space is only 20% of what it would be if the file were not fragged.

Because a domain with many small files of uneven size will exhibit a greater waste of storage space than a domain with fewer larger files, and because fragging a file adds processing overhead, AdvFS only attempts to frag relatively small files. Internally, a file will be considered for fragging when the wasted space after the end-of-file will be greater than 5% (#defined by `FRAG_PERCENT`) of the total file size. Because of this calculation, any file 160K in size or larger will never be fragged.

A file is fragged on the last close of the file so long as none of the following conditions are true:

- The file is a reserved file.
- The file is a quota file.
- The file is empty.
- The file is opened for direct I/O.
- The file is marked for using persistent atomic-write data logging (see Section 8.2).
- The fileset is marked for 'no fragging' (see below)
- The fileset is mounted read-only
- The amount of unused storage in the last page of the file is less than 5% of the size of the file.

On the last close, if none of the above conditions are met, the last page of the file is moved into the **fragment bitfile**, which is explained in the next section. This is done by allocating a fragment from the

fragment bitfile, copying the data into the fragment, deallocating the final page of the data file, and storing the information necessary to find the fragged data in the file's `bfAccess.fragId` structure.

For existing files with fragments, when a write exceeds the fragment size, a new page is allocated to the file, the fragment is copied to the new page, and the fragment is deallocated.

Let's briefly discuss two of the above reasons why files are not fragged. First, files opened for direct I/O are not fragged on close in anticipation that it will be opened for direct I/O again. When the data in the frag is modified in a direct I/O file, the frag must be brought into the normal extent maps before the I/O can be done. This is added overhead that is usually not wanted in a direct I/O application, so the fragging of the file is inhibited to prevent this unneeded overhead.

Another reason that a file might not be fragged is because the fileset has been marked to prevent it. Applications like web servers that generate many small files may rather trade off the additional storage space required to eliminate the overhead necessary to frag and unfrag the files as they are closed and reopened. In this case, the application can keep all of the files that are not to be fragged in a fileset that is marked to prevent fragging. This is done by using one of the following commands:

```
mkfset –o nofrag <domain> <fileset>

chfsets –o nofrag <fileset>
```

The `showfsets` utility will indicate if a fileset is marked for fragging by displaying a line of the type:

```
Fragging:  On   or

Fragging: Off
```

6.2.2 Fragment Bitfile Layout

File fragments are stored in the fragment bitfile, usually called the **frag file**. There is one frag file per bitfile set, and it always has a tag of 1. The frag file can be viewed in two different ways. One way is as an array of 1K slots. This means that each page of the file has 8 slots. The page number of the frag can be found by dividing the slot address by 8, and the remainder corresponds to the slot inside that page. The `bfAccess.fragId.frag` field is an index into this virtual array. A frag value of 0x12a77 refers to a frag starting in page 9550, slot 7 of the frag file.

Another way to view the frag file is as a collection of **fragment groups**, where each group contains a header and an array of fragments of a uniform size. A fragment group is defined by the size of fragments it contains: 1K, 2K, 3K, ... , 7K. Each group consists of sixteen 8K pages. Groups are addressed by the page number of the first page in the group. Groups of the same type (that is, fragment size) are linked together if they contain at least one free fragment (See figure below). Groups with no in-use fragments are kept in a special list of free groups. When a group is allocated, it is initialized and assigned a type. A group that has no free fragments is not on the linked list for its fragment type; if fragments are subsequently freed, then the group is placed back on the linked list.

Once the number of free groups on the free list exceeds a maximum threshold, the number of free groups is reduced to a minimum threshold by deallocating the corresponding pages from the frag file. Group deallocation is done by a global kernel thread. *frag_group_dealloc()* sends a message to *bs_fragbf_thread()* to do the deallocation. A separate thread is necessary because *bs_frag_dealloc()* runs within a transaction, and the deallocation of fragment groups requires starting a new root transaction inside *del_dealloc_stg()*. Since root transactions can not be nested, a separate thread is needed. There is one *bs_fragbf_thread()* per system. Curiously, the free list headers for the frag file are maintained in the

BSR_BFS_ATTR record in the mcell chain of the bitfile set's tag directory. This appears to have been done as a matter of convenience rather than necessity. The other attributes of the bitfile set are in the tag directory's metadata, so this attribute is there as well.

Prior to version 5.1B, the minimum and maximum threshold values were configurable via `sysconfig`. The variables were `AdvfsMinFragGroups` and `AdvfsMaxFragGrps`, and defaulted to 16 and 48, respectively. As of 5.1B, these default values are still used, but they are no longer configurable, even via `dbx`.

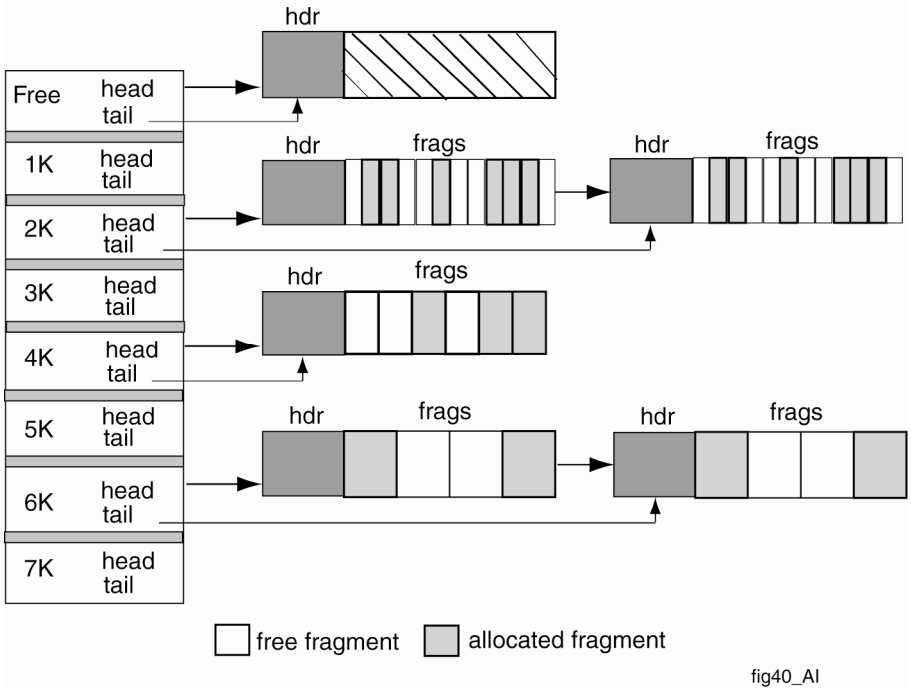


Figure 28: Logical structure of the fragment bitfile

A third way to view the frag file is as an array of 16-page groups. The figure below shows the layout of the frag file in the previous figure. Note that two groups have been completely deallocated; these are sparse holes in the file:

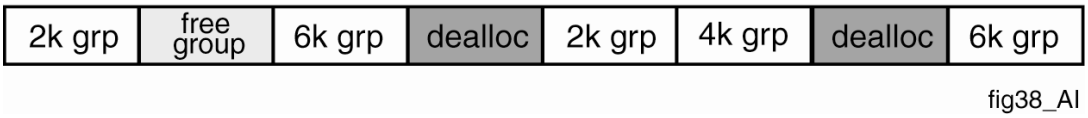


Figure 29: Frag file in 16-page groups

To summarize, the frag file can be addressed in several ways:

- By slot number (as an array of 1K slots)
- By page number

- By group (which is by the page number of the first page in the group)

Fragments other than 1K and 2K sizes may span underlying frag file pages. By definition all 1K fragments fit inside an 8K page. The first page of the 2K group is coerced such that the first fragment starts 2K into the page, guaranteeing that none of the 2K fragments will span the underlying pages. All other fragment types may have fragments that span the underlying pages. There is logic throughout the fragment code that detects spanning and pins or refs both pages as necessary.

All changes to the fragment bitfile, including data modifications within allocated fragments, are done under transaction control. This seems rather incongruous since the data saved in the frag file is user data, and user data is not typically logged. In addition, the logging of frag data adds significant traffic to the log file. However, transactions serve to maintain the consistency of the frag file and to prevent **object reuse**. There was an experiment in the v51 era in which an AdvFS engineer stopped logging much of the frag file traffic, and object reuse became significant, particularly in the clustered environment. This may be an area worthy of additional study and experimentation.

Chapter 7: Buffer Cache

7.1 Overview

Once upon a time in the predawn of computing when silicon wafers were evolving in a methane-rich atmosphere, reading a file meant retrieving the data from disk and placing it into the application's memory. Conversely, writing a file meant transferring the data from a user's buffer and placing it onto disk. The write was not considered complete until the data was safely on disk. However, primordial programmers quickly realized that transfers to disk were orders of magnitude slower than transfers of data to memory. They also realized that the data for a given file page may be accessed many times, and reading and writing to disk each time was not very efficient. It was speculated that a copy of the data for each file page could be kept (**cached**) in memory, and each read would be satisfied from this cached page. Further, any writes to the page could be accomplished by updating the cached page, temporarily avoiding the more-costly update to disk. Thus was born the idea of the filesystem's buffer cache.

The AdvFS buffer cache has had two different implementations. Prior to Tru64 Unix Version 5.1, AdvFS wired a predetermined number of virtual memory pages and maintained absolute control of these pages for use as its buffer cache. The number of pages set aside for the cache was determined by the configurable variable `AdvfsMaxCachePercent`, which could range from 1 to 30% of total system memory (default value was 7%). One drawback with this scheme was that if the system administrator guessed wrong when setting the value for this variable, the cache could be either woefully undersized for the amount of file activity, or it could end up consuming memory resources that were not needed for file activity, but were needed for other purposes.

As of Version 5.1, AdvFS makes use of the **Unified Buffer Cache (UBC)** for its cached pages. The AdvFS buffer cache can be considered a wrapper around the memory resources and routines offered by the UBC. The term 'buffer cache' will be used throughout this chapter to mean the AdvFS buffer caching routines and mechanisms plus their underlying UBC resources; explicit references to the UBC and its routines will be made where applicable.

The UBC is part of the Virtual Memory (VM) subsystem. The UBC monitors the number of memory pages needed by each filesystem and dynamically allocates the number of pages as demands change. Conversely, as other memory demands on the system increase (e.g. processes start, memory is allocated), the UBC may be required to take away some pages in use by the file systems to give back to VM for such demands. This give-and-take allows the system to self-tune and allows memory resources to be used where most needed at any given time.

AdvFS user and metadata pages are typically stored in the buffer cache. Several operations that bypass the caching of data are direct I/O requests, AdvFS raw I/O requests, and certain domain-related metadata initializations that use a raw I/O interface. The UBC manages the recycling of clean pages either for use as a new UBC page or to give it back to the Virtual Memory subsystem. In addition, the UBC may request AdvFS to flush dirty pages to disk so that they can be reclaimed as clean pages by the UBC. This will be discussed in more depth shortly.

The following sections will discuss the AdvFS buffer cache, its structures, and functions primarily from Version 5.1A perspectives. (There are some significant changes in Version 5.1B that complicate several of the code paths presented here. We hope to update this chapter to reflect those changes shortly).

7.2 In-Memory Structures

Every open file in AdvFS has a `vnode`, which is associated with both a `vm_ubc_object` and a `bfAccess` structure. (Reserved AdvFS files don't have a `vnode`, but that isn't important in this discussion). Each `vm_ubc_object` has a list of UBC pages (`struct vm_page`) that represent the on-disk pages for the file. In parallel with this, AdvFS maintains a `bsBuf` structure for each UBC page in the cache (see figure below). For AdvFS only, the UBC page's `vm_page.pg_private` field contains a pointer to its associated `bsBuf` structure. Additionally, the `bsBuf.vmpage` field contains a pointer to its associated `vm_page` structure (this field was called the `bsBuf.vmpl.pl[0]` field in 5.1A). `vm_page` structures are linked together with the `pg_onext` and `pg_oprev` pointers.

The `bsBuf` is always associated with an 8k page of memory. The UBC `page_t` structure has traditionally been associated with an 8k page also, but may be associated with a Big Page of larger size as of version 5.1B. (More on Big Pages in section 7.8.) Also notice in the figure that AdvFS maintains a list of dirty buffers in the `bfAccess` structure, but does not maintain a list of clean buffers. This is because there is no need for AdvFS to monitor clean buffers. If these pages are needed elsewhere, then the UBC can recycle them as needed without intervention from AdvFS. If a clean page is recycled and is later needed by AdvFS, it must be brought back into the cache. Maintaining dirty buffers is more complicated, since the manipulation of dirty pages may be initiated by either AdvFS or the UBC, as we shall shortly see.

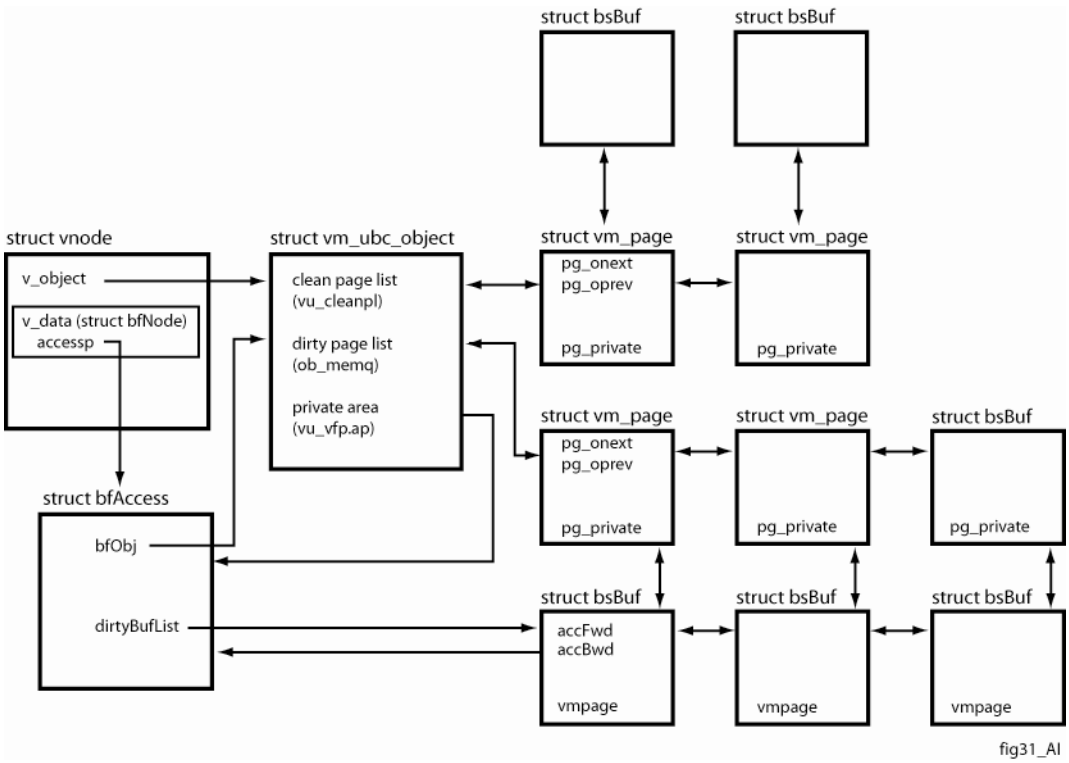


Figure 30: VFS, UBC, and VM in-memory structures

7.3 Buffer Cache Actions during System Calls

7.3.1 Page Lookup, Pinning , and Reffing

Before looking at some file-specific system calls, let's look at a UBC operation that is common to many of those paths. This operation is the page lookup, and is done whenever the filesystem or the UBC needs to determine if a certain file page is already in the cache. Pages are uniquely identified by their `vm_ubc_object` and their offset in the file.

To determine if a given file page is in the cache, AdvFS calls *ubc_lookup(),* passing the object, offset, and a flag controlling the lookup. Within this routine, the object's lock (`vm_ubc_object.ob_lock`) is seized to synchronize threads with racing lookups on the same object. After acquiring the lock, the UBC searches its cache for an existing reference to the page. If the page is not in the cache (a **cache miss**), the UBC allocates a new UBC page (a `vm_page` structure), marks the page busy (by setting `vm_page.pg_busy`), and increments the page's reference counter (`vm_page.pg_hold`). The latter field is incremented to prevent the page from being manipulated by other threads while it is in use. This busy, held page is then returned to the caller. In the case of a cache miss, *ubc_lookup()* also sets the B_NOCACHE bit in the flags parameter to indicate to AdvFS that the page was not found in the cache. This is because additional work is usually necessary to set up a new page when it is returned to AdvFS.

If the page is found in the cache (a **cache hit**), but is currently marked as busy (`pg_busy` flag is set), then the object lock is released and the caller is put to sleep until the `pg_busy` flag is cleared. The page can be busy either because it is newly initialized or there is an I/O in progress to that page. Either way, the current thread seeking access to the page must wait until the page is in a consistent state. Once the `pg_busy` condition is clear, threads waiting for the page reference are awakened and given the reference to the page. After that, the page has its reference counter (`vm_page.pg_hold`) incremented, and is returned to the caller.

The act of bringing a page into the buffer cache and incrementing its `pg_hold` so that the data in the page can be manipulated by a single thread is called **referencing** or **reffing** the page (if the page is being read) or **pinning** the page (if the page is being modified). Each time a page is reffed or pinned, it must be followed by a deref (call to *`bs_derefpg()`*) or unpinned (call to *`bs_unpinpg()`*) to allow other threads to manipulate these pages. Basically, a deref just decrements the `pg_hold` count on the page. An unpin operation is slightly more complicated depending on the type of page being unpinned and the mode of unpin required. This will be discussed further in Section 7.3.5 on buffer cache write operations.

There are two additional features that *ubc_lookup()* provides that are useful to AdvFS. One of the parameters to this function is a flag, which AdvFS typically passes as B_READ or B_WRITE. If B_WRITE is passed, then it is assumed that the page will be modified, and the `vm_page.pg_dirty` bit will be set when it is returned to the caller. If B_READ is passed, then it is assumed that the page will be used for a read operation and `vm_page.pg_dirty` will not be set (unless the page is already dirty).

The second feature also uses the flag parameter. If AdvFS passes the B_CACHE flag, this is a signal to *ubc_lookup()* that it wants to know if the page is in the cache, but does not want a new page to be initialized if it does not already exist. In this case, if there is a cache miss, no page is initialized and B_NOCACHE is returned. For example, direct I/O uses this flag to check for the existence of a page in the cache without causing a new cached page to be created if it doesn't already exist.

Simplified *ubc_lookup()* code flow summary:

ubc_lookup(vm_ubc_object, offset, blocksize, length, *pageptr, *flags, policy)
    Seize object lock
    Lookup the page in the cache

```
If cache miss:
    If B_CACHE was set:
        set B_NOCACHE flag
        Set *pageptr to NULL to indicate no page being returned
    Else
        Allocate a new page by calling ubc_page_alloc()
        Set pg_busy and increment pg_hold
        Set *pageptr to address of new page
Else cache hit:
    Increment vm_page.pg_hold
    Set *pageptr to address of page found in cache
Release object lock
return
```

7.3.2 File Open

The first open of a file causes AdvFS to create its own internal file structure to manage the file in memory. This `bfAccess` structure stores a pointer to the file's `vnode` along with other information that AdvFS needs to manage the file's activity. The `vnode` also holds a pointer back to the `bfAccess` structure in the `bfNode` structure that is found at `vnode->v_data`. AdvFS will then obtain a UBC file object to associate with the `bfAccess` structure by calling *ubc_object_allocate()*. The UBC allocates a new `vm_ubc_object` structure which contains a pointer to the `bfAccess` structure and a filesystem-specific callback dispatch table. The `vm_ubc_object` is the structure that the UBC will use to manage the cached pages associated with this file. File structures are discussed in more detail in Chapter 2.

In addition, for metadata files or files under transaction log control, AdvFS will indicate to UBC that the file object must be managed specially by calling *UBC_OBJECT_PREVENT_FLUSH()*. This is needed to maintain the write-ahead log rule. Special management considerations for metadata files are discussed in Section 7.4.

Simplified code flow summary for file open:

```
Object = ubc_object_allocate(handle, dispatch table, funnel)
If AdvFS metadata controlled file:
    UBC_OBJECT_PREVENT_FLUSH(object) sets object's UBC_NOFLUSH flag
```

7.3.3 File Close and Inactivation

Some time after a file is closed for the last time, AdvFS will recycle its `bfAccess` structure to end its relationship with that file. This recycling can be initiated by AdvFS when it needs a new `bfAccess` structure, or by the VFS subsystem when it needs to reclaim a `vnode`. (The freeing of the `bfAccess` structure is not done immediately upon closing the file in anticipation that the file will be reopened and the existing `bfAccess` and `vnode` structures will be reused, saving the overhead of initializing them again). During this reclamation, AdvFS will call *ubc_object_free()* to disassociate the `vm_ubc_object` from the `bfAccess` structure and deallocate it. The UBC assumes that AdvFS has invalidated all of the cache pages associated with that object before *ubc_object_free()* is called.

7.3.4 Read

AdvFS processes each *read()* request as a series of 8 KB data buffers, each equivalent to a page of data. For each page, a call to *ubc_lookup*(B_READ) searches the UBC cache for the specified page. It returns either a new UBC page (if the page is not in the cache), or the existing page (if it was found in the cache). This is called **reffing** the page.

After a cache miss, AdvFS dynamically allocates its own state structure, called a `bsBuf`, and attaches it to the UBC page at `vm_page._upg._pg_private`. This `vm_page/bsBuf` association will remain until UBC recycles the page. Next, AdvFS issues a read to the disk driver to initialize the page with data from the on-disk file. After the I/O completes, AdvFS calls *ubc_page_release*(B_DONE) so that UBC can clear the busy state and wakeup any threads waiting for the page. Next, the cached data is copied into the application's memory buffer, and *ubc_page_release()* is called a second time to allow the UBC to decrement the page reference counter.

After a cache hit, AdvFS simply copies the cached data to the application's buffer and calls *ubc_page_release()* to decrement the page reference counter (deref the page).

read() system call code flow summary:

read(file handle, offset, length)
    For each page (8Kb) of data:
        Reference the page:
            ubc_lookup(object, pg offset, pg length, B_READ)
            If cache miss:
                Allocate `bsBuf` and associate with UBC page
                Start I/O to read file data into new cache page
                ubc_page_wait(page) to wait for I/O completion
                *ubc_page_release(*B_DONE) to clear `vm_page.pg_busy`
        Copy cached data to application buffer via *uiomove()*
        Dereference the page:
            ubc_page_release() decrements `vm_page.pg_hold`

7.3.5 Write

AdvFS processes each *write()* request as a series of 8 KB data buffers, each equivalent to a page of data. Each of these pages must be pinned before it can have its data modified. For each page a call to *ubc_lookup(B_WRITE)* searches the UBC cache for the specified page. It returns either a new UBC page if the page is not in the cache, or the existing page if it was found in the cache.

After a cache miss, AdvFS allocates a new `bsBuf` structure and attaches it to the UBC page in the `vm_page._upg._pg_private` field. Next, AdvFS issues an I/O to read the data from disk into the new page. When the I/O completes, AdvFS calls *ubc_page_release*(B_DONE) to clear the busy state and wakeup any threads waiting on the page. At this point the page is pinned. The application's data is then copied into the cache page, and the page is unpinned.

After a cache hit, the page is returned from *ubc_lookup()* with `vm_page.pg_dirty` set because the B_WRITE flag was passed in. At this point the page is pinned. Then the data is copied from the application buffer into the cached page, and the page is unpinned.

Unpinning a page may involve different steps depending on the kind of page and its state. The *bs_unpinpg()* routine has several different code paths that can be specified by the calling routine. Each path needs to decrement the `pg_hold` on the buffer, but there may be other work that is also required.

One path is for log pages (BS_LOG_PAGE). In this case, the LSN for the page is checked, and, if there is a flush in progress that spans this page, then it is put onto the blocking I/O queue so that it will be flushed immediately. Otherwise the page has its `pg_hold` decremented, and there is a check to see if the log needs to be flushed (it is flushed every 8 pages). Another path is for a page that is being unpinned and must be flushed to disk immediately (BS_MOD_SYNC). In this case, the page is put onto the file's dirty buffer list if it is not already there, it is marked DIRTY and BUSY, it is staged for I/O, and put onto the blocking I/O queue. Then the code waits for the I/O to complete and decrements the `pg_hold` value. A third path involves pages that have not been modified (BS_NOMOD). If this is the last unpin of the buffer and the page is not already dirty, then the `pg_hold` is simply decremented. Otherwise, some other thread has also modified this page, and it needs to be placed onto an I/O queue. It will be placed onto the blocking I/O queue if a thread is waiting for a file flush that includes this page range. Otherwise it is simply put onto the lazy I/O queue. The last type of unpin is the most common, the lazy unpin (BS_MOD_LAZY). In this case the page is marked DIRTY, and put onto the file's dirty buffer list. It is put onto the blocking I/O queue if there is a flush outstanding that includes this page; otherwise it is put onto the lazy I/O queue and `pg_hold` is decremented. As mentioned in Chapter 8, a buffer that is on the blocking I/O queue will be flushed to disk almost immediately, while a buffer on the lazy I/O queue may languish for a while before being flushed to disk.

Note that this code path gets the data from the application buffer to the cache, but does not deal with flushing the cached page to disk. That path is discussed in Sections 7.3.6. and 7.3.8.

Write() system call code flow (simplified):

*write(*file handle, offset, length)
    For each page (8Kb) of data:
        Pin the Page*:*
            *ubc_lookup(*object, pg offset, pg length, B_WRITE)
            If cache miss:
                Allocate `bsBuf` structure and associate with UBC page
                Start I/O to read file data into new cache page
                *ubc_page_wait(*page) to wait for I/O completion
                *ubc_page_release(*B_DONE) clears page BUSY state
        Copy data from application buffer to cache via *uiomove()*
        Unpin the Page*:*
            Move the `ioDesc` for buffer to AdvFS I/O queue
            ubc_page_release() decrements page reference

7.3.6 Flushing Pages

AdvFS and UBC both have the ability to flush dirty cached pages to disk. AdvFS supports its own file sync routines (*sync()* and *fsync()*) as well as its own cache I/O scheduling mechanisms. AdvFS also supports the smoothsync mechanism that periodically flushes dirty data based upon an aging timestamp scheme (See Section 8.10). The UBC may need to flush pages if it needs to reclaim pages for other uses, either for new file pages or for other memory requirements.

If the the UBC is initiating the flush, it will select a dirty page to be flushed based on a '**least-recently-used (LRU)**' basis, meaning that the page that has not been touched the longest will be flushed. To start the flush, the UBC will first stage the page for I/O by moving it from the object's dirty list to the clean list, clearing `vm_page.pg_dirty,` and setting `vm_page.pg_busy`. The UBC then tells the filesystem to flush the page by calling *FSOP_PUTPAGE()*. For an AdvFS filesystem, this will dispatch into *msfs_putpage()*. *msfs_putpage()* will call *ubc_dirty_kluster()* to get a list of pages for this object that can fit into one I/O. This is an optimization that assumes that if the UBC is requesting one page to be flushed, it probably needs more than one page. So AdvFS tries to collect nearby pages that can be

consolidated into a single I/O operation. These pages are then flushed to disk using an asynchronous, non-blocking I/O request that the AdvFS I/O subsystem will manage. The code flow for this path is given at the end of this section.

Flushing of dirty metadata is initiated only by AdvFS and is described Section 7.5. Flushing of application data by AdvFS can be from a number of routines, some of which are included in the following table:

Flush Routine	Purpose
bfflush()	Flush pages within a page range (or all pages) of a file
bs_bflush()	Flush buffers on lazy queue for a given disk
bs_bfdmn_flush_bfrs()	Flush buffers on all disks in a domain
bs_bfdmn_flush_all	Flush buffers on all active domains
bs_bfs_flush	Flush and invalidate all buffers within a fileset

The routines that flush buffers for disks and domains really just ensure that all the buffers on the appropriate lazy I/O queues get moved onto the flush I/O queues for immediate flushing to disk. The *bfflush()* routine is a more generally-used routine for flushing files via *fsync()* or when a file is opened with the O_SYNC flag. This routine actually walks through the file's dirty buffer list (`bfap->dirtyBufList`), staging each page for I/O, and moving the buffer to the flush I/O queue. (See Chapter 8 for a discussion of the I/O queues).

msfs_putpage() code flow:

*msfs_putpage(*object, pagelist, pagecount, flags, ucred)
For each page in the pagelist:
ubc_dirty_kluster(object, page, offset, kluster pg length wanted, flags, pgcnt): This creates a list of contiguous pages that are all staged for I/O
Consolidate the list of pages and insert it onto the ubcReq I/O queue
Issue the I/O to disk
Upon I/O completion, call *ubc_page_release*(B_DONE) to clear each page's BUSY state

7.3.7 Invalidating Pages

When the UBC recycles a cached page, it calls back to AdvFS via *FSOP_FS_CLEANUP()* to deallocate any `bsBuf` structure that is associated with the UBC page. The UBC will then recycle the UBC page and remove it from the object's clean list. Recycling removes the page from the cache. This whole process is known as invalidating pages. Note that invalidating pages means removing them from the cache and throwing away the contents, even if the page has been modified. Paths that want to flush and then invalidate certain pages do the flushing first to make the pages clean, and then invalidate them.

When AdvFS needs to invalidate pages, such as when a file is deleted or truncated, AdvFS will usually call *bs_invalidate_pages()*. This routine walks through the file's dirty buffer list, remove the `bsBuf` from any I/O queue on

which it resides, and then calls *ubc_invalidate()* to invalidate the page from the cache and wake up any threads waiting for the page. Waiters are returned a status indicating that the page no longer exists. In

several paths, particularly error paths within the buffer cache code, *ubc_invalidate()* may be called directly.

7.3.8 Performing I/O on Cached Pages

To perform I/O on a cached page, the kernel must prepare the page for I/O, start the I/O, and later process the I/O completion. While an I/O is in progress, the page is marked busy (`vm_page.pg_busy` is set) to prevent other threads from accessing the page. At I/O completion, the `vm_page.pg_busy` flag is cleared and waiting threads awakened. The following figure illustrates the I/O processing for cached pages.

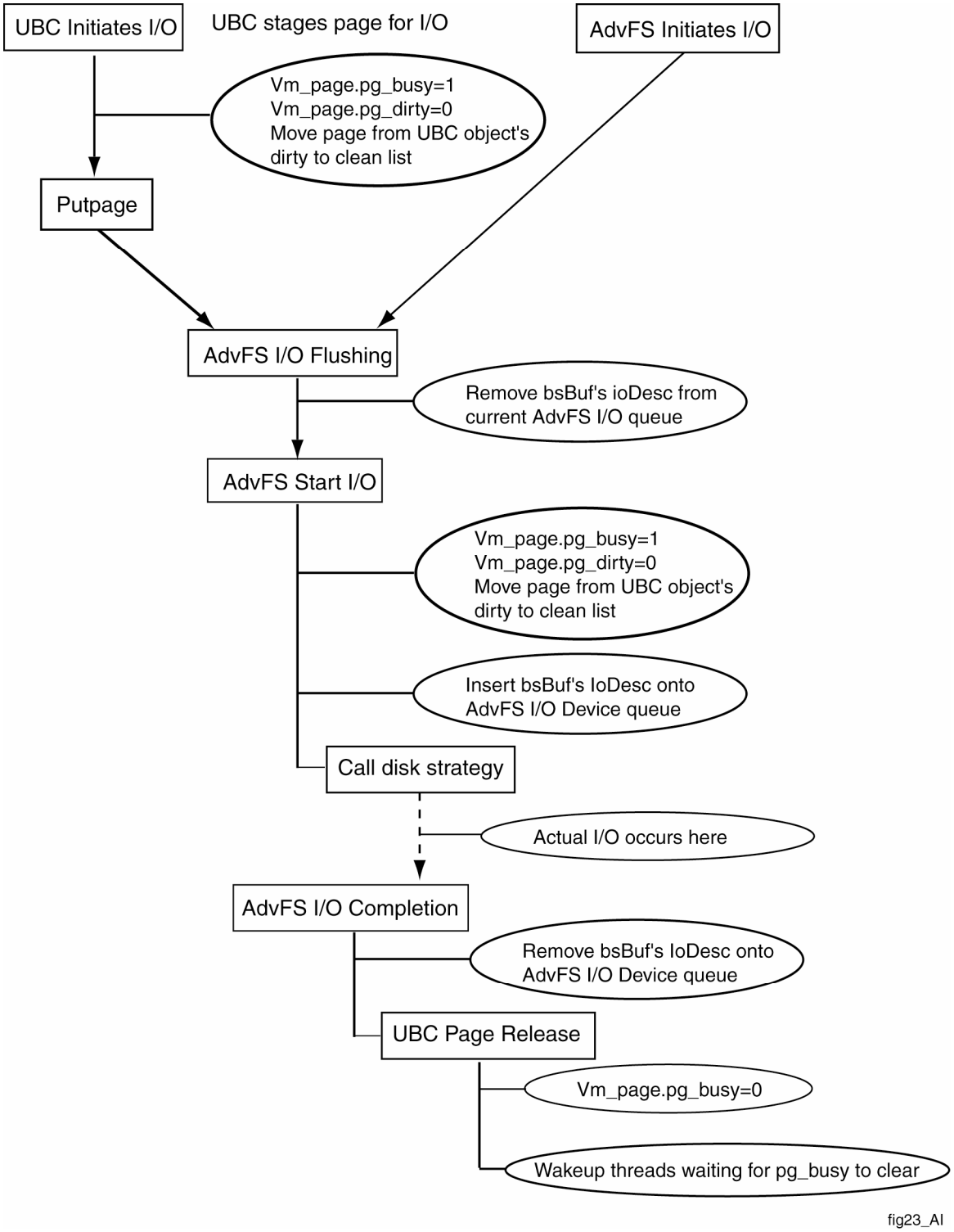


Figure 31: Flowchart of I/O processing for cached pages

If UBC initiates the I/O, the UBC sets up the page as described in Section 7.3.6 and then calls *FSOP_PUTPAGE()*. The filesystem then removes the page from whatever lazy I/O queue that it is

currently on, inserts it onto the AdvFS ubcReq queue, and then calls the *bs_startio()* routine to start the I/O.

If AdvFS initiates the I/O, AdvFS removes the page from whatever lazy I/O queue it is on, and stages the page for I/O by calling *advfs_page_busy()*. This routine does much the same thing as the UBC does in staging a page for I/O: the page is moved to the object's clean list, `vm_page.pg_dirty` is cleared, and `vm_page.pg_busy` is set. Next, AdvFS inserts the pages onto either the blocking, ubcReq, or flush queue, and then calls the *bs_startio()* routine. Which I/O queue the page is placed on depends on the calling routine. See Chapter 8 for more details about the I/O queues.

UBC follows these steps to flush a modified cache page to disk:

Remove the page from the object's dirty list
Insert the page onto the object's clean list
Set `vm_page.pg_busy` and clear `vm_page.pg_dirty`
Call *FSOP_PUTPAGE()* to have the filesystem start the I/O.

AdvFS follows these steps to flush a modified cache page to disk:

Remove the page's I/O descriptor from the lazy I/O queue on which it resides
Call *advfs_page_busy()* to:
    Remove the page from the object's dirty list
    Insert the page onto object's clean page list
    Set `vm_page.pg_busy` and clear `vm_page.pg_dirty`
Insert the page's I/O descriptor onto the appropriate non-lazy I/O queue
Call *bs_startio()*

When the I/O completes, the buffer's I/O descriptor is removed from the device queue, the page remains on the object's clean list, and the page's `vm_page.pg_busy` flag is cleared. Threads waiting for the I/O to complete, as determined by the clearing of the `pg_busy` flag, are awakened.

7.4 Metadata Handling

Metadata refers to information needed to describe files within the filesystem. Examples of metadata for a given file include its size, date and time of modification, and storage location on disk. Metadata must be handled carefully to ensure the integrity of the filesystem as a whole. Most metadata is stored in a reserved file called the BMT, but AdvFS organizes some types of metadata into distinct files, such as the storage allocation bitmap (SBM). AdvFS also supports **atomic data logging** (**ADL**; see Section 8.2) of user files, which requires that the data in those files be managed the same as metadata.

AdvFS uses the UBC to cache metadata as well as normal data files. This arrangement allows AdvFS to use a single cache interface and to maintain a pool of dynamic cache memory for varying loads of metadata-intensive operations.

Clean metadata pages that are not referenced are subject to recycling at anytime by UBC. By definition, clean pages contain no modifications (contain the same data as the page on disk). Since AdvFS has no special ordering requirements on these pages (as there are for dirty metadata pages), the UBC is allowed to manage the clean metadata pages just like regular user data pages.

To ensure the recoverability of metadata in the event of a system crash, AdvFS must control the sequence in which metadata and transaction log modifications are written to disk. This is the **write-ahead log rule**, and guarantees that log records describing metadata changes are on disk before the actual metadata changes are written to disk (see Section 9.4). Therefore, AdvFS, and not the UBC, must control the

timing of metadata flushing. To ensure that this is true, the UBC routines that do dirty page flushing first call *FSOP_WRITE_CHECK()* to see if a file object is under metadata control. If an object has the UBC_NOFLUSH flag set, then it is under metadata control (see Section 7.3.2) and UBC will not flush the dirty page.

When AdvFS modifies a metadata page and then calls *ubc_page_release(),* the UBC will insert the dirty page onto the UBC dirty metadata LRU when this is the last reference to the page. The dirty metadata LRU is a linked list at `ubc_cntl_t->ubc_dirty_metadata_lru` with a count of the pages on this list in `ubc_cntl_t->ubc_dirty_metadata_count`. The dirty metadata LRU allows the UBC to monitor the percentage of all UBC pages that are dirty metadata. If the percentage exceeds a UBC threshold limit (kept in `ubc_cntl_t->ubc_dirty_metadata_pcnt`), the UBC can request that the filesystem start flushing some of its dirty metadata. This request is made by calling *FSOP_FS_METADATA_PUT()* and passing in the UBC object of a dirty metadata page. For AdvFS, this dispatches to the routine *msfs_log_and_meta_flush(),* that flushes all pages for the file requested. This allows UBC to indirectly manage the number of dirty UBC pages while allowing AdvFS to maintain the ordering of log and metadata writes.

The threshold percentage of dirty UBC metadata pages is modifiable by the VM configurable parameter `ubc_maxdirtymetadata_pcnt` (default value 70%).

7.5 UBC Page Recycling

Recycling (or freeing) of UBC pages may be initiated by either AdvFS or the UBC. The UBC will initiate recycling of cached pages either to get memory for new UBC page requests or to give the memory back to the Virtual Memory subsystem. This allows UBC to dynamically adjust its consumption of memory for file caching versus general VM requirements.

AdvFS may also invalidate and free a file's cached pages during file delete, file truncate, fileset `unmount,` or a direct I/O write of a previously-cached page. To do this, AdvFS calls *ubc_invalidate()* passing in the page range that the UBC should invalidate.

When the UBC recycles a page, it calls *FSOP_FS_CLEANUP()* to allow the filesystem to cleanup its filesystem-specific state structures in the page's `vm_page._upg._pg_private` field. For AdvFS, the filesystem state structure is the `bsBuf` structure. The *FSOP_FS_CLEANUP()* call dispatches to *msfs_fs_cleanup(),* where AdvFS retrieves its page-specific `bsBuf` structure pointer from `vm_page._upg._pg_private` and then deallocates the `bsBuf` structure. The UBC permits filesystems to use the `_pg_private` field without restriction while the page represents valid file data. After the call returns, the UBC zeroes this field, completing the disassociation of the UBC page structure from the file system structure.

7.6 Memory Mapping

As mentioned in section 4.6.11, memory mapping functions (*mmap()* and *munmap())* can be used by an application to represent a file by a region of memory. Using this method, simply modifying the mapped memory region will result in the eventual writing of that data to the file. This mechanism is largely handled by the VM subsystem. If a mapped memory page is touched (either for reading or writing) and this page is not resident in virtual memory, the VM will **fault** the page. Faulting is a process whereby the page is brought into virtual memory and made available to the application. For a memory-mapped file,

part of this process involves reading the page from disk into memory so that the data can be read and/or modified. When a file page is faulted and brought into memory, the VM calls a filesystem-specific 'getpage' routine via the *FSOP_GETPAGE()* macro. For AdvFS this will invoke *msfs_getpage()*. As can be seen in the code flow below, *msfs_getpage()* must do operations similar to reffing or pinning the page. It will return each page with its `pg_hold` variable incremented. The fault handler will set a read or write permission on the page and then decrement the `pg_hold`. This means that the page can be recycled by the UBC at any time after the `pg_hold` is dropped. If the page is recycled and the application touches the page again, the whole faulting process will restart to bring the page back into the cache.

When *mmap()* is called by the application, a parameter is specified that allows the mapped pages to be read-only (READ) or read/write enabled (WRITE). If the page being faulted is to be read-only, then *msfs_getpage()* reads the page into the UBC and it is marked clean. The data can be read by the application, but the fault handler puts a write-protection on this page so that if the application tries to modify the page, a memory fault is generated (but not handled). Also, since the page is clean, the UBC can reclaim it at any time without the intervention or knowledge of AdvFS. If a page is memory-mapped for writing, however, the page is returned from *msfs_getpage()* marked DIRTY. This ensures that if VM subsequently reclaims the page it will get flushed to disk. This page is not explicitly put onto the `bfap->dirtyBufList`, however. (We are not sure why this is, exactly. This could be an area of investigation.) This means that calls to *fsync()* will not cause memory-mapped pages to be flushed to disk. This is because *fsync()* will call *bfflush()* which walks the `bfap->dirtyBufList` for pages that need to be flushed. There are only three mechanisms that will cause memory-mapped pages to be flushed to disk. First, the *msync()* call which will call into *ubc_msync()* to walk the list of dirty pages on the file's `vm_ubc_object`. Second, the coarse-granularity smoothsync call to *msfs_sync_mmap()* that flushes pages by calling *ubc_flush_dirty()*. Third, the UBC may flush these pages as they need to be reclaimed. This is done by calling *msfs_putpage()* to flush their contents to disk, and then the clean page can be reclaimed.

msfs_getpage code flow:

msfs_getpage(object, offset, length, protection, pagelist, read/write flag, ucred)
- If writing to a sparse hole, allocate storage for the page (cannot extend the file here):
- For each page of data:
  - If mapping for read:
    - *ubc_lookup*(object, pg offset, pg length, READ)
    - If cache miss:
      - Allocate bsBuf and associate with UBC page
      - Queue I/O to read file data into new cache page
      - *ubc_page_wait*(page) to wait for I/O completion
      - *ubc_page_release*(B_DONE) clears page BUSY state
  - Else mapping for write:
    - *ubc_lookup*(object, pg offset, pg length, WRITE)
    - If cache miss:
      - Allocate bsBuf and associate with UBC page
      - Queue I/O to read file data into new cache page
      - *ubc_page_wait*(page) to wait for I/O completion
      - *ubc_page_release*(B_DONE) clears page BUSY state
    - Mark the page DIRTY so it will get written to disk if reclaimed.
- Return pages with `pg_hold` incremented.

7.7 Interaction with directIO

By definition, direct I/O bypasses the buffer cache. This means that any data already in cached pages must be handled to avoid data corruption. In summary, for non-clustered systems, any pages already in the cache will be used to return data for a direct I/O read, and the pages are left in the cache. For direct I/O writes, the new data is merged with the cached data, then flushed to disk, and the cached pages are invalidated. This means that direct I/O writes tend to remove pages from the cache. In clustered systems, cached pages are flushed and invalidated when the file is first opened for direct I/O, and pages are not brought into the cache. In operations such as migration which operate by bringing pages into the cache, all pages brought into the cache are invalidated before the operation ends. See Section 8.13.3 for details.

7.8 Big Pages

Needs to be added.

Chapter 8: I/O Subsystem

The I/O subsystem is responsible for writing data that is in memory to permanent storage devices (volumes) and for retrieving data from storage into memory. This subsystem is tightly integrated with the buffer caching subsystem. See Chapter 7 for an explanation of buffer caching and Chapter 4 for a discussion of read-ahead and aggressive write flushing.

While the time to access data in memory may be 25 to 50 machine cycles, the time required to transfer that same data from disk may involve millions of machine cycles. Therefore, it is important that the I/O subsystem be as efficient as possible. To increase efficiency, data is buffered in memory when possible and appropriate, I/Os are avoided when possible, and the data is transferred to the driver in as efficient a size as possible.

8.1 Asynchronous and Synchronous I/O

Write requests, by default, are cached. That is, data is written to the buffer cache and scheduled for later transfer to disk. This method, called **asynchronous I/O**, generally gives the highest throughput, because the application does not have to wait for disk I/O to complete and multiple changes to the same page may be combined into one physical write to disk. Asynchronous I/O decreases disk traffic and increases the concurrent access of common data by multiple threads. In addition, delaying the write to disk increases the likelihood that a page will be combined with other contiguous pages into a single, more efficient transfer. This is known as **I/O consolidation** (see section 8.6).

One disadvantage of asynchronous I/O is that the likelihood of losing data during a system crash increases. When a fileset is mounted after a crash, the completed log transactions are replayed and incomplete transactions are backed out so that the original metadata on disk is restored. These log transactions, by default, save only metadata, not the data written to the file. This means that file sizes and locations on disk are consistent but, depending on when the crash occurs, the user data from recent writes may be out of date because the data was left in the buffer cache and lost during reboot.

Synchronous I/O, by contrast, ensures that both the metadata and the application data are written to disk before the write request returns to the calling application. This means that if a write is successful, the data is guaranteed to be on disk. Synchronous I/O reduces throughput because the write does not return until after the I/O has completed. In addition, because the application, not the file system, determines when the data needs to be flushed to disk, the likelihood of consolidating I/Os is reduced. The advantage, however, is that application data is safely in permanent storage, and a system crash cannot jeopardize its data consistency.

To avoid the I/O latency in synchronous operations such as reads and direct I/O writes, there is an AIO interface that allows the application to work around the I/O latency. This interface allows the application to start a series of operations such as read and write (using *aio_read()* and *aio_write()* calls) and later to go back and check the status of each I/O requested. See section 15.2, plus the `aio` reference pages for more information on the AIO interface.

8.2 Atomic-write Data Logging

Atomic-write data logging (ADL) writes user data (in addition to the normally logged metadata) to the log file so data is consistent in the event of a system crash. ADL guarantees that either the metadata and

the associated file data are written to disk, or neither is updated on disk. The I/O method that ADL uses depends on whether your I/O has been set to asynchronous or synchronous.

8.2.1 Asynchronous ADL

Asynchronous ADL I/O is similar to asynchronous I/O except that the user data written to the buffer cache is also written to the log file for each write request. This is done in 8k increments. The extra write of the data to the log file ensures data consistency in the event of a crash, but it can degrade throughput compared with using only asynchronous I/O. If atomic-write data logging is enabled while asynchronous I/O is being used, the resulting I/O is asynchronous ADL.

If a crash occurs, the data is recovered from the log file when the fileset is remounted. As in asynchronous I/O, all completed log transactions are replayed and incomplete transactions are backed out. Unlike asynchronous I/O, however, the user's data has been written to the log, so both the metadata and the data intended for the file can be restored. This guarantees that each 8k increment of a write is atomic. It is either completely written to disk or is not written to disk at all.

Because only completed write requests are processed, obsolete and possibly sensitive data located where the system was about to write at the time of the crash can never be accessed. Protecting such data is called **object safety**. Out-of-order disk writes, which might cause inconsistencies in the event of a crash, can never occur.

8.2.2 Synchronous ADL

Synchronous ADL is similar to asynchronous ADL except that the logged data is flushed from the buffer cache to disk before the write request returns to the calling application. Throughput is likely to be degraded compared with using asynchronous ADL, because the write does not return until after the log-flushing I/O is complete. If ADL is enabled while synchronous I/O is being used, the resulting I/O is synchronous ADL.

The benefit of synchronous ADL, compared with asynchronous ADL, is the guarantee of data consistency when a crash occurs after a write call returns to the application. On reboot, the log file is replayed and the user's entire write request is written to the appropriate user data file. In contrast, asynchronous ADL guarantees the consistency of only 8K increments of data after the write call returns.

Two types of atomic-write data logging are available: persistent and temporary. Persistent data logging remains in effect across mounts and unmounts. Temporary data logging is activated for the duration of the mount. The logging status of a file can be checked using the `chfile` command with no options.

To turn persistent atomic-write data logging I/O on and off, use the `fcntl()` function or enter the `chfile` command with the `-L` option:

```
chfile -L on <filename>

chfile -L off <filename>
```

If a file has a frag, persistent ADL cannot be activated. Conversely, if persistent ADL is activated on a file, the file will not be fragged (see section 6.2.1 for more information on frags). Persistent ADL files cannot be fragged because they have `bfAccess->dataSafety = BFD_FTX_AGENT`. This is the value used by metadata files. The way data logging is implemented for user files is to make them look like, and therefore be treated as, metadata files. Metadata files (BMT, RBMT, SBM, etc) are not fragged. To activate data logging on a file that has a frag, either use temporary atomic-write data logging, or disable frags by using the command:

```
chfsets -o nofrag <domain> <fileset>
```

Data logging incurs a performance cost because data as well as metadata is written to the transaction log. This increases the amount of traffic to the log for that domain and doubles the I/O for each write requested. Grouping files or filesets for which ADL is activated into certain domains can be used to reduce the increased logging burden on other more performance-sensitive domains.

Files that use persistent ADL cannot be memory mapped or opened for direct I/O. In either case, an error is returned. Temporary ADL files can be memory-mapped but cannot be opened for direct I/O. Internally, when a temporary ADL file is mmapped, the temporary data logging is suspended until the last thread using the mmapped file unmaps it. Although it would be complicated, there is no technical reason why ADL (persistent or temporary) and direct I/O are mutually exclusive. However, ADL would greatly reduce performance, which is a significant bonus gained by using direct I/O. Additionally, there is no technical reason that persistent ADL must be mutually exclusive with memory mapping. These dual functionalities were not implemented due to a tight schedule when ADL code was added to the code base.

8.3 I/O Queues

When a page in the buffer cache is dirty, AdvFS places it onto one of the I/O queues so that it can be scheduled for I/O. The same thing happens when a read is requested. The following illustration shows the movement of the synchronous and asynchronous I/O requests through the AdvFS I/O queues.

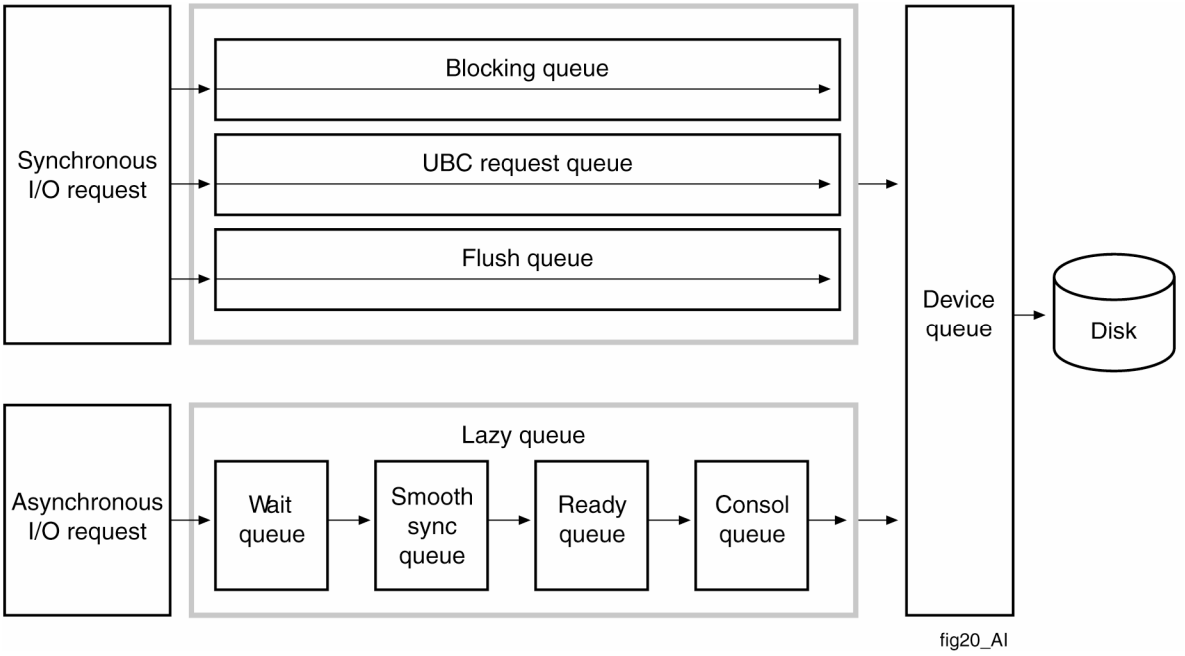


Figure 32: AdvFS I/O Queues

All reads are placed onto the blocking queue, while synchronous write requests are placed onto either the blocking, ubcreq, or flush queues. A synchronous write request must be written to disk before it is considered complete and the application can continue.

The **blocking queue** is used primarily for reads and for AdvFS-generated synchronous write requests (for example. log flushes). The **ubcreq queue** caches synchronous write requests that come from the UBC. These are required primarily to flush dirty pages to disk so that the memory pages can be freed. The **flush queue** is used primarily for buffer write requests from applications, either through *fsync(), sync(),* or synchronous writes. Because the buffers on the blocking queue are given slightly higher priority than those on the ubcreq or flush queues, kernel requests are handled more expeditiously and are not blocked if many buffers are waiting to be written to disk.

Processes that need to read or modify data in a buffer that is on the blocking, flush, or ubcreq queues must wait for the data to be written to disk. This is in direct contrast to buffers on the lazy queues that can be modified at any time until they are finally moved down to the device queue.

The **lazy queue** is a logical series of queues in which asynchronous write requests are cached. When an asynchronous I/O request enters the lazy queue, it is assigned a time stamp. This time stamp is used to periodically flush the buffers down toward the disk in numbers large enough to allow them to be consolidated into larger I/Os. Processes can modify data in buffers at any time if they are on the lazy queue, potentially avoiding additional I/Os. Scheduling the writing of buffers on the lazy queue is a trade-off between writing the buffer as soon as the device is not busy and waiting as long as possible in anticipation that the page will be repinned and an extra I/O will not be needed.

There are four subqueues that make up the logical lazy queue:

wait queue

Asynchronous I/O requests that are waiting for a write to the AdvFS transaction log to complete enter the wait queue. Waiting maintains the write-ahead rule of the log, which ensures consistency of file metadata. Only buffers for metadata files or files being written with atomic-write data logging are put onto the wait queue. Buffers can be moved from this queue to the smoothsync queue only after their associated log data has been flushed to disk. The flushing is detected during I/O completion of a log page in *bs_io_complete()*, which sends a message of type CK_WAITQ to the background I/O thread. This thread runs the *check_cont_bits()* routine that moves the buffers that are now eligible to be written to either the smoothsync or the ready queue.

smoothsync queue

If smoothsync is enabled (see 8.10), buffers from the wait queue or buffers for user data are put onto the smoothsync queue. Asynchronous I/O requests remain, by default, on the smoothsync queue for at least 30 seconds. Because there are 16 subqueues in the smoothsync queue, most buffers stay in each queue for approximately two seconds. Under normal conditions, the smoothsync queue is where buffers spend the most time. Allowing requests to languish in the smoothsync queue prevents I/O spikes, increases cache hit rates, and improves the consolidation of requests. The addition and removal of buffers from the smoothsync queues are based on time stamps in the buffer. These time stamps are, by default, set when the buffer is first dirtied; but this behavior can be modified to reset the time stamp each time the buffer is modified by mounting the fileset with the `-o smsync2` attribute (see the mount reference page).

ready queue

After buffers have aged in the smoothsync queue or if smoothsync is disabled, they move to the ready queue. Asynchronous I/O requests are sorted by logical block number (LBN) as they are moved onto the ready queue, increasing the likelihood of consolidating the I/O for these buffers. After the ready queue reaches a specified size (based on the `AdvfsReadyQLim` tunable parameter), all buffers on the ready queue are moved onto the consol queue.

consol queue

> The name consol queue appears to be a holdover from an older architecture where the buffers were actually consolidated and then put onto this queue. This is no longer the case. The consol queue can be thought of as a holding tank for the presorted buffers waiting to be written. If a device is busy, the buffers can build up on the consol queue until they can be moved to the device for I/O. There is no limit to the size that the consol queue can reach. In fact, this queue is needed primarily to prevent the starvation of buffers representing pages with high LBNs. For example, if AdvFS sorted the buffers by LBN directly onto the consol queue, and the algorithm that feeds the device from the consol queue removes buffers from the head of the queue, it might be possible for the incoming rate of buffers with low LBNs to exceed that of the rate of removal from the consol queue. This might effectively cause buffers with higher LBNs to remain in the cache without being flushed for an indeterminate amount of time.

All I/O consolidation (see Section 8.6) is done as the buffers are moved from the blocking, flush, ubcreq, and consol queues onto the **device queue**. As buffers are moved onto the device queue, logically contiguous I/Os from the incoming queues are consolidated into larger I/O requests to reduce the number of I/Os that must be completed.

The algorithm that moves the buffers onto the device queue favors taking buffers from the blocking queue over the ubcreq and flush queues, and all three are favored over the lazy queue. Device and driver resources limit the number of buffers that can be placed onto the device queue. The algorithms that load the device queue use feedback from the drivers to know when the device queue is full (see section 8.5). At that point the device is saturated and continued movement of buffers to the device queue only degrades throughput of the device. The capacity of the device queue and how full it is ultimately determine how long it may take to complete a synchronous I/O operation.

Buffers on the device queue cannot be repinned or modified until their I/O has completed. For this reason, there is a flag in the `bsBuf` structure called REMOVE_FROM_IOQ that is set in the buffer pin path when it is known that this page is about to be pinned. The algorithm in *bs_startio()* explicitly avoids moving buffers with this flag set from the consol queue onto the device queue to prevent the pinning thread from having to wait until the I/O has completed.

While buffers are on any of the lazy queues, they can be repinned and removed from the I/O queues. This makes it possible to get several updates into a page before it is flushed to disk, making the whole process more efficient. For this reason, the algorithm that tries to move the buffers from the consol queue to the device queue is not very aggressive unless it is passed a flag (`IO_FLUSH`) telling it that the lazy queues need to be emptied.

In contrast, when a buffer is on the blocking, ubcreq, flush, or device queues, it cannot be repinned or referenced until the I/O is complete. Pages on the lazy queues are not staged for I/O (that is, they are not marked as busy and put onto the object's clean list) until they are moved onto the device queue. All buffers on the blocking, flush, ubcreq, and device queues are already staged for I/O.

Through Tru64 Unix Version 5.1A, there was one simple lock, `vdIoLock`, that was used to guard all the I/O queues. This lock was one of the most contentious locks in AdvFS. As of Version 5.1B, we broke this lock down into one simple lock per I/O queue. This lock is in the queue's `ioDescHdrT` structure, which also contains the links to the `ioDescT` structures on the queue, the queue length, the queue's length limit, and a field containing the cumulative number of elements placed onto this queue. The `lenLimit` field is only utilized for the readyLazy and device queues; it is zero for all the others.

8.4 Starting I/Os

The *bs_startio()* routine is responsible for starting I/O. It fills the device queue from the four incoming queues (blocking, flush, ubcreq, and consol queues) in a manner that maintains the priority of the incoming queues. This means that the buffers on the blocking queue get highest priority; the ubcreq queue is next highest, followed by the flush queue and the consol queue. This priority order is always maintained.

The caller specifies the number of buffers taken from the consol queue by passing one of three values for the flush flags:

- IO_FLUSH - the lazy queues should be completely drained
- IO_SOMEFLUSH - the lazy queue should be drained but not aggressively
- IO_NOFLUSH, - no buffers need be taken from the lazy queue

Regardless of the flush flags, *bs_startio()* must not overfill the device queue. There is a feedback mechanism in the I/O completion routine that maintains the theoretical maximum number of buffers that can be efficiently handled by the device. *bs_startio()* does not put more buffers onto the device queue than this value allows. See section 8.5 for an explanation of how this queue length limit is maintained.

One of the variables used in *bs_startio()* is the `vdT.qtodev` variable. This defaults to 5 and is used to determine the level of aggressiveness that is used when the IO_SOMEFLUSH flag is passed. The number of buffers on the consol queue divided by 2^{qtodev} yields the number of buffers to move from the consol queue to the device queue. Therefore, a `vdT.qtodev` value of 5 yields a divisor of 32 that results in moving approximately 3.125% (that is, 1/32) of the buffers on the consol queue to the device queue. This value can be changed for a given device using the `chvol -q` command. The *-q* flag is hidden, so it does not show up in the reference page for `chvol`. The default value is 5. If you have a reason to modify this value, it is probably best to keep it in the range of 1 to 8.

For each I/O that is placed onto the device queue, the routine *call_disk()* is invoked to hand the I/O off to the device driver. The *call_disk()* routine:

1. Allocates a `struct buf` and initializes it with the data needed by the driver to do the I/O.
2. Sets `buf.b_flags` to B_RAW and B_PHYS flags if this is a direct I/O call.
3. Sets `buf.b_vp` to NULL to avoid having *biodone()* change its vnode, then sets `buf.b_iodone` to point to *msfs_iodone()* and `buf.b_pagelist` to the address of the `ioDescT` address.
4. Invokes the device's strategy routine. As long as the device driver controls an I/O, its `ioDescT` remains on the device queue. In most recent versions of the code (post-Version 5.1A), a time stamp for when the I/O was started and the address of the `struct buf` that was handed to the device driver are saved in each `ioDescT`. This is done to allow easier debugging of I/Os that appear to be hung in the device driver code because it is possible to tell how long an I/O has been pending and which `struct buf` the device driver is dealing with.

Also, while there are I/Os outstanding on a device, `vdT.vdIoOut` is greater than 0 and `vdT.active.state` is ACTIVE_DISK.

8.5 Completing I/Os

I/O completion is done in several steps.

1. The disk driver calls *biodone()*, passing the `struct buf` for the I/O being completed. Because *`buf.b_vp`* was set to NULL (see Section 8.4), *biodone()* only calls the routine in `buf.b_iodone`, which has been set to *msfs_iodone()*. This routine links the `struct buf` onto the processor's structure at `pr->MsfsIodoneBuf` and schedules a lightweight context event for this processor.

2. The lightweight context thread calls *msfs_async_iodone_lwc()*, which walks through all the buffers chained onto `pr->MsfsIodoneBuf` and passes each in turn to *bs_osf_complete()*. This is the routine that, from the AdvFS perspective, is responsible for completion of all actual I/Os. No sleeping or complex locks are allowed. The `struct buf` that was originally handed off to the device driver is passed back to this routine. The `ioDescT` is extracted from `buf.b_pagelist`, and the `struct buf` is freed.

3. If the I/O was consolidated, the list of buffers is pulled apart and any waiting threads are awakened. If there are multiple I/Os scheduled to be completed for a given *`bsBuf`*, then the waiting threads are not awakened until the last I/O has actually completed `(bsBuf.ioCount == 0)`.

4. If the device queue is less than half full, which is determined by comparing the queue's `length` to its `lenLimit`, a message is sent to the background I/O thread so that it can try to start more I/Os on this device. This mechanism keeps the queued I/Os moving onto the device as resources allow.

The I/O completion routine is also responsible for performance sampling and maintaining the length limit for the device queue. Prior to Version 5.1 this was set with a system-wide configurable variable `AdvfsMaxDevQLen`. This variable was not sufficient because there can be devices with different capabilities on the system, and a single value might not be appropriate for all of them. For Version 5.1 and later, the length limit (`vdT.devQ.lenLimit`) is maintained internally for each device and is not configurable.

There are two algorithms for maintaining `lenLimit` in *bs_osf_complete()*. Both are responsible for setting the `vdT.devQ.lenLimit` field, which contains the length limit, in buffers, for the device queue.

1. The preferred algorithm uses feedback from the device driver. This algorithm can adjust the length limit either up or down. When an I/O is started, AdvFS places the current length of the device queue in `buf.b_iostats.devqlen`. When the I/O completes, the device driver places an *active time* and a *pending time* in the other `buf.b_iostats` fields. The total time required to service the I/O is the sum of the active and pending times.

   From the AdvFS perspective, pending time is wasted time. If possible, AdvFS would like to have most I/Os complete with the active time as close to the total time as possible. This seems to be the best way to maximize throughput to the device over time. So, on I/O completion, if the pending time is less than 1.3% of the total time, AdvFS increases the value of the `vdT.devQ.lenLimit` by the number of buffers declared in the global variable `Advfs_biostats_devq_delta`. This means that more buffers are outstanding on the device at one time. If pending time exceeds 25% of the total time, then AdvFS decreases the value of `vdT.devQ.lenLimit`. Because the need to scale back the number of buffers is usually more immediate than the need to increase it, this algorithm decreases the length limit at a rate three times faster than it is increased (`3 * Advfs_biostats_devq_delta`).

2. The second algorithm sets the length limit if the device driver does not provide the feedback. LSM and third-party drivers are examples of drivers that do not provide feedback. This algorithm currently (Version 5.1B) scales the length limit up but not down. It records the number of buffers having I/O completed in a 16 second sampling period and calculates the average number of buffers processed per unit time throughout the sampling period. If the number of buffers that can be processed in *n* seconds is greater than the current value of `vdT.devQ.lenLimit`, the length limit is set to this new value. The *n* seconds accrual value is set by the `sysconfig` variable `io_throttle_shift`.

   Changing `io_throttle_shift` sets the value in the internal variable `bio_time_shift` (`-4 < bio_time_shift < 4`). The following table shows the relationship between the values for `io_throttle_shift` and the statistical maximum time that an I/O spends on the device queue (accrual time).

io_throttle_shift	Accrual Time (secs)
-3	0.1253
-2	0.25
-1	0.5
0	1
1 (default)	2
2	4
3	8

   The length of the sampling period is not configurable, but it can be modified with a debugger. The internal variable `bio_sample_shift` determines the sampling period. By default this has a value of 4, which gives a sampling period of 16 seconds. (*hz* is left-shifted by this value, so this gives the number of cycles per second * 2^4 or 16.) Use the following table to set the desired value.

bio_sample_shift	Sampling period (secs)
5	32
4	16
3	8
2	4
1	2
0	1

AdvFS has a background I/O thread, *bs_io_thread()*. Prior to Version V5.1B, there was one I/O thread per system. As of Version 5.1B, there is one I/O thread per RAD. Not all I/Os are started by this I/O thread. Rather, it is present to maintain a consistent amount of activity on the device when the other threads are not keeping the device busy. The I/O thread is activated by messages from other threads, typically during an I/O completion. Because a volume may be removed from the domain before the I/O thread handles the message, the thread checks that a device is still valid before attempting to start more I/Os on it.

8.6 I/O Consolidation

I/O consolidation refers to the collection of individual I/O requests accessing contiguous space on the disk which can be combined into a single read or write to be handled by the device driver. This is an optimization that allows better throughput to the disk. I/Os are consolidated only to the point that the resulting transfer does not exceed the disk's `preferred_transfer_size.`

I/Os are not consolidated if they represent raw or direct I/O transfer requests. This is because the driver uses Direct Memory Access (DMA) to transfer the data directly from user space to the disk. Because the virtual memory addresses in user space can be mapped differently for different users, AdvFS cannot use the same mapping for I/Os started from different threads.

I/O consolidation is done in *consecutive_list_io()* when I/O is started and the buffers are moved onto the device queue (see section 8.4). To help maximize the possibility that I/Os can be consolidated, buffers are kept in ascending order by Logical Block Number (LBN) on the readyLazy and consol queues. The buffers are sorted as they are placed onto the readyLazy queue. This makes it easy to detect if two of the I/Os refer to contiguous blocks on the disk. There is no equivalent sorting of I/O requests on the blocking, UBC request, or flush queues.

Consolidation depends on special mapping routines, *bp_map_alloc()* and *list_map()*, provided by the Virtual Memory subsystem. These routines allow AdvFS to map the individual memory buffers for each of the individual I/O requests into a single virtual memory location. The resulting `ioDesc` structure is then passed to the device driver. Upon I/O completion (in *bs_osf_complete()*), the consolidated I/O structure must be examined, and each of the individual *`ioDesc`* structures are passed to *bs_io_complete()* if this represents the final I/O for this buffer.

8.7 Error Handling and I/O Retries

Errors may be returned from the device drivers to AdvFS through the `buf.b_flags` and `buf.b_eei` fields. If there is an error, B_ERROR is set in `buf.b_flags` and `buf.b_eei` contains a detailed error code. AdvFS will retry the I/O if the error is caused by temporary conditions at the device. This is done by *bs_osf_complete()* detecting an error that can be retried and sending a message of type RETRY_IO to the background I/O thread along with the `struct buf` from the original I/O. The background I/O thread again queues this `buf` onto the device queue after incrementing the retry statistics counters in the `ioDescT` and `vdT` structures. Retrying I/Os is disabled by default, but setting the sysconfig variable `AdvfsIORetryControl` to the maximum number of times that each I/O should be retried can enable it.

8.8 Relationship with UBC and Buffer Caching

When a file's page is in the buffer cache, AdvFS maintains a `bsBuf` structure that defines the attributes that are needed to maintain the page. There is a pointer to the `bsBuf` in the `vm_page.pg_opfs` field. In the `bsBuf`, there is a substructure called the `ioDescT`. This is the actual structure that is placed on the I/O queues and is responsible for mapping the buffer to the storage device.

Normally there is one ioDescT per bsBuf, but there are two exceptions:

1. When a file is being migrated and a page is in the buffer cache, there is a period of time during which the page has two ioDescT structures, one mapping the page to the location on the old volume and one mapping the page to the new volume. When I/O is started on this page, its I/O is not considered to be complete until the page has been written to both locations.

2. If direct I/O is enabled (see Section 8.12) and a direct I/O request exceeds the volume's maximum transfer size (vdT.max_iosize_wr), then the request is divided into a series of I/Os that do not exceed the maximum transfer size. This is done using a series of ioDescT structs associated with the bsBuf. For example, if a 4M direct I/O write is requested on a device with a 1M maximum transfer size, the *blkmap_direct()* routine builds four separate ioDescT structs and links them to the bsBuf. *bs_pinpg_direct()* adjusts the target addresses from the user's buffer so that each covers one quarter of the span in the original request. The bsBuf.ioCount is set to 4 to indicate that there are 4 I/Os that make up the transfer of this data. During I/O completion, *bs_osf_complete()* is called for these four I/Os. Each is handled normally, except that *bs_io_complete()* is not called until the last I/O for the bsBuf has completed (bp->ioCount == 0). Only then does *bs_io_complete()* get called, which wakes any threads waiting for this I/O to complete.

For cached I/O, each bsBuf represents a UBC page. The UBC page is held (the vm_page.pg_hold variable is greater than zero) from the time the page is pinned until it is unpinned. Holding a vm page prevents it from being removed from the cache. Staging an AdvFS page for I/O involves moving the UBC page from the dirty to the clean list, clearing the dirty flag, and setting the busy flag. This is done in AdvFS by calling the routine *advfs_page_busy()*. Buffers on the lazy queues are never staged for I/O, but buffers on the blocking, ubcreq, flush, and device queues are always staged for I/O. Buffers on the lazy queues are staged for I/O when they are moved onto the device queue.

From the AdvFS perspective, buffers cannot be repinned while they are staged for I/O. This may cause interthread interactions if one thread causes many buffers to be placed on the flush queue and another thread is trying to modify the data in those pages. In this case, the second thread will be forced to wait until the I/Os have been completed for the pages attempting to be modified. For more information on the UBC, see Chapter 7.

8.9 Smoothsync

Smoothsync is a mechanism whereby the dirty buffers are drained to disk at a constant, time-based rate. This is in contrast to the older mechanism where the update daemon caused all dirty buffers to be flushed to disk every 30 seconds, even though some of those buffers had been dirty for 30 seconds and some for only 1 second. Smoothsync works by assigning a time stamp to a buffer when it is dirtied, then moving the buffer to one of 16 smoothsync buckets. Which bucket the buffer is moved to depends on the current time and the value of the smsync_age variable. The default smsync_age of 30 seconds means that a dirty buffer is eligible to be flushed after it has been dirty for 30 seconds. Therefore, with this value, all buffers dirtied in any two-second interval are linked onto one smoothsync bucket. Buffers dirtied in the next two-second interval are placed onto the next bucket, and so on. (The mount command with the -o smsync2 option modifies this behavior (see Section 8.9.1.)

The smoothsync thread is responsible for removing the buffers from each bucket in a timely fashion. This wakes at one-second intervals and scans through all the mounted filesets. It calls VFS_SMOOTHSYNC() for each fileset, so the *msfs_smoothsync()* routine gets called for each AdvFS mounted fileset.

For each pass through all the mounted filesets, *smoothsync_thread()* passes a constant value for the bit at the SMSYNC_LATCH position in the `smsync_flag` parameter. This allows each filesystem to distinguish between successive passes through all the mounted filesets. The static variable `latch` in *msfs_smoothsync()* is used to track when smoothsync passes change. For example, if / and /usr are AdvFS-mounted filesets, on the first smoothsync pass, the value 0 is passed to *msfs_smoothsync()* for both the / and /usr filesets. On the next two calls to *msfs_smoothsync()* for / and /usr, the value SMSYNC_LATCH is passed. After that, the value for SMSYNC_LATCH bit alternates between 0 and 1 on successive passes. This provides a mechanism for the underlying filesystem to detect a new run on a set of mounted filesystems, since the value changes on each run. For instance, if there are ten AdvFS filesets mounted, *msfs_smoothsync()* will be called ten times during a single pass of the smoothsync thread through all of the mounted filesets on the system. AdvFS has a frequency counter that is maintained in *msfs_smoothsync()* and is updated only once per pass through all the filesets, so it uses the change of the SMSYNC_LATCH value to detect when this variable needs to be updated.

The *smoothsync_thread()* passes the current value of `smsync_age` to *msfs_smoothsync()* in the `sync_age` parameter. This enables *msfs_smoothsync()* to know that this is a typical call on behalf of the smoothsync thread. Occasionally, *msfs_smoothsync()* must do some special processing, which is called 'coarse granularity processing'. This processing includes flushing the log, flushing memory-mapped pages, and updating the last modified time for the disk. Coarse granularity processing may be requested by the `sync()` system call by passing a `sync_age` value of 0 to *msfs_smoothsync(),* or by the smoothsync thread when it passes a SMSYNC_PERIOD flag. In each case, *msfs_smoothsync()* will do the requested coarse granularity processing. By default, this coarse-granularity processing occurs every 30 seconds.

Even though *msfs_smoothsync()* is called every second, it might not run each time it is called. For example, because there are 16 smoothsync buckets, a `smsync_age` value of 30 means that a bucket needs to be flushed every other second to have them all flushed in 30 seconds. For a `smsync_age` of 60, a bucket needs to be flushed every 4 seconds. For a `smsync_age` of value of 15, a bucket must be flushed on each call. *msfs_smoothsync()* keeps a variable called `freq_cntr` to determine whether or not smoothsync flushing is required on this call. If this is a non-smoothed call (`sync_age` was passed as 0), the flushing must be done on this call.

The following table shows the relationship between values for three variables maintained in *msfs_smoothsync()*.

smsync_age (seconds)	smsync_period	smsync_step
5	1	5
10	1	10
15	1	15
20	2	10
30	2	15
35	3	11
40	3	13
50	4	12
60	4	15

`smsync_period` is the interval between AdvFS smoothsync runs. This is calculated based on the number of smoothsync buckets (16) and how long the `smsync_age` gives to get them all flushed. For a `smsync_age` of 30, `smsync_period` is 2 because all 16 buckets can be flushed in 30 seconds, if a bucket is flushed every two seconds. If `smsync_age` is 10, the `smsync_period` becomes 1 (second), and *msfs_smoothsync()* runs and flushes at least one bucket at each call, which happens at one second intervals.

`smsync_step` is a variable that is used to determine which smoothsync queue is appropriate for adding buffers with a given timestamp. It is calculated as (`smsync_age/smsync_period`), and is recalculated whenever `smsync_age` changes. This value is to decide which of the smoothsync queues to insert a buffer onto using the formula:

((timestamp / smsync_period) + smsync_step) % 16

This generates a bucket index (from 0 to 15) that is the appropriate smoothsync queue onto which a dirty buffer will be inserted when it is unpinned.

The `smoothsync_age` attribute is enabled when the system boots to multi-user mode and disabled when the system changes from multi-user mode to single-user mode. To change the value of the `smoothsync_age` attribute, edit the following lines in the `/etc/inittab` file:

```
smsync:23:wait:/sbin/sysconfig -r vfs smoothsync_age=30 > /dev/null 2>&1

smsyncS:Ss:wait:/sbin/sysconfig -r vfs smoothsync_age=0 > /dev/null 2>&1
```

The value of `smsync_age` can be modified with `dbx` or `sysconfig`. Change the sysconfig value for `smoothsync_age` (see 8.11) to modify the internal `smsync_age` variable. If `smoothsync_age` is set to 0, smoothsync is disabled and dirty page flushing is controlled by the `update` daemon at 30-second intervals.

To get the current `smoothsync_age` value, enter:

```
sysconfig –q vfs | grep smoothsync_age
```

To reset the value, enter:

```
sysconfig -r vfs smoothsync_age=15
```

`smoothsync_age` has a limit of 60 seconds if you are using sysconfig. If you want it to be greater, use `dbx`.

8.9.1 Modifying smoothsync Behavior (`-o smsync2`)

The mount option, `-o smsync2`, modifies the way smoothsync works for a fileset (and domain). When this option is enabled, the timestamp on each buffer is reset every time the buffer is modified. By default, the timestamp is set only the first time the buffer is modified. The buffer remains on the smoothsync queues until the value of (`timestamp + smoothsync_age`) has expired. Then it may be moved down the I/O queues and scheduled for I/O. Therefore, enabling the `smsync2` option tends to flush idle buffers, while active buffers tend to remain on the smoothsync queues indefinitely. Be aware that keeping the buffers on the smoothsync queues indefinitely prevents the typical flushing of the file buffers every few seconds but may give better performance for an application that constantly refers to the same set of file pages.

If you enable the `smsync2` option on a mount point in a domain, the alternate smooth sync policy goes into effect for all the filesets in the domain. It remains in effect until all of the filesets in that domain are unmounted.

8.10 Load balancing

If a domain has more than one storage device, load balancing ensures that a specific device does not become a bottleneck while another is underutilized. You can choose to spread the files out over the existing resources dynamically with the `vfast` utility (see section 13.6) or by changing system parameters.

AdvFS has several utilities that monitor volume activity in a domain. The `advstat -v [1 - 3]` command examines the volume statistics. The more generic `iostat` utility can be used to compare I/O rates between specific devices. To determine if there are specific hot files that are responsible for a load imbalance, use the `vfast` utility: `vfast -l hotfiles <domain>` to print a report of the most actively paging files and the volumes on which they reside. If a given file is known to be responsible for a serious load imbalance, that file can be migrated to a volume that has less I/O traffic.

8.11 Tuning the I/O Subsystem

The `smoothsync_age` VFS attribute specifies the amount of time, in seconds, that a modified page ages before becoming eligible for the smoothsync mechanism to flush it to disk. This value defaults to 30 seconds, but can be set from 0 to 60 seconds. Setting `smoothsync_age` to 0 sends data to the ready queue every 30 seconds, regardless of how long the data has been cached. Increasing the value increases the chance of lost data if the system crashes, but can decrease net I/O load (improve performance) by allowing the dirty pages to remain cached longer. This value is applied to all devices on the system. See section 8.9 for more information on smoothsync.

The `AdvfsReadyQLim` AdvFS attribute specifies the size limit of the ready queue. This defaults to 16k blocks (1024 pages) and can be set from 0 to 32k blocks. As long as smoothsync is enabled, there is no reason to change this attribute. If smoothsync is disabled, then increasing this value increases the likelihood of I/O consolidation, but it also increases the overhead needed to sort the buffers onto the ready queue. `AdvfsReadyQLim` is a global variable and affects all volumes on the system. The variable can

be overridden on a per-volume basis using the `chvol -t` command. This command allows the user to specify the maximum size of the ready queue for a given device. To allow for optimal I/O consolidation, try to keep the ready queue limit larger than the device's preferred transfer size.

The device's preferred I/O transfer size may be overridden using the `chvol -r or chvol -w` command. Making this value smaller causes AdvFS to issue I/O requests to that device in smaller chunks, breaking large read/write requests down into a series of smaller requests. This may add overhead to handle all the I/O requests. Making this value larger causes AdvFS to issue larger transfers when possible. If the device can handle the larger transfers efficiently, then this is likely to improve throughput. However, making this value too large can cause the device to handle the transfers inefficiently.

8.12 Direct I/O

Direct I/O is a method of I/O that allows applications to read and write data directly from disk into their application buffers by bypassing the file system's buffer cache. This can be thought of as a replacement for the use of raw partitions that retains ease of administration (file naming, copy, backup, etc). The ultimate purpose of direct I/O is to improve the performance for applications that do not depend on the filesystem's caching of file pages. Such applications do not access the same block frequently and usually perform large I/O writes.

There is more information on the use of direct I/O from the application perspective in Section 4.6.2 where the O_DIRECT I/O `open()` parameter is discussed.

AdvFS uses the following internal rules for direct I/O:

1. Changing the file-open mode MUST be done with the `file_lock` seized. This holds for setting direct I/O, data logging, mmapping, and caching (the `file_lock` is discussed in section 8.12.1).

2. Once an I/O of a given type has been started (direct I/O, cached, mmapped), modifying the file's open mode must not interfere with that I/O's completion.

3. On read or write, once the open mode has been determined to be direct I/O, the `file_lock` may NOT be released until the active range structure covering that I/O is on the file's active range list (active ranges are discussed in section 8.12.2).

4. Simultaneous I/O within the same active range by direct I/O and migrate is not allowed and is prevented. Successive threads will sleep waiting for an overlapping active range to be released by the thread actively doing I/O. The completion of the last I/O to any active range awakens any threads waiting for the range to be completed. In the future, we may, if possible, relax this restriction with respect to simultaneous direct I/O readers in the same range, but specifically block threads doing writes, migrates, truncates, or page invalidation in the same range.

8.12.1 Use of the file lock

The `file_lock` in the AdvFS context structure is typically used to synchronize threads that are concurrently manipulating the same file. One example is the use of the file lock to coordinate threads that have the file opened for O_APPEND mode. In this case, the lock guards the `bfAccessT.filesize` field, and this field is used to determine which file offset each thread uses to append its data.

The direct I/O read and write paths explicitly drop the `file_lock` to allow multiple threads to read and write data to the same file concurrently. This is very useful to databases, which have many threads

concurrently manipulating different regions of the same database file. Because the file lock has been dropped, active ranges are used to synchronize I/O with finer granularity.

8.12.2 Use of active ranges

To synchronize threads that run concurrently on different parts of the same file, AdvFS uses active ranges. These ranges cover file blocks (not pages) and are mutually exclusive, meaning that there cannot be two active ranges on a file that overlap. Typically a thread sets an active range while the file lock is held and then drops the file lock. Paths that currently use active ranges for synchronization are

- direct I/O reads and writes
- truncate
- migration
- CFS calls to *msfs_flush_and_invalidate()*.

The active ranges are used in migration to prevent migration code from bringing a page into the buffer cache at the same time a direct I/O write is trying to remove it from the cache. The truncate path also takes an active range from the truncation point to infinity to keep any direct I/O threads out of that region until the truncation has completed. The CFS calls to *msfs_flush_and_invalidate()* use the range lock to synchronize with any migrate that is running on the server.

There is a special case active range that is used when a file is opened for direct I/O and for O_APPEND. In this case, an active range is seized from the current end-of-file to infinity to prevent another O_APPEND thread or a truncate thread from concurrently modifying the end-of-file while the original append is in progress. This serializes these threads so be aware that opening files for O_APPEND and O_DIRECTIO concurrently does not give optimal throughput or scalability.

The active range algorithms cause a thread that is attempting to take out a range that overlaps an existing range to sleep until non-overlapping ranges are available. There is also a fairness aspect to the algorithm that prevents threads taking out small ranges from starving out a thread that is waiting for a large range.

8.12.3 Interaction with cached pages

By definition, direct I/O bypasses the buffer cache. This means that

- direct I/O does not perform any prefetch or read-ahead operations to prime the cache.
- Any data already in cached pages must be handled to avoid data corruption. How this is handled is described below.

It is tempting to think that AdvFS can eliminate the overhead of the direct I/O code that handles a page already in the buffer cache by merely flushing and invalidating all cached pages when the file is initially opened for direct I/O. However, this cannot be done so long as AdvFS allows files that are opened for direct I/O to be migrated. File migration occurs by bringing the pages of a file into the cache, and then writing them out to the newly mapped locations. The pages that remain in the cache following migration must be handled by any subsequent direct I/O read or write.

What happens to pages that are already in the cache when a direct I/O request is made?

- During a direct I/O read, if a page is in the cache, AdvFS returns the data to the user from the cache, and there is no invalidation of the page (the page remains in the UBC). This makes the read faster because there is no need for a disk access.

- When there is a write to a page that is in the cache, AdvFS must avoid losing or corrupting the data that is in the cached page. If the cached page is going to be completely overwritten by the new data, it is merely invalidated (removed from the cache) and then the write to underlying storage is done with direct I/O. If the page is being partially overwritten, the application data is merged into the cached page, the page is flushed to disk, and then the page is invalidated (removed from the UBC).

There are several code paths that can bring pages into the cache for a file opened for direct I/O:

- When an application is extending a file or adding storage in a hole, the storage is first allocated at the top of *fs_write()*. The storage-allocation routines call *fs_zero_fill_pages()* that does special processing for pages that are to be partially overwritten on this write request. Any 8K page that is not completely overwritten must be zeroed and have the new data merged with the page before being flushed to disk. This happens at the beginning or end of a range to be written. Therefore, for partial writes, direct I/O takes advantage of the fact that *fs_zero_fill_pages(),*pins the underlying page (dragging it into the UBC), zeroes it, and then unpins that page lazily. This newly zeroed page is then repinned in *fs_write_direct(),* the new data is merged, the page is flushed to disk, and then the page is invalidated. Testing has shown that using temporary UBC caching yields a 30 to 40% performance gain during file extension, as compared to maintaining special-case code.

- The truncation code in *fs_setattr()* may cache a page for a file opened for direct I/O. If a truncation causes the end-of-file to fall in the middle of an underlying page, that page is pinned, the bytes beyond the new end-of-file are zeroed, and the page is unpinned. If this page is in a hole, the pin code causes a page marked UNMAPPED to be left in the cache. Trying to reference this page results in the E_PAGE_NOT_MAPPED error, which needs to be handled correctly by the routines that could be referencing or pinning the page.

- A file opened for direct I/O can have pages brought into the cache by the copy-on-write (COW) code, which is done in *bs_cow_pg()*. When a file with a cloned fileset is modified, the old data must be copied to the clone before the modification can be performed. This routine references the original page (bringing it into the cache), pins the analogous page in the cloned file, and copies the data from the original to the clone. This is done prior to modifying the original file's data. Once this is done, the direct I/O code detects that the page is in the UBC cache and invalidates it prior to writing the data to disk. All this is done atomically (to outside readers and writers) under protection of a range lock. So that all COW'd pages are invalidated, the COW-ahead code in *bs_cow_page()* is inhibited, and only one page is COW'd at a time if the original file is opened for direct I/O. This assures that only pages protected by the current range lock can be brought into the UBC during the COW operation.

In the direct I/O write path there is considerable code devoted to serializing threads that have separate active ranges that all lie on the same underlying 8k page. These threads race with each other to flush the data to disk and invalidate the page. If not correctly serialized, one thread will be trying to invalidate the page while the others are trying to reference or pin the page. Contention is handled by marking the `bsBuf` as DIO_REMOVING by the first thread that discovers that this page is dirty and needs to be flushed and removed from the cache. Any subsequent threads, on checking to see if this page is cached, see this flag, sleep until the page has been flushed to disk, and then return to the caller with a status reflecting the fact that the page is not in the cache. Ideally, the DIO_REMOVING would not be removed until after the page has been invalidated, but this is not possible because of UBC-AdvFS interactions in handling `bsBuf` and UBC page structures. This means that there is a period after the page is flushed to disk (clearing DIO_REMOVING on I/O completion) and before the page is actually invalidated during which this page can be found in the UBC cache. Threads that look up and find this now clean page in the

UBC attempt to invalidate the page. This means that direct I/O writers to a clean page in the UBC all race to invalidate the page. This race is somewhat inefficient, but it is benign. It only happens the first time a dirty page is found in the cache by direct I/O threads all writing to the same page, so this is a fairly uncommon code path.

When the direct I/O code calls *ubc_lookup()* to see if a page is already in the cache, it passes the B_READ flag. This is done so that the UBC code does not mark the page dirty if it is found in the cache. This also allows *ubc_lookup()* to pass back the B_DIRTY flag if the page in the cache is already dirty. The direct I/O write path must flush this page before invalidating it if it is already dirty. Otherwise, it can just invalidate a clean page.

8.12.4 Block alignment

The direct I/O code needs to know the underlying block size (sector size) of the disk being written or read. This is because AdvFS MUST present data to the driver in buffers that are an exact multiple of the block size because it will use Direct Memory Access (DMA) to transfer this data directly to disk. Because the user can request a data transfer of any size, if the request is neither aligned on a block boundary, nor an even multiple of the block size, then AdvFS must adjust for this.

If the application presents data that is correctly aligned and sized, the underlying storage is already allocated, and there is no cached page, AdvFS can execute a minimal code path to do the I/O. Basically, this means that AdvFS wires down the user's buffer pages, builds a `bsBuf` that maps the user's buffer to the underlying storage, and puts the `bsBuf` onto the AdvFS blocking queue. From there, the I/O subsystem hands the pages off to the device driver to do the transfer.

If the data is not aligned or correctly sized, AdvFS must deal with partial block sizes on the read, and partial page sizes on the writes. If a read doesn't begin on a disk sector boundary, a disk read of the entire disk sector is performed using a temporary buffer. The requested data is then copied to the user's buffer. An aligned transfer needs no intermediate buffer allocation or data copy. Similarly, any transfer of less than a full sector in length also requires an intermediate copy to a temporary buffer. This is because some drivers, including LSM, do not handle less than full sector-sized transfers.

Unaligned writes are slightly more complicated than unaligned reads; writes that need to allocate underlying file storage also fall into this category. (Requests that are aligned and already have file storage fall into the minimal code path discussed above.) A direct I/O write can require up to three steps.

1. If the I/O has an unaligned write in the first page, an 8K page is malloc'd, read in from disk, merged with the application data, and written synchronously to disk. Any error is returned to the caller.

2. If the end of the transfer does not end on a page boundary, the same set of steps is followed (malloc, read, merge, write).

3. The unaligned portions are now accounted for, and the rest of the transfer will be aligned on the underlying sectors. Therefore, AdvFS can simply hand the write for the remaining pages to the disk driver.

8.12.5 Interaction with file fragments (frags)

If a file has a frag (see 6.2) and AdvFS detects that a write will be done to the fragged region, the fragged page is brought into the file's normal extent map prior to the data transfer. AdvFS never does direct I/O to pages in the frag file.

When reading a fragged file with direct I/O, there is no attempt to bring the frag into the file's normal extent map. The code in *fs_read()* requests that all pages prior to the frag be read with *fs_read_direct()*. When only the frag data remains to be read, *fs_read()* calls *uiomove_frag()* to transfer the data from the frag page (still in the frag file) to the user's buffer.

8.12.6 Transfer size considerations

There is no explicit limit on the size of a direct I/O transfer, but moderation is the best guideline. A lot of small transfers take longer than a few larger ones. For example, writing a 100M file in 1M increments is generally faster than writing the same file in 1K increments. However, making the application's I/O buffer too large can start to slow things down. For example, writing the 100M file in a single 100M transfer may not produce the best results for two reasons:

1. The pages in the application's I/O buffer must be wired before the DMA transfer can take place. Wiring 100M of pages on a small-memory system may constrain the memory resources available to concurrently running threads. In addition, although an application can malloc and write data to a 1Gb virtual memory space, wiring all those pages on a system with only 512M of physical memory does not give good results and may even hang that thread.

2. The write may have to be broken down into a series of transfers that can be managed by the disk driver. Although large transfers can be requested, AdvFS breaks the request into a series of smaller transfers if it exceeds the underlying disk's `maximum_transfer_size`. This is transparent to the application. To determine a disk's maximum transfer size, use the `chvol -l <volume> <domain>` command and observe the values displayed after "`max =`". This value is expressed in 512-byte blocks, so multiply by 512 to get the transfer size in bytes. Note that direct I/O scales to the device's maximum value, not the value that is resettable with `chvol -r` or `chvol -w`. Currently there is no way to override this transfer size value.

   For example, if a volume has a maximum transfer size of 1M, and a file on that disk is written using direct I/O in 4M requests, each request is broken down into a series of four 1M transfers to the disk. This is accomplished by allocating four `ioDescT` structures for the one `bsBuf` that describes the application's data buffer.

8.12.7 Mitigating the synchronous nature of direct I/O

Direct I/O transfers are, by default, handled synchronously (see 8.1), meaning that the *write()* call does not return until the data is on disk. This differs from cached calls that guarantee only that the data has been written to the filesystem's buffer cache, not to disk. Because this synchronous behavior can cause performance problems, the application has the option of using the slightly more complicated Asynchronous I/O (AIO) Interface. This allows the application to queue up a series of I/Os and to check their status at a later time. The AIO package is a software layer between the application and the kernel that allows the application to deal with synchronous I/Os in an asynchronous fashion.

Because of this asynchronous behavior, AdvFS must make several changes when the call has been made through the AIO interface. The steps are as follows:

1. The code detects that it has been called through AIO because the value in `uio->uio_rw` is UIO_ AIORW instead of UIO_WRITE.

2. The `vnode.v_numoutput` field must be decremented after I/O has completed if this was an *aio_write()* request.

3. The AIO I/O completion routine must be called by the AdvFS I/O completion code.

4. The active range that covered the blocks being written is removed. This is typically done in *bs_osf_complete()* as the I/O is completed, but there is similar code in *fs_write_direct()* if an asynchronous I/O has not been started or an error path is taken.

Active ranges are more effective than simple and complex locks for controlling file ranges. The kernel cannot return to the user with locks held, but when AIO is involved, AdvFS must return to the user with some kind of locking on the file range while the I/O completes. The active range serves this purpose well because it is not a formal lock and can be released by the I/O completion code.

See Section 14.2 for more information on the AIO interface, and some considerations when modifying code in this path.

8.12.8 I/O consolidation

AdvFS does not consolidate any I/Os associated with direct I/O. This is because the driver is going to use Direct Memory Access (DMA) to transfer the data directly from user space to the disk. Because the virtual memory addresses in user space can be mapped differently for different users, AdvFS cannot use the same mapping for I/Os started from different threads.

9 Transaction Management

AdvFS uses transactions to guarantee the integrity of metadata on disk. For performance reasons, AdvFS does not attempt to guarantee the integrity of all user data. However, at the user's request, the filesystem is able to log all user and metadata (see Section 8.2).

Data integrity in AdvFS is provided through three cooperative subsystems: the transaction manager, the logging subsystem, and the buffer cache.

One of the requirements of the AdvFS filesystem is that the overhead associated with the transaction manager be kept to a minimum. For this reason, the transaction manager is referred to as the **flyweight transaction manager**. The term *flyweight transaction* is often abbreviated as FTX. (Since there is only one transaction manager in AdvFS, the term 'flyweight' will be omitted throughout the rest of this chapter for brevity). The transaction manager employs ideas from database theory to provide **transaction primitives** that are used to make **non-atomic operations** on the filesystem occur in a way that will be **atomic** (all-or-nothing). The **transaction log** is used to store transaction records that can be used to either redo or undo any operations under transaction control. A single transaction log exists per domain, and is accessed through the **logging subsystem**. The logging subsystem (also called the **logger**) provides a set of primitives that can be used to write to and read from a log file. The buffer cache (see Chapter 7) interacts with the logger primarily via the write-ahead log rule, which will be discussed shortly. The transaction manager, the buffer cache, and the logger collaborate to ensure that basic transaction rules are maintained. These rules will be discussed next.

9.1 Transaction Basics

A **transaction** is a collection of operations that perform a single logical function. In the context of AdvFS, the logical operation may be the addition of storage to a file, while the collection of operations might include allocating a new mcell for the file and updating several pages of the SBM.

A transaction is a key concept that allows a sequence of primitive storage updates to be linked together into a single all-or-nothing **atomic action**. Transactions are necessary to reliably update persistent storage structures of any significant complexity with high performance, yet still be assured of the ability to completely recover from system and media failures.

AdvFS's transaction manager maintains transaction state in a hierarchical scheme. The first transaction begun for a set of related operations that must be completed atomically is known as the **root transaction**. All transactions that comprise smaller units of work within that transaction are called **subtransactions**. A more in-depth discussion of root transactions and subtransactions occurs later in this chapter.

9.1.1 Generic rules to maintain data consistency

In order to guarantee that the logical function being performed maintains consistency in the system, each transaction must follow the ACID properties. These are generic transaction rules, and are not limited to the AdvFS transaction manager implementation.

- **Atomicity** – The actions that a transaction performs must be atomic; either none of the transaction is done or the entire transaction is done. The transaction system must ensure that if a failure occurs in the middle of a transaction, all steps done before the failure are backed out in such a way that the

system is left in the same state as before the transaction was started. AdvFS implements this property through redo/undo transaction records that are recorded in the log. In the event that a transaction cannot complete, these records are used to ensure that all steps done before the failure are backed out.

- **Consistency**—Execution of a transaction in isolation preserves the consistency of the database. Each transaction must be serializable in such a way that if transaction A commits before transaction B then the system will never be in a state where the effects of B are seen and the effects of A are not. This is a fundamental issue in AdvFS and is maintained through locks at a file level. Because transactions must be serializable, all changes made within a transaction tree must be single threaded; two threads cannot both make changes that are recorded under the same transaction.
- **Isolation**—The effects of a transaction must not be seen by other threads until the transaction is either committed or aborted. Like consistency, this is a matter of correct locking.
- **Durability**—Once a transaction has completed (been committed), the transaction system must ensure that the actions done by the transaction are guaranteed not to disappear. AdvFS uses a **lazy commit** model in which it only guarantees durability after the transaction is committed to disk. A portion of the log is kept in memory. Until the portion of the log containing the transaction has been flushed to disk, that transaction may not be redone in the event of a system crash. This is acceptable only because of the write-ahead log rule which is discussed below.

In addition to the ACID rules, there is one other property of transactions that must be considered.

- **Idempotence** — This is a property of an operation that allows it to be repeated many times on a given object and will either transform it to the desired state, or will leave it in the desired state. This is often important in redo operations during recovery. If certain operations are redone during recovery, and the system crashes in the middle of recovery, and recovery has to be done yet again, the redo operations must be able to leave the transaction in the desired state, no matter how many times the recovery is run.

9.1.2 AdvFS-specific Transaction Rules

The transaction manager, the buffer cache, and the logger collaborate to ensure that the ACID properties are maintained. The most important interaction between these three subsystems is to maintain the **write-ahead log rule**. This rule states that:

> A modification to metadata made under a transaction must not be flushed to disk before the corresponding transaction records are written to the transaction log and flushed to disk. In other words, the transaction log must be written ahead of the actual metadata.

This rule ensures that AdvFS can always use the transaction log to create a consistent view of its metadata. In the event of a system failure, only changes made under a transaction whose records are on disk could have been written to disk. Therefore, only changes with associated on-disk log records will need to be redone or undone. If a metadata change is made, but the transaction log record is not written to disk, then the change exists only in memory and does not need to be undone following a system crash. Likewise, any transactions that have been written to disk will have all the information required to redo or undo the operations on the metadata and restore the metadata to a consistent view. Maintenance of the metadata pages in relationship with this rule is discussed in Section 9.2.10.

There are several additional rules to consider when using AdvFS transactions.

- There are limits on the number of pinned pages per transaction. The limits are defined in `ftx_public.h` and `ftx_privates.h`. One limit is defined by FTX_TOTAL_PINS (currently

10). This represents the total number of pages that can be pinned by all active subtransactions within the same root transaction at one time. As subtransactions finish, their pages are unpinned and subtracted from this total count. Both a root transaction and subtransaction can pin the same page, and this will only count as one page pinned in this total. Another limit is FTX_MX_PINP (currently 7), which places a limit on the number of pages that can be pinned at a given subtransaction level.

- There are limits on the number of records that can be pinned at one time within each page at each subtransaction level. The limit is defined by FTX_MX_PINR (currently 7).
- There are limits on the number of nested subtransactions that can be created. The longest path from a subtransaction to the root transaction can be no more than FTX_MX_NEST (currently 11) levels.
- Once a subtransaction pins a record, that subtransaction may not fail. AdvFS only records after-images of transactions (a binary copy of what the record looks like after being changed). As a result, a subtransaction that has pinned records can not be failed because it can not be undone. See the discussion of undo and redo records for more information. A subtransaction must always have an undo record in case its parent transaction should fail.
- A thread must never start a root transaction if it already has a root transaction started; the system will likely hang. Also, a thread that has started a root transaction must not wait for another thread that also might start a root transaction; this mostly applies to DMAPI which calls out of the kernel to DMAPI applications that may call back into the file system and possibly start another root transaction. See section 9.2.9 on transaction slots for a better understanding of why the system may hang.
- Undo and Redo operations should be logical rather than physical. This means that the records should redo and undo operations logically, not by simply recording before images or after images of the operation being performed.

9.1.3 Transaction Terminology

Root Transactions and Subtransactions

The first transaction begun for a set of related operations that must be completed atomically is known as the **root transaction**. All transactions that comprise smaller units of work within that transaction are called **subtransactions**.

Before and After Images

The **before image** of a datum is its value before it has been modified by the transaction. Its **after image** is its value after it has been modified. Undo operations typically restore a before image, and redo operations typically restore after images. Consider a transaction-based banking system used to withdraw $100 from a checking account. If we start with $300, and remove $100, the before image value of our account balance is $300, and the after image is $200.

Undo and Redo Operations

An undo operation ensures that any data changes made during the transaction are set back to their values before the transaction started. Redo operations ensure that all data changes made during a committed transaction are present in non-volatile storage. Consider again a banking example where we will transfer $100 from our bank account that contains $300 to another account that contains $500. If the transaction removes $100 from our account, but finds the other account is no longer valid, an undo operation ‘undoes’ the withdrawal from our account, essentially adding the $100 back so that we still have the $300

we started with. Conversely, if the system crashes after removing the $100 from our account but before it has been added to the other account, a redo operation after reboot will ensure that the $100 is properly added to the other account.

Logical and Physical operations

Undo and redo operations may be accomplished via logical or physical operations. Logical operations take the current data and make it correct via some logical sequence of steps (addition, subtraction, etc). Physical operations merely overlay the current data with the correct data. Each method has its advantages, and each must conform to the broad transaction rules specified earlier (Section 9.1.1). Let's consider the previous banking example again. The transaction which removed $100 from our bank account logically subtracted $100 from our balance. So the logical operation to undo this transaction would be to add $100 to the balance. A physical operation to undo this transaction would be to remember that the before image of our bank account was $300. However, AdvFS does not save before images, so any undo operation will always be a logical undo. Before images are not saved due to performance reasons.

Transaction Slot

Each root transaction is assigned a slot in the domain's transaction table. The transaction table is an in-memory structure and has a fixed number of slots. Only root transactions are assigned to slots. See Section 9.2.9 for more details.

Global Pin Table and Per-level Pin Table

The global pin table (`ftx.PinTbl[]`) is a structure used to monitor the pages currently pinned by the current transaction. So the term 'global' here is used to mean 'global to the transaction', and refers to pages pinned at all levels by the current subtransactions. There is also a per-level pin table (`ftx.cLvl[].lvlPinTbl[]`) that keeps track of pages pinned within the current level (subtransaction) of the transaction. These tables are of fixed size, such that there can be at most 10 pages pinned per transaction (size of the global pin table) at one time, and at most 7 pages pinned at one time in each subtransaction level (size of the per-level pin table). See Section 9.5.1 for more information on the structures for these tables.

Undo Record

An undo record for a transaction is any structure that contains information necessary to logically undo the operation under transaction control. For example, if an mcell were to be modified, the undo record may be a structure that maintains the `mcellId` and information about what data was changed. Undo records are written into the log as an array of bytes; it is up to the undo routines to correctly cast the byte array back into an appropriate structure.

Redo Record

A redo record for a transaction works like an undo record, but contains information to redo an operation under transaction control rather than to undo it. This could be a simple byte image of the final data being changed.

Root Done Record

A root done record is a structure that contains needed information to take final actions on behalf of a transaction at the point when the root transaction is committed. For example, if a subtransaction is making a change that should not be visible until the root transaction is committed, it may register a root done record with the transaction. At root commit time, each root done record will be processed and a call

back to a root done function for the subtransaction will be made using the root done structure as a parameter. This allows a subtransaction to postpone resource release until root transaction commit.

Transaction Continuation Record

A transaction continuation record is a structure that allows one transaction to establish a set of follow-on operations that will be done after the current transaction is committed. This is sometimes used to break a long-lived transaction down into a series of smaller transactions.

The Agent Concept

Each subtransaction is associated with an **agent**. The agent helps to identify what the transaction was doing, or had done, in the event of a recovery. Since we are storing log information on disk, and we are not guaranteed that from one boot to the next a function pointer will reference the same function, we can not store undo routine references on disk. Instead, the concept of an agent is used. In the kernel, each agent is assigned an **Id**, an undo routine, a redo routine, and a root done function pointer. At recovery time, the `agentId` can be looked up in the kernel to find the correct function to use for undo/redo. A transaction's agentId is an index into FtxAgents[].

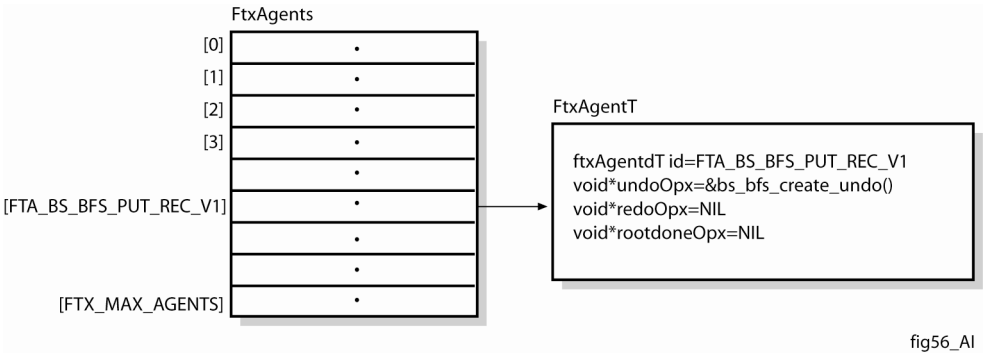


Figure 33: FtxAgent structures

Whenever a subtransaction or a root transaction are started, an `ftx_agent` must be identified. The `agentId` is then stored in the log records associated with that transaction.

9.1.4 Introductory Transaction Example

Below is a simplified version of what happens when a file is created under AdvFS's transaction control.

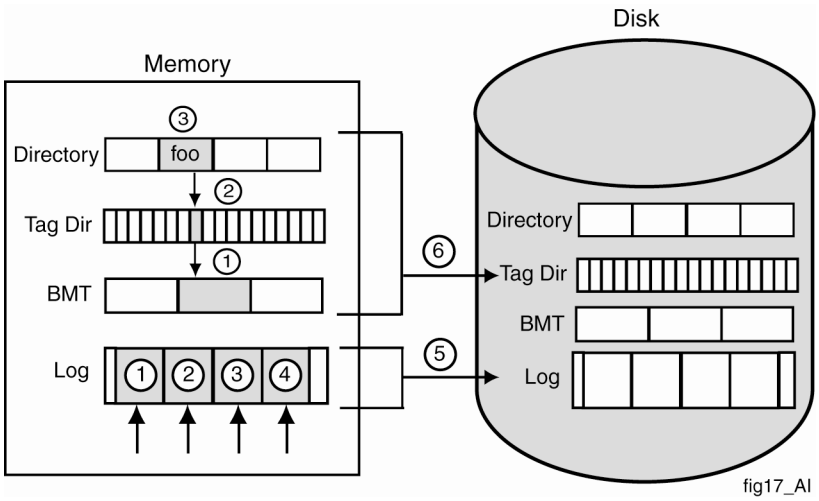


Figure 34: Creating a new file under transaction control

Step 1: Storage allocated within the BMT to hold the new bitfile's attributes. The changes to the affected mcell are written to the in-memory transaction log as log record "1".

Step 2: A bitfile tag slot is allocated and modified to reference the newly initialized mcell from step 1. The changes to the tag slot in the tag directory are written as log record "2".

Step 3: A file system directory entry is created with the specified file name ("foo") to reference the bitfile tag created in step 2. The directory changes are written as log record "3".

Step 4: The entire transaction is committed and a final log record is written as record "4" to indicate the commit.

Note that at this point all of the above changes are in-memory only, so if the system crashes it would be as if the transaction never occurred and the file had never been created (which leaves all the metadata structures consistent).

Step 5: If the file is to be synchronously created, then the in-memory log is flushed to disk. Otherwise, log records are buffered in memory until multiple log pages can be written to disk in a single large I/O to improve performance.

If the system fails at this point the log would be used to redo the changes done in steps 1-3 when the domain is recovered.

Step 6: Once the log records are flushed to disk the buffered metadata pages modified in steps 1-3 can be written to disk. Once the metadata changes are written to disk, the log records describing the changes in steps 1-3 are no longer needed.

9.2 AdvFS Transaction Management

Transaction primitives provide the necessary tools to create root transactions and subtransactions, to register undo and redo operations for those transactions and to fail or complete transactions as appropriate.

9.2.1 Overview of Transaction Primitives

ftx_start (parentFtxH) - Starts a transaction, returns a transaction handle. If the parent transaction handle is NULL then a root transaction is started, otherwise a subtransaction is started.

ftx_done (ftxH, agentId, undoArgs, rootDoneArgs) - Commits a transaction; optionally passing logical undo or root done information.

ftx_fail (ftxH) - Fails a transaction; subtransactions that have been completed are backed out. Must not be called after `rbf_pin_record()` has been used in the current (sub)transaction.

ftx_lock_* (ftxH, lock) - Acquires a lock and associates it with the (sub)transaction at the current level. When that (sub)transaction is done, the lock is automatically released once the associated log record is written to the log. There is also a function to add an already-acquired lock to the current level.

rbf_pingpg (bfH, page) - This routine is like a "begin to modify" function. It pins the specified bitfile page in the buffer cache (preventing it from being written to disk) and tells the buffer cache that the thread intends to modify the page as part of the transaction.

rbf_pin_record (pgH, recAddress, recSize) - This routine is called just before modifying a byte range in a page (must call `rbf_pingpg()` first to obtain the page handle (***pgH***)). It allows the transaction manager to capture the byte-level modification for use in the redo pass of recovery.

ftx_register_agent () -The code that does the recoverable modification is called a *transaction agent* which must be registered with the transaction manager. The main purpose of this registration is to identify the routines for undo and root done functions.

The basic model is to use the primitives to perform some atomic and recoverable modification. If the modification can be done in a root transaction without subtransactions (also called a flat transaction) then no undo action is needed. This is because for a root transaction all modifications are in-memory and nothing is logged until *ftx_done()* is called which commits the transaction (so only redo information is associated with the transaction so that it can be rolled forward during recovery).

AdvFS also supports subtransactions. If a recoverable modification requires subtransactions then the subtransactions generally need to provide undo actions; these are defined by a separate undo routine.

9.2.2 Transaction start: ftx_start()

The *ftx_start()* primitive comes in many flavors all of which call *_ftx_start_i()* to do the actual work of starting a transaction. *_ftx_start_i()* is the primitive used to start a transaction or subtransaction. If *_ftx_start_i()* is called with no parent transaction handle, a new root transaction is created, causing a transaction slot in the domain's transaction table to be utilized. If a root transaction handle is specified, a new subtransaction is created.

The parameters of *ftx_start_i* are:

- *`ftxHT *ftxH`* – the transaction handle of the transaction to be started. This pointer is passed in and initialized during the *`_ftx_start_i()`* routine. This handle will contain an index value into the *`domainT.ftxTbld.tablep`*. However, the *`hndl`* value stored in *`ftxHT`* is one-based, so *`hndl-1`* is the slot this transaction handle references.
- *`ftxHT parentFtxH`* –the transaction handle for the parent transaction, if any. If this is *`FtxNilFtxH`*, a new root transaction is allocated, otherwise, *`ftxH`* becomes a subtransaction of the *`parentFtxH`*

- `ftxAgentIdT agentId`—The agent ID for this transaction. This defines the undo routine, the redo routine and the rootdone routine for this transaction.
- `domainT dmnP` –This field specifies the domain in which this transaction is occurring and in which this transaction will be tracked. It also provides a means of getting the `ftx` structures for the parent transaction.
- `int page_reservation`—currently unused.
- `unsigned int atomicRPass`—recovery is done in several stages, this defines the **recovery pass** in which this transaction should be processed.
- `int flag`—used to pass in flags to the transaction. Currently a flag value of 1 indicates an exclusive transaction, meaning that no other transactions on the domain can be executing in parallel with this transaction.
- `long xid`—a CFS generated transaction id.

_ftx_start_i() will take different actions depending on whether the transaction to be started is a root transaction (`parentFtxH = FtxNilFtxH`) or a subtransaction (`parentFtxH != FtxNilFtxH`). To start a root transaction, *_ftx_start_i()* must acquire a free slot from the `domainT's ftxTbld` structure. If the transaction is specified as exclusive, *_ftx_start_i()* must wait for all other transactions on the domain to complete before reserving a slot. Slots are acquired in FIFO order to preserve fairness among multiple threads.

Once a root transaction has a slot, an `ftxStateT` structure is allocated and initialized to maintain state information for the entire transaction, including future subtransactions. Finally, a `perlvlT` structure is allocated and initialized to track pages pinned only at this level of the transaction (level 0 for root transaction).

For a subtransaction, the current transaction level is incremented in the `ftxStateT` structure, and a new `perlvlT` structure is allocated and initialized to track the pages pinned in this subtransaction.

The initialization of the `perlvlT` structure is done in a common code block for both the root and subtransactions. *_ftx_start_i()* finishes by initializing the return `ftxH` parameter to contain a handle to the transaction that was begun.

9.2.3 Transaction Commit: ftx_done()

The primary goal of this routine is to commit a (sub)transaction by writing the transaction records to a pinned log page in memory; there is no responsibility by this routine to ensure that the records are written to disk. Like *ftx_start()*, *ftx_done()* comes in many flavors all of which call *ftx_done_urdr()* to perform the actual work. *ftx_done_urdr()* (short for *ftx_done_undo_rootdone_redo*) is the primitive used to declare that a transaction is complete and should be committed to the log. All other versions of *ftx_done_**() are wrappers around *ftx_done_urdr()*.

Once *ftx_done()* has been called and the transaction log is written to disk, this transaction will be recovered in the event of a system failure. After *ftx_done()* is called on a root transaction, and the transaction record has been written to disk, if the system crashes prior to the logged operations being completed, the entire transaction tree (root and all subtransactions) will be redone when the domain is recovered. If, however, *ftx_done()* is called on a subtransaction and the transaction record for the

subtransaction is pushed out to disk and the system fails prior to the root transaction having been committed, when the domain is recovered, the subtransactions will be undone because the root transaction was never completed.

The parameters to *ftx_done_urdr()* are:

- `ftxHT ftx` -- the handle to the transaction being committed. If this transaction created subtransactions, those subtransactions must already be committed.
- `ftxAgentId agentId` -- the agent type of the transaction being committed. This is required to associate the undo/redo/rootdone routines with the transaction.
- `Int undoOpSz` – size of the undo record parameter (in bytes).
- `Void* undoOp` – a pointer to the undo record structure, the structure type is determined by the agent type.
- `Int redoOpSz` – size of the redo record parameter (in bytes).
- `Void* redoOp` – a pointer to the redo record structure, the structure type is determined by the agent type.
- `Int rootDnOpSz` – size of the root done record parameter (in bytes).
- `Void* rootDnOp` – a pointer to the root done record structure, the structure type is determined by the agent type.

The steps followed by *ftx_done_urdr()* include:

- Create a log record header (`ftxDoneLRT` structure) that maintains information required to read back undo and redo records during recovery. The record maintains size information for undo, redo, and root done records, as well as some additional recovery information.
- Add the undo, redo, and root-done records for the transaction to the `lrDescT` structure, which represents all the data to be logged for this (sub)transaction. Note that there is no undo record for a root transaction. Committing a root transaction to disk ensures that the changes made in the transaction will always be made, even if the system fails.
- Call *log_donerec_nunpin()* to place the transaction record into the log and unpin all the pages pinned at this transaction level.
- Call *release_ftx_locks()* to release all locks seized at the current transaction level via *ftx_lock()*.
- A loop is executed to apply all the root done records for the transaction. Subtransactions that were previously committed and had registered a root done record now have the root done logic completed. Executing the root done records could require additional information be to be placed into the log, so for each root done record executed, an `lrDescT` record is created and logged with a call to *log_donerec_nunpin()*.
- Finally, the entire transaction is cleaned up. First, the transaction slot is marked as available, then the `ftxT` structure is freed and the `FtxMutex` is released.

As has already been alluded to, a log page is not sent to disk immediately when it is unpinned, and any records of a log page that are not on disk when the system crashes will not be redone. This is generally fine since the write-ahead log rule guarantees that the actual metadata changes are not on disk, and are, therefore, "undone" when the in-memory log page is lost.

The system of delayed log page writes is called **lazy commit mode**. This means that there is no guarantee of transaction persistence at the time the transaction is committed via *ftx_done()*. It buffers the transaction records in-memory (in log buffers) until an explicit `sync(2), fsync(2)` or when there are LOG_FLUSH_THRESHOLD (currently 8) log pages ready to be written. Lazy commit mode ensures file system metadata integrity, but potentially loses transactions when the system crashes. This is acceptable in a UNIX system since UNIX file system semantics utilize lazy writing as a general rule to maximize file system performance.

9.2.4 Transaction Abort: ftx_fail()

ftx_fail() is the primitive used to abort a transaction that can not continue for any reason. Note that *ftx_fail()* must NOT be called once a record has been pinned in a (sub)transaction, or a domain panic will occur. This is because before-images of data are not collected during *rbf_pin_record(),* and therefore the data change can not be undone. This stipulates that all necessary resources required for the (sub)transaction to complete successfully (these being possible situations resulting in a call to fail the transaction due to lack of a particular resource) must be obtained prior to calling *rbf_pin_record()* and making any metadata modifications. To enforce this, pinning a record causes the page to be marked dirty. *Ftx_fail()* will domain panic if it detects a dirty page, since to do otherwise would force inconsistent metadata to disk. In addition, *ftx_fail()* can only be called passing in the handle for the current level's transaction. If *ftx_fail()* is called on a transaction with subtransactions that have completed successfully, these subtransactions will be undone if they provided undo information when they called *ftx_done()*.

The parameters of *ftx_fail()* are:

- *`ftxHT ftxH`* – The transaction to fail.

Ftx_fail() begins by unpinning each page pinned within the transaction being failed. The `perlvlT` structure that maintains per level information is used to determine what pages are pinned. While unpinning each page, *ftx_fail()* verifies that the page is not dirty and domain panics if the page is dirty.

Next, *ftx_fail()* opens the log for reading so that it can undo each of the previously completed subtransactions of the transaction being failed. The log is read starting at the last log record committed (`ftxp->lastLogRec`) and moving backwards one record at a time (only reading records associated with this transaction). For each record, if an undo record exists, the undo record is executed and the resources associated with that record are released. Processing the undo records may cause new records to be written to the log.

All locks for this transaction are released, and, if any new records were written during the fail, a final done record is written to the log. Finally, the transaction slot is made available and the `ftxH` structure is freed.

9.2.5 Registering an agent: ftx_register_agent()

ftx_register_agent() is used to associate a set of functions with a single agent id. This is used to allow a transaction of a specific type to store only an integer id value in the transaction log. The kernel uses the agent id to determine which functions need to be called with an undo, redo or root done record for the transaction. Agent ids are statically assigned, and agents are tracked in an array of `FtxAgentT` structures.

9.2.6 Transaction Locking: ftx_lock_*()

ftx_lock_read() and *ftx_lock_write()* are used to seize a lock for read/write access and to associate the lock with the current transaction level. All locks seized on the same level are chained together (on `perLvlT.lkList`) for quick access when releasing locks during transaction commit or abort.

_ftx_add_lock() is provided to allow a lock that is already seized to be associated with the current (sub)transaction. This allows locks that have been seized prior to a transaction start to be automatically released when the transaction commits.

9.2.7 Pinning Pages: rbf_pinpg()

rbf_pinpg() is the *bs_pinpg()* equivalent for use in transactions. *rbf_pinpg()* is a wrapper around *bs_pinpg(),* but takes actions to associate the page being pinned with the current (sub)transaction so that the page can be unpinned or repinned when the transaction is committed, failed, or recovered.

The parameters to *rbf_pinpg()* are:

- *`rbfPgRefHT pinPgH`* – a return parameter used to pass back a reference handle to the page being pinned.
- *`Void **bfPageAddr`* – a return parameter used to pass back a pointer to the data of the page.
- *`struct bfAccess *bfap`* – the access structure for the file whose page is being pinned.
- *`unsigned long bsPage`* – the page in the file to be pinned.
- *`bfPageRefHintT refHint`* – future hint.
- *`ftxHT ftxH`* – the transaction under which the page is to be pinned.

rbf_pinpg() first scans the global pin table in the `ftx` structure to see if the page has already been pinned within this transaction. If the page is pinned globally (global to the root transaction), the `perlvlT` structure is checked to see if the page was already pinned within the current subtransaction. If so, nothing needs to be done and the function simply returns. Otherwise, the page handle (a reference to the page in the transaction's global pin table) is inserted into the per-level pin table to associate the pinned page with the current subtransaction.

If the page to be pinned is not already globally pinned, a page handle is acquired in the global pin table, and a `perlvlT` pin slot is allocated. After both a global and local pin handle are successfully allocated, *bs_pinpg_ftx()* is called to actually pin the page.

Pages pinned during a (sub)transaction will remain pinned until that (sub)transaction is committed. The page does not remain pinned until the root transaction has completed. To accomplish this, *rbf_pinpg()* associates each page that is pinned within the (sub)transaction with the current transaction level. Then, when the (sub)transaction is committed, all pages that were pinned at that level are unpinned. This is done in the *log_donerec_nunpin()* routine. If a subtransaction needs to be subsequently undone, any pages required are repinned at that time.

9.2.8 Pinning Records: rbf_pin_record()

rbf_pin_record() is the transaction primitive that actually allows the transaction to know what data has changed and to gather after-images of the data being modified. Only data that is explicitly pinned will be redone during recovery. Because before-images are not saved before record modifications are performed, undo records are not automatically created when *ftx_done()* is later called for this transaction.

The parameters to *rbf_pin_record()* are:

- `rbfPgRefHT pinPgH` – the page reference for the pinned page in which the data will be changed.
- `void *recAddr` – address of the record to be pinned. This is not page relative, but the in-memory address of the record to be pinned.
- `int recSz` – the size in bytes of the record to be changed. A record is simply a set of bytes; it does not necessarily correspond to any single data structure.

rbf_pin_record() calculates the page-relative offset and size of the record to be modified and stores this information into the `perlvlT` structure. If a record overlaps with a record already being modified by this (sub)transaction, the two records are merged. *rbf_pin_record()* also changes the `unpinmode` of the page from BS_NOMOD to BS_MOD_LAZY, to indicate that a change to this page has been made. The after-image of the data to be redone is later collected from the pinned page during *ftx_done()* in *addone_recredo()*. If the transaction is committed and there is a page that has not been modified, then that page is simply unpinned.

9.2.9 Transaction Table

Each root transaction is assigned a slot in the domain's transaction table (`domainT.ftxTbld`). The transaction table is an in-memory structure and has a fixed number of slots (currently 30). Thus, the number of concurrent transactions per domain is limited. Only root transactions are assigned to slots; subtransactions are associated with their root transaction's slot.

On a historical note, the designers of AdvFS originally chose a maximum of 30 concurrent transactions system-wide because of the number of pages that could be pinned concurrently in the AdvFS buffer cache. At that time each transaction could have only 8 pinned pages, and the AdvFS buffer cache was fixed at 256 pages. (256 pages / 8 pages/transaction = 32 transactions). They reduced the number of transactions from 32 to 30 to allow for 16 cached pages that were not part of a transaction. Later the buffer cache was allowed to scale with memory size, and the number of concurrent transactions was changed from being system-wide to being domain-wide. The 30 transaction slots per domain has never changed, however.

Transaction slots are allocated in a strict round-robin fashion. This means that if the next slot to be assigned is currently in use, all subsequent root transactions must wait to start until that particular slot becomes available, even if other slots are already available. This was implemented to ensure that there are no long-lived transactions. The round-robin allocation of slots is one reason why root transactions cannot be nested. If the next slot available is the previous root transaction, the thread will deadlock waiting for that transaction slot to be released.

There is a relationship between the number of concurrent transactions system-wide and the size of the buffer cache. The relationship is:

max_transaction_slots_per_system = # cache buffers / max_pinned_pages_per_transaction

This was more of a restriction in v4.0 streams where the size of the AdvFS buffer cache was limited and determined at boot time. In the v5.x streams, we tend to ignore this restriction since the UBC will, theoretically at least, supply the number of buffers we need.

There is also a close relationship between the number of transaction slots per domain and the size of the log. As reported in the original HitchHiker's Guide, the formula is:

slots_per_domain = log_size (bytes) / 4 (quadrants) / max_transaction_size (bytes).

The original designers didn't know what the maximum size of any transaction was, so they tested the system with a 512k log and 30 transaction slots. They didn't run into any problems, so they guessed that making the log 8 times larger (4Mb) should prevent getting 'log half full' errors with 30 transaction slots. If the transaction table size is ever increased, this should be kept in mind. Also, it might be useful to instrument the kernel and try to ascertain the maximum transaction size that is ever achieved.

9.2.10 Metadata Management

Metadata is any file data that must be logged. This includes directories, indices, reserved files, and files with atomic data logging turned on. To guarantee the write-ahead log rule is maintained, a domain-wide **log sequence number** (`lsn`) tracks the last log record to be written to disk. Once a record is on disk, the record stored can be redone or undone if necessary. Therefore, it is safe to allow the metadata changes be flushed to disk.

Each logged page contains an `lsn` value in the `bsBuf` structure that timestamps the changes to that page. When the log has been flushed past the `lsn` value, the associated metadata can safely be written. A system failure before the log gets written to disk would lose the metadata changes, thus automatically leaving the system in the state before the changes took place. This has the effect of making an undo unnecessary, so losing the in-memory portion of the log itself has no effect on data consistency.

The first time a metadata page is modified under transaction control, it is placed at the end of a domain-wide list (`domainT.lsnList`). This **lsn list** is maintained in increasing order by its `bsBuf.origLogRec` field. This is the field that tracks the oldest modification to this page. Therefore, when a page reaches the head of this list, it should be associated with the oldest transaction on that domain. This makes it easy for the kernel to find dirty metadata pages that need to be flushed when the log needs to be checkpointed (see Section 9.3.4).

What prevents a metadata page from being flushed to disk before the log records are written? Whenever a page is modified in AdvFS, it is first pinned, then it is modified, and then it is unpinned. The last unpin of a metadata page will place that page onto a special I/O queue, the wait queue. This queue can be thought of as a holding area for metadata pages that have log data waiting to be flushed. The metadata pages will remain on this queue until one of two things happens. Either the page will be repinned and removed from the wait queue until it is unpinned, or it can be moved from the wait queue to the appropriate lazy queue (see Chapter 8) after the log page is written to disk. The writing of the log page is detected during the I/O completion in *bs_io_complete(),* which sends a message of type CK_WAITQ to the background I/O thread. This thread runs *check_cont_bits()* that moves the metadata buffers that are now eligible to be flushed to disk to the appropriate lazy queue (smoothsync or readyLazy).

9.3 Log File Management

9.3.1 Log Size

By default the log is 4 Mb in size. The reason for this size is given in Section 9.2.9 where the relationship between the size of the log and the size of the transaction table is discussed. It is possible to increase the

size of the log, however, when a new domain is created. The `mkfdmn` command has a `-l` command that allows the creator to specify the log size in pages. Alternatively, the `switchlog` command has a `-l` option that can be used to resize the log while moving the log to a different volume. The `man` page for `switchlog` does not make reference to the `-l` option, but it does work.

9.3.2 Log Writing

Transaction records are most typically written to the log in the context of a transaction commit or abort. The routine *log_donerec_nunpin()* is called to write the transaction record and unpin the pinned pages. To write the transaction record, the routine *lgr_writev_ftx()* is called. This routine reserves the correct amount of space in the log, pins the appropriate log page(s), assigns a new `lsn` to the log record, copies the record to the log page, and unpins the page(s). It will also add any metadata pages to the lsn list if required. The write may trigger a flush of the in-memory log pages to disk if a synchronous write to the log was requested.

9.3.3 Log Flushing

Flushing the log is handled primarily by the *lgr_flush_start()* and *lgr_flush_sync()* routines. *lgr_flush_start()* determines if we need to unpin the last log page to flush up to the lsn required. If so, it unpins that page and places it onto the blocking I/O queue. Pages up to this last page are flushed by calling *bs_logflush_start()* which essentially walks through the buffers on the log's dirty buffer list, moving them to the blocking I/O queue if they are not already there. *lgr_flush_sync()* will wait for the log pages up to and including the specified `lsn` to be flushed to disk. Waking of this thread is handled in *bs_wakeup_flush_threads()* which is called by *bs_io_complete()* whenever a write to a file has been completed. If the appropriate `lsn` has been flushed to disk, then the thread is awakened.

Flushing may be initiated by the log write routines if a synchronous write was requested. Flushing will also be initiated as old pages in the log are needed for new transaction records. This situation is discussed in the next section.

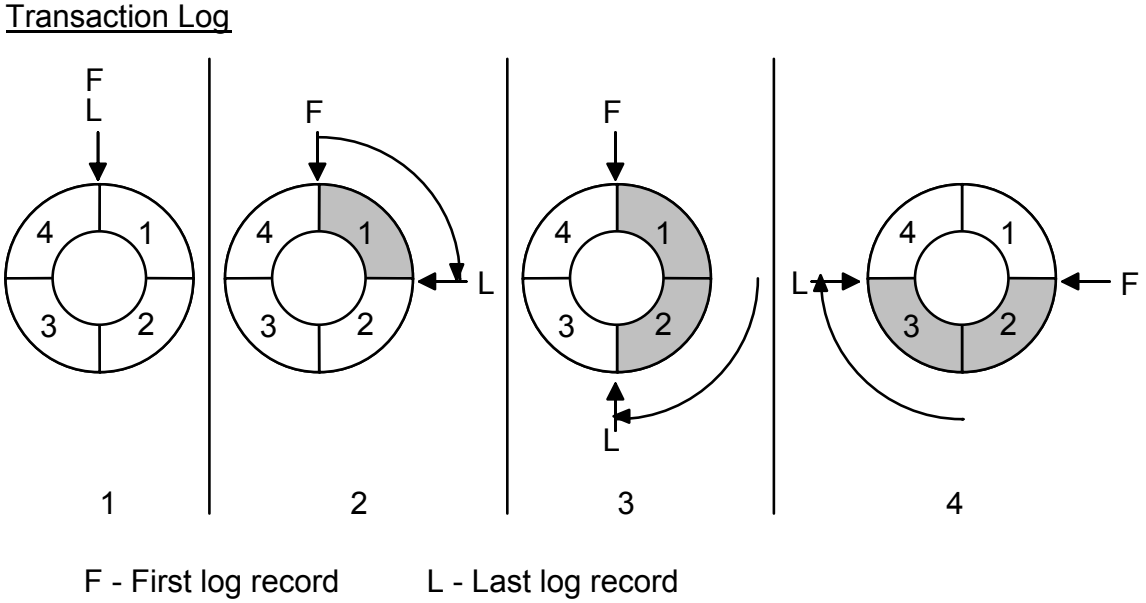
9.3.4 Log Checkpointing

The log can be viewed as a large circular buffer, with strict rules regarding when old log pages can be overwritten. This is sometimes called 'wrapping' the log. The main rule is that there may be no dirty metadata buffers that are associated with the transactions whose records are about to be overwritten (destroyed) in the log. In a way, this is an addendum to the write-ahead log rule. Just as it is important to flush log pages associated with metadata changes, so too, it is important to ensure that the metadata changes are flushed to disk before the associated log pages are overwritten. Consider the metadata pages and log pages moving through the following steps:

Operation	State
Read metadata page into the buffer cache	
Modify the metadata within a transaction	Log pages must precede metadata to disk
Write transaction records to log buffers	

Flush the log buffers to disk	Metadata may now be flushed to disk, but there is no requirement to do this quickly
(Possibly, do more modifications on this metadata)	
Notice impending need to wrap the log	Must be sure that associated metadata buffers have been flushed to disk
Flush metadata buffers associated with log region that needs to be wrapped.	After this is done, neither the log transaction records nor the metadata changes need to be saved any longer; the modifications have been permanently made. It is safe to overwrite the old log records.
Allow the logger to overwrite the old log records up to the next quadrant.	

The process of flushing metadata buffers to ensure that the log can safely wrap is called **log checkpointing**. AdvFS does checkpointing on log quadrant boundaries. The log is divided into four quadrants (see next figure). Whenever the log crosses a quadrant boundary, AdvFS flushes all dirty metadata buffers associated with the previous log quadrant. This is done by allowing all outstanding transactions to complete but prohibiting new transactions from starting until the buffers are flushed and the log is trimmed up to the end of the previous quadrant. AdvFS also ensures that the log is not more than half full when a quadrant boundary is crossed; this ensures that there is enough space in the log for recovery if the system fails.



The flushing of the metadata buffers is done in the routines *bs_pinblock()* and *bs_pinblock_sync(),* which are passed an `lsn` value. These routines will flush any metadata buffers on a domain's `lsn` list whose `origLogRec` values are older than that value. When those buffers have been flushed, it is safe to overwrite the next quadrant of the log.

9.3.5 Log Isolation

The log file of an AdvFS domain is initially placed onto the first virtual disk added to the domain. If this disk also contains many actively-accessed files, the interlacing of I/O to the log and the active files may overload the bandwidth of the disk or its hardware controller. One way to alleviate this is to place the log onto a volume that is not so active. This can be done manually by using one of the 2 following methods:

Method 1. This will simply move the existing log to a new disk and isolate it from other files being used by applications.

- Pick a disk or disk partition for the log. You may need to use the `addvol` utility to add a new disk to the domain.
- Use the `switchlog` utility to move the log to that disk.
- Use `showfdmn` to figure out how many free blocks are left on that disk.
- Make a file that is that size.
- Use the `migrate` command to move the file to that disk or partition.

Method 2. This method will build a 32 Mb log (currently believed to be optimal in clustered systems) on a disk partition with no free space remaining so no other files can be placed there. Feel free to adjust this procedure to make a log that is something other than 32 Mb in size.

- Unmount the domain on which you are isolating the log.
- Use `/sbin/disklabel -e <disk>` to edit the disk label of a new disk. You will only be using the 'a' partition to store the log. Edit this disklabel so that the 'a' partition is 65664 blocks long.
- Use `/usr/sbin/addvol <disk> <domain>` to add the 'a' partition of this disk to the domain where you want to isolate the log.
- Use `showfdmn <domain>` to see what the volume index is for the newly-added disk. This will be needed in the next step.
- Run `/sbin/advfs/switchlog -l 4096 <domain> <disk index>` to move the log to the specified disk and make it 4096 pages long (32 Mb). Note that the `man` page for `switchlog` does not make reference to the `-l` option, but it does work.
- Remount the domain.

There was a project to allow the kernel to isolate the log without manual intervention. The goal was to move the log to its own virtual disk within the domain, and this disk would not be used for any other AdvFS files. Although this code was never submitted to the development pools, there are functional and design specifications for this work that can be found on the Tru64 Repository AdvFS page.

9.4 Domain Recovery

In the event of a system failure, AdvFS will recover a domain when its first fileset is mounted (this activates the domain). RBMT information is used to open the reserved files (including the log). The log manager finds the end of the log which allows the transaction manager to also find the beginning of the log (or more accurately, the crash redo log address). The transaction manager keeps its log address in each log record.

Recovery is done in three passes:

Pass 1: The RBMT is recovered. This is primarily making the extent maps for the reserved metadata files consistent for the next pass. This is referred to as the Meta-Meta Pass.

Pass 2: The reserved metadata files are recovered (this includes the BMT, Storage Bitmap and Root Tag Directory). Mainly the extent maps are made consistent for the next pass. This is referred to as the Meta Pass

Pass 3: All other logged files are recovered. (This includes directories and atomic-write data-logging files). This is referred to as the Data Pass

During each pass the following phases are performed:

Redo phase: Applies all after-image records logged (these were captured via *rbf_pin_record()*).

Roll-back / roll-forward phase: The log is used to create in-memory transaction information. All committed transactions are completed (rolled forward) and all uncommitted transactions are undone (rolled back).

Consider a scenario where a transaction is in progress and 3 (of 5) subtransactions are committed when the logger pushes a portion of the log to disk (probably because the LOG_FLUSH_THRESHOLD-th page was unpinned and the previous group of pages is being flushed). The transaction continues and commits the last two subtransactions. The amount of the log information that gets flushed to disk before the system crashes will determine whether this transaction will be redone or undone during domain recovery.

When a transaction log record is written to disk and the system crashes before the associated metadata changes have been written to disk, the changes must be redone. A transaction can only be redone if all the subtransactions associated with the transaction have been written to disk at the time of the system crash. If the last two subtransaction log records are written to disk before the system crashes, the entire transaction will be redone, starting with the first subtransaction.

Alternatively, when only a portion of the transaction's log records are written to disk prior to a crash, the entire transaction must be undone. In the above example, if subtransactions 4 and 5 were not written to disk prior to the crash, the entire transaction would be undone. There would be no on disk indication that transactions 4 and 5 ever existed, and no record indicating the entire transaction had completed. Therefore, the undo would progress from the last complete log record (subtransaction 3) and proceed backwards undoing each subtransaction.

Note: When undoing subtransactions, although changes are undone from the last subtransaction backwards, locks are reacquired from the start of the transaction forward and released as the undos progress.

9.5 Structure overview

9.5.1 In Memory

The in-memory structures for a single in-progress transaction are illustrated below.

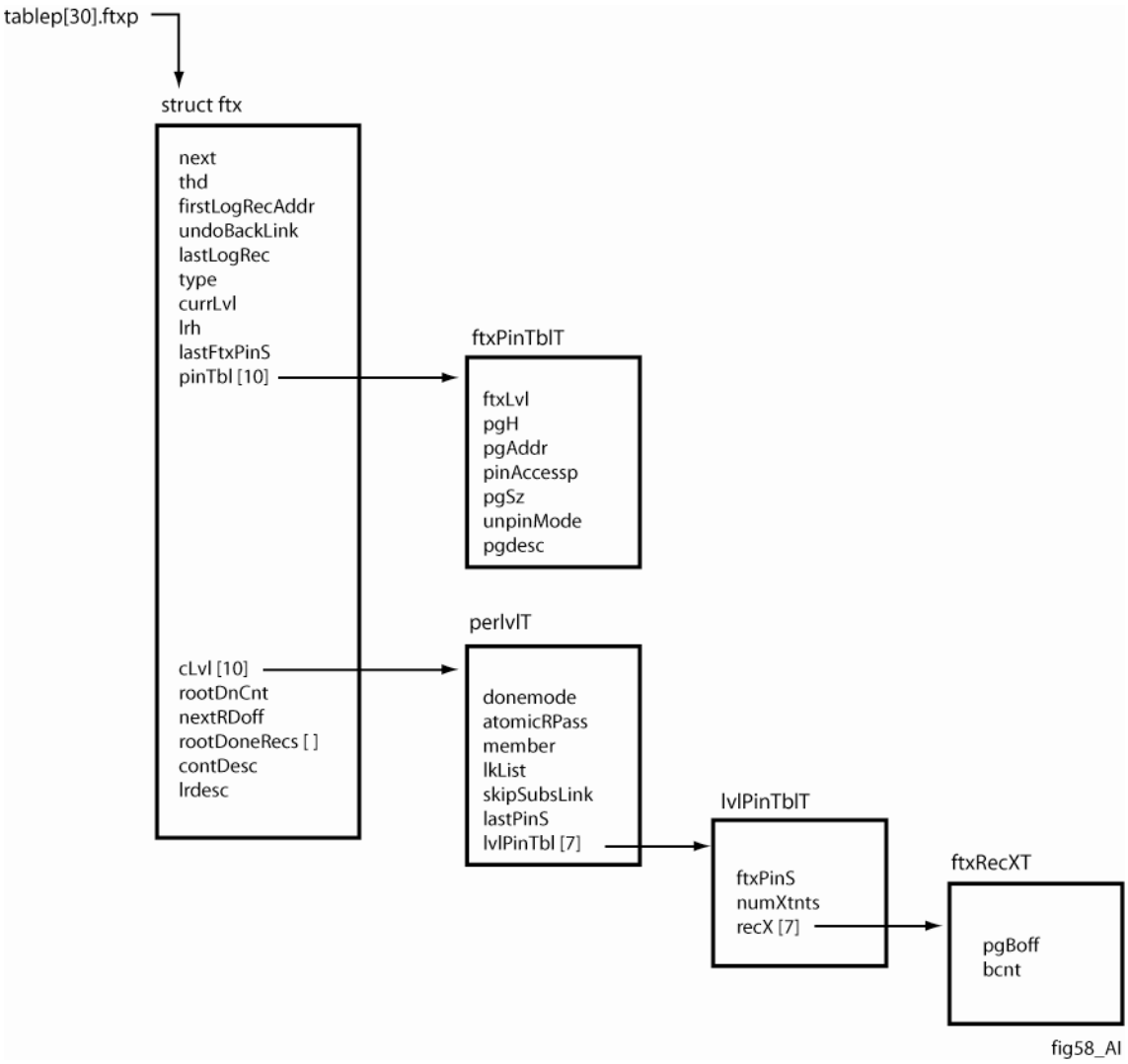


Figure 35: In-memory structures for a single transaction

Transaction Table Descriptor (ftxTblD)

There is one transaction table per `domainT` which is used to hold an array of transaction descriptors (`ftx` structures) for active transactions. As previously mentioned, a root transaction must have a slot in the transaction table in order to start. The `ftxTblD` structure provides an array of `ftxSlotT` structures which maintain a pointer to an `ftx` structure and a state (available, busy, in use exclusively). This array composes the basic transaction slot system. The transaction table also provides state information used to trim the log (see Section 9.3.4 on Log Checkpointing) and to track the slot use.

Transaction Descriptor (ftxStateT & ftx)

`ftxStateT` is a type definition for an `ftx` structure. It is used frequently, but the name can become confusing since this represents a descriptor for an entire root transaction.

The `ftx` structure tracks information for a transaction and all its subtransactions. The `ftx` maintains information including all currently pinned pages (at all active transaction levels), and the current

transaction level (the root transaction is level 0). Additionally, the `ftx` structure maintains pointers to the first and last log records for the entire transaction tree.

Transaction Handle (ftxH)

An `ftxH` structure represents a handle to a transaction. The structure provides the domain on which the transaction exists, an index into the transaction table, and a transaction level.

Global Pin Table (ftxPinTblT)

The global pin table is a structure used to monitor the pages currently pinned by the current transaction. So the term 'global' here is used to mean 'global to the transaction', and refers to pages pinned at all levels in the current transaction. This table is of fixed size, such that there can be at most FTX_TOTAL_PINS (currently 10) pages pinned per transaction.

Per-level State (perlvlT)

This structure represents a single node of a transaction or subtransaction. It is used to track which pages are pinned at this level, what records have been pinned in which pages, and how large the pinned regions are. Even a single-level transaction will have both an `ftx` structure and a `perlvlT` structure, describing exactly what the transaction has done on that level and at that node of the transaction tree.

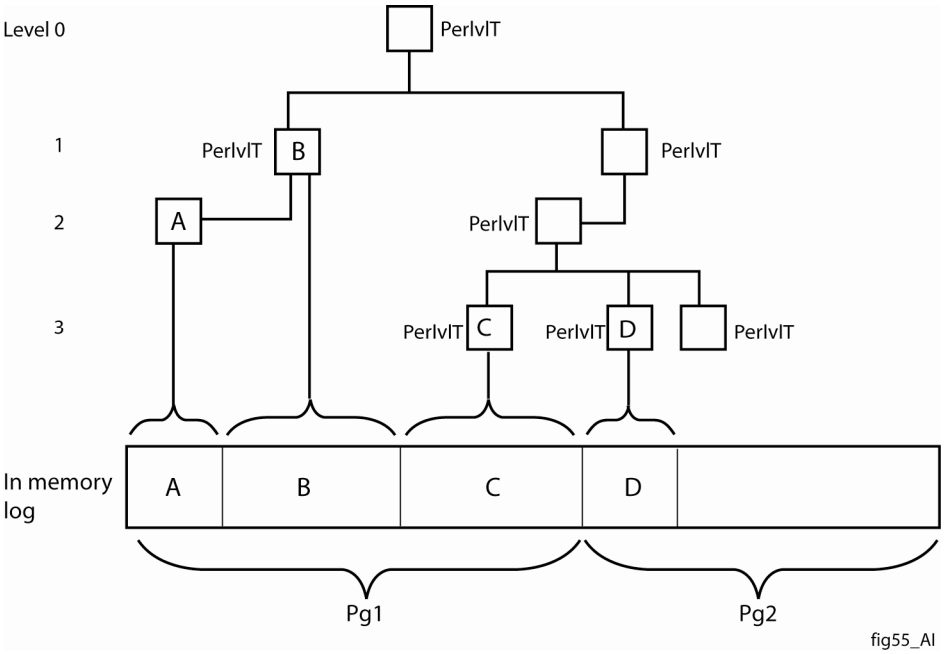


Figure 36: Per-level transaction tree

Per-level Pin Table (lvlPinTblT)

The per-level pin table keeps track of pages pinned within the current level (subtransaction) of the transaction. This table is of fixed size, such that there can be at most FTX_MX_PINP (currently 7) pages pinned per subtransaction level. Each entry in this table maintains the information about a page pinned at this level, including a vector of records that have been pinned on this page.

Pinned-record State (ftxRecXT)

This structure represents the location (page offset and size) of a single pinned record on a page. There can be at most FTX_MX_PINR (currently 7) pinned records per page.

Log Record Descriptor (lrDescT)

The log record descriptor structure is used to compose one large log record from several different smaller records (header, undo record, redo record, root done record). When composing a log record for a transaction, this structure is used to associate several disjoint buffers into one buffer. This structure maintains an array of `logBufVectorT` structures that each point to a portion of the entire log record for the subtransaction. Note that there is no standard redo, undo, or root done structure. Each of these records is unique to each type of transaction.

The following figure shows the in-memory representation of a subtransaction. The structures in the figure map directly to the on-disk records created for each subtransaction. These on-disk structures are discussed in the next section.

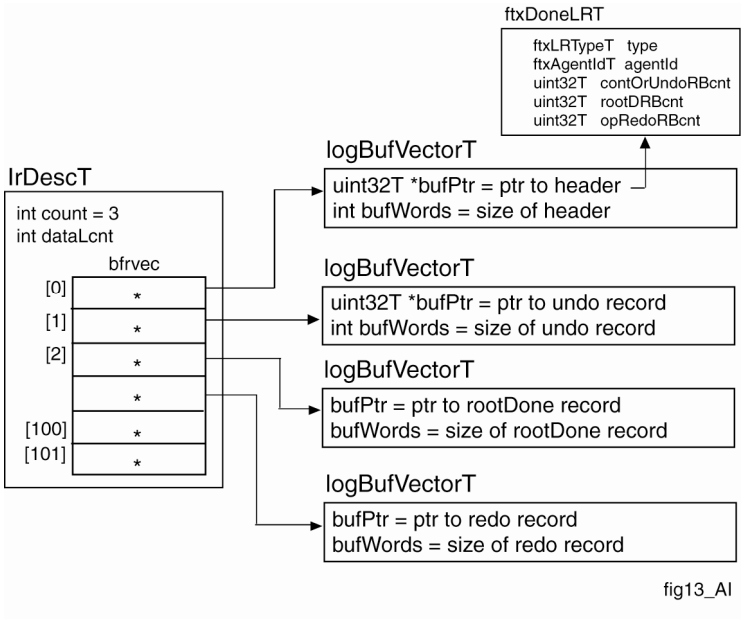


Figure 37: In-memory representation of a subtransaction

9.5.2 On Disk

9.5.2.1 Logical View

The log has several logical views. First, the log is a circular buffer of pages. Each log page contains a header (`logPgHdrT`) and a trailer (`logPgTrlrT`), with a stream of transaction records in between (See the second figure in Section 2.3.4). There is an `lsn` associated with each log page header. Typically the `lsns` will increase as we view the log pages sequentially. At some point, however, the `lsns` will drop dramatically between two sequential pages; this is the tail/head of the log.

The second view of the log is a series of transaction records between the page header and trailer. Each transaction record is comprised of 3 parts: a `logRecHdr` record, an `ftxDoneLRT` record, and zero or

more `undoRec`, `redoRec`, or `rootDoneRec` records. Each of these can be thought of as a logical record. These records are chained forward and backward via the `logRecHdrT.nextRec` and `logRecHdrT.prevRec` fields. Walking through the records via this chain will traverse all records in the log. The records are also chained backward via the `logRecHdrT.prevClientRec` field, and links subtransaction records within a given transaction. This allows the logger to start with the last record in a given transaction, and to quickly find all subtransaction records associated with that transaction.

The next figure shows a hypothetical example of a log page with 3 transaction records. Note that the ftxDoneLRT and the undo, redo, and root done records map directly to the in-memory structures shown in the previous figure.

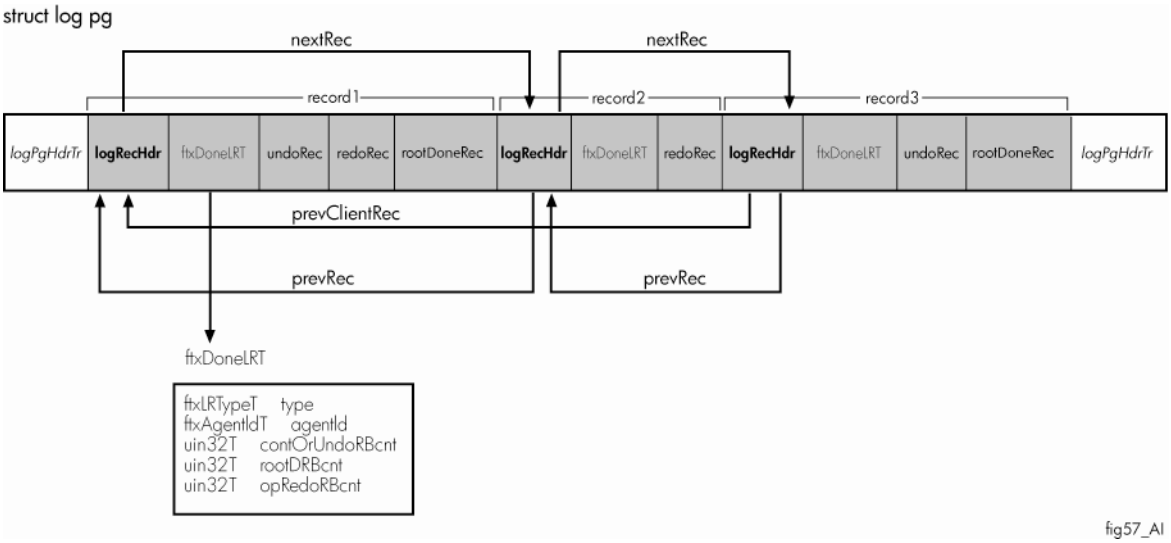


Figure 38: On-disk log page

9.5.2.2 Physical View

Each log page contains a header and a trailer, with a stream of transaction records in between (`struct logPgT`). However, because the logical unit of storage on the disk is the 512-byte sector, the logger can also view the page as a series of 16 sectors, and adds some information to the beginning of each sector within the log page as a way of verifying that all the data within the page is consistent. The first four bytes of each sector are overwritten with the `lsn` for the page (See Figure 7 in Section 2.3.4) It gets this value from the page header (`logPgHdrT`). Because overwriting this data will result in the destruction of the underlying data in the transaction records, the logger first saves the data that will be overwritten to a fixed-length array in the page trailer (`logPgTrlrT`) which has been reserved for exactly this purpose. So the sequence of events when unpinning a log page would be: 1) copy the leading four bytes from each logical sector within the page to the `lsnOverwriteVal` vector in the page trailer; 2) copy the `lsn` value from the page header to the leading bytes in each sector; 3) release the page. This is all done in the routine `lgr_make_pg_safe()`. When the page is read back in from the disk, the procedure is reversed: the page is read, the `lsn` values at the beginning of each sector are verified to be all the same, and then the data from the page trailer is copied over the leading bytes in each sector throughout the page. The in-memory page is then ready for use by other routines. This is done by the routine `lgr_restore_pg()`.

9.6 Infinite Log Sequence Numbers (LSNs)

9.6.1 Assumptions about LSNs

- AdvFS uses an unsigned integer for the LSN and lets its value wrap (overflow).
- An LSN value of 0 is not valid; it is used to mark a log page which has never been written.
- Numerically larger LSNs are "older LSNs". By older we mean that they were generated and written to the log at a later time. So, if LSN(a) is less than LSN(b), then LSN(b) is the older LSN.
- An LSN **wrap** occurs when the sign bit of an LSN changes. Since we generally consider LSNs to be unsigned we are mostly interested in the LSN wrap that occurs when the sign bit changes from 1 to 0. The secondary LSN wrap occurs when the sign bit changes from 0 to 1.
- There is a maximum number of LSNs that can be in the log at any time; we call this MAX_LOG_LSNS.
- There is a maximum LSN that can be represented, called MAX_LSN. For example, a 16-bit unsigned LSN has a maximum value of 65536.

9.6.2 Conditions that have to be dealt with when the LSN wraps:

When the LSN wraps (MAX_LSN + 1) we must skip over LSN 0 because it is invalid. Also, once the LSN has wrapped we have to be able to determine which LSNs are older when comparing the LSNs before the LSN wrapped with the LSNs after the LSN wrapped. This can be done by casting all LSNs to *signed* integers when comparing them. This way all pre-wrap LSNs are monotonically *decreasing negative* numbers (due to 2's complement binary arithmetic) and all post-wrap LSNs are monotonically *increasing positive* numbers.

This switch to using a *signed* comparison works fine until we have a transition from positive LSNs to negative LSNs (a secondary LSN wrap). When we make this secondary transition, we need to switch back to using an *unsigned* comparison of LSNs.

To summarize, we use the **current LSN's** sign to determine if a signed or unsigned comparison is to be used when comparing any two LSNs in the log (this does not work for arbitrary LSNs, they must be in the log):

If the current LSN is unsigned, use a signed comparison
If the current LSN is signed, use an unsigned comparison

9.6.3 Conditions that have to be dealt with when locating the log's end page:

The above method of comparing LSNs using the current LSN's sign works fine for a running system. However, when the system is booting there is no concept of a current LSN until after the log's END_PG has been located. To locate the log's END_PG we must be able to compare two LSNs in the log. The same basic LSN comparison principle applies: we must be able to determine whether we should use a signed or unsigned comparison. Since we have no current LSN to aid us in making this decision we must use a different method.

To determine this method, consider that there is some maximum number of LSNs that can be in the log at any time (MAX_LOG_LSNS). In addition, there is a maximum LSN that can be represented. For example, a 16-bit unsigned LSN has a maximum value of 65536 (MAX_LSN).

Rule 1a: MAX_LSN must be greater than MAX_LOG_LSNS.

This guarantees that we never have an LSN in the log more than once. Therefore, there can be only one LSN wrap in the log at any time. However, given that there is also a **secondary LSN wrap** where we go from unsigned LSNs to signed LSNs then we need to make Rule 1 more strict if we want to guarantee that we never see more than one LSN wrap (signed to unsigned, or unsigned to signed) as follows:

Rule 1b: (MAX_LSN / 2) must be greater than MAX_LOG_LSNS.

Given that only one LSN wrap can exist in the log at a time, we can formulate a rule that allows us to detect whether or not the log contains an LSN wrap by simply looking at random LSNs in the log (we choose to use the first LSN in the first log page).

Rule 2: An LSN wrap may exist in the log if the following condition is true:

-MAX_LOG_LSNS < (any LSN in the log) < MAX_LOG_LSNS

Call this range between –MAX_LOG_LSNS and MAX_LOG_LSNS the LSN_WRAP_RANGE.

Rule 3: An LSN wrap does not exist if the log contains any LSNs outside the LSN_WRAP_RANGE defined by Rule 2.

Rule 4: If an LSN wrap may be in the log, then use a signed comparison for the LSNs in the log. Otherwise, use an unsigned comparison.

Consider an example using a 4-bit LSN and a log that can hold 4 LSNs:

4-bit LSNs can be viewed as follows:

Unsigned LSNs:	8 9 a b c d e f 1 2 3 4 5 6 7 ...
Signed LSNs:	-8 -7 -6 -5 -4 -3 -2 -1 1 2 3 4 5 6 7 ...

A 4-LSN log can be viewed as a forward-scrolling window through these LSNs. For these examples we assume that the first LSN in the log is the "Log Beginning" and the last LSN is the "Log End". In other words, we don't show cases where the "Log End" is somewhere in the middle of the log as this does not affect the examples.

The two extremes where the log contains an LSN wrap are as follows:

	Case 1	Case 2
Unsigned:	d e f 1	f 1 2 3
Signed:	-3 -2 -1 1	-1 1 2 3

All other cases where the log contains the LSN wrap are between these two end cases.

Using Rule 2 we know that if the log contains any LSN in the range -3 to 3 then we know that the log may contain a log wrap (MAX_LOG_LSNS is 4, so the LSN_WRAP_RANGE is -3 to 3). This is confirmed by the above two cases. Using Rule 4 we must use a signed comparison for the LSNs in the log. The two cases show that by using a signed comparison we get the correct increasing LSNs (-3, -2, -1, 1) and (-1, 1, 2, 3). An unsigned comparison would have given the

incorrect LSNs of (d, e, f, 1) and (f, 1, 2, 3) which would cause us to miss the correct end of the log.

Rule 3 tells us that if an LSN outside the range -3 to 3 exists in the log then we know that there is no LSN wrap in the log which by Rule 4 we can use an unsigned comparison. In the following cases we've moved the log "window" so that -4 is in the log for Case 1 and 4 is in the log for Case 2. Notice that the log wrap is now no longer in the log.

	Case 1	Case 2
Unsigned:	c d e f	1 2 3 4
Signed:	-4 -3 -2 -1	1 2 3 4

Using the above two cases we also see that if we picked a random LSN from these two logs we could get an LSN that is in the -3 to 3 range causing us to use a signed comparison or we could get an LSN that is outside this range which causes is to use an unsigned comparison. Note that either type of comparison works in this case.

The question now is: "If the log contains LSNs both inside and outside the LSN_WRAP_RANGE, can we always use either signed or unsigned comparisons?". The answer is YES if we guarantee that the log cannot contain the secondary wrap and an LSN in the LSN_WRAP_RANGE. The following rule guarantees this and is a further restriction of Rules 1a and 1b:

Rule 5: MAX_LOG_LSNS must be less than or equal to MAX_LSN / 4.

In practice this is not an issue because we typically use a 32-bit LSN and the log is relatively small compared to the MAX_LSN of a 32-bit LSN.

Back to our example: For our 4-bit LSN we have a maximum log size of 4. The following cases show that even if -3 or 3 are in the log we cannot also have the secondary LSN wrap in the log (the secondary wrap is LSN 7 and LSN 8 for a 4-bit LSN):

	Case 1	Case 2
Unsigned:	a b c d	3 4 5 6
Signed:	-6 -5 -4 -3	3 4 5 6

9.6.4 Additional rules if the LSNs are 'jumped' during recovery:

Normally MAX_LOG_LSNS can be calculated as follows:

MAX_LOG_LSNS = MAX_LOG_PGS * MAX_RECS_PER_LOG_PG

However, AdvFS allows a certain number of log pages to be written to the log simultaneously without worrying about the order in which the writes complete. The number of pages that can be outstanding is called LOG_FLUSH_THRESHOLD.

Since we don't know which of these pages were actually written to the log during a system crash, log recovery ensures that the new "post recovery current LSN" is unique by adding

(LOG_FLUSH_THRESHOLD * MAX_RECS_PER_LOG_PG + 1) to the "end LSN" in the log (this is the last LSN in the log end page).

We will call (LOG_FLUSH_THRESHOLD * MAX_RECS_PER_LOG_PG) the CRASH_ADDER from now on for simplicity.

Theoretically, the system could crash right after it filled just one page. This would cause us to add the CRASH_ADDER after each page that was written. This could be done (MAX_LOG_PGS - 1) times. Therefore, the calculation for MAX_LOG_LSNS becomes:

MAX_LOG_LSNS = MAX_LOG_PGS * MAX_RECS_PER_LOG_PG +
(MAX_LOG_PGS - 1) * CRASH_ADDER.

or (simplified):

MAX_LOG_LSNS = MAX_RECS_PER_LOG_PG *
(MAX_LOG_PGS + (MAX_LOG_PGS - 1) * LOG_FLUSH_THRESHOLD)

Chapter 10: Quotas

In an environment where storage can be used by more than one user, there is the potential for one user to use more than their fair share of the storage. It is easy to forget to delete temporary or unused files. It can also become very time consuming for system administrators to maintain and track storage allocation when they are supporting a large group of users. To mitigate this problem, both UFS and AdvFS file systems provide system administrators with a mechanism called **quotas** that allow them to gain more control over how storage is allocated. Quotas are a way to limit the number of files created or the amount of space consumed by a specific user, group, or within an entire fileset. Quotas are established and maintained via quota system commands, and there are also tools that offer reports of storage consumption. UFS and AdvFS both allow the administrator to limit storage based on `user-id` and `group-id`, but AdvFS also allows quotas on filesets. This is not needed on UFS because its file systems do not share storage the way AdvFS filesets can share storage in a common domain. Fileset quotas are used to control the growth of a fileset, allowing them to compete fairly for storage within the domain. In this chapter we will look at how AdvFS manages user, group, and fileset quotas.

AdvFS can limit storage allocation both by disk blocks and number of files. Thus, each user could be restricted to using 1G of storage (by disk blocks) and 10,000 files. The file system can impose either soft or hard limits. When a **hard limit** is reached, AdvFS will disallow any new disk space allocations or file creations that would exceed the allowed limit. A **soft limit** may be exceeded for a period of time known as a **grace period**. If the soft limit is still exceeded after the grace period has expired, then no more allocations or file creations are allowed until the space is freed or files deleted that would allow the quotas to be below the soft limit. Enforcement of quotas can be enabled or disabled by user, group, or fileset. Quota enforcement can be enabled for a subset of filesets in a domain.

AdvFS maintains quota statistics for users and groups in two files: `<mount_point>/user.quota` and `<mount_point>/group.quota`. These are special files that can not be modified by user programs. For all filesets, the fileset quota information is kept in entry 0 of the `group.quota` file for that fileset. Quota statistics are maintained by AdvFS even if the quotas are not being enforced; more on enforcement later.

When a fileset is mounted, the group quota file is opened, the extent map is brought into memory, the pages are walked, and the block and inode usage for all parties is tallied.

The following information is stored in the two quota files:

- **Inode soft limit** - the number of files allowed to be allocated to a particular user or group (**party**), which, when exceeded, will cause a warning to be issued at the console
- **Inode hard limit** - the maximum number of files allowed to be allocated to a party
- **Block soft limit** - the number of blocks allowed to be allocated to a party, which, when exceeded, will cause a warning to be issued at the console
- **Block hard limit** - the maximum number of blocks allowed to be allocated to a party
- **Grace Period** - the amount of time for which a party is allowed to exceed a soft limit
- **Usage Inodes** - the number of files actually allocated by a party
- **Usage Blocks** - the number of blocks actually allocated by a party

10.1 Quota Utilities

The following table is a summary of the AdvFS quota Utilities. Please note that unless otherwise indicated, you must have root permissions to use these commands.

Quota Utility	Function
quot	Displays disk space used and number of files created for each user. (UFS or AdvFS)
quota	Displays disk space usage and quota limits by user and group (UFS or AdvFS)
quotacheck	Checks fileset quota consistency (UFS or AdvFS)
repquota	Displays a summary of user, group, or fileset quotas (UFS or AdvFS)
edquota	Allows editing of user and group quotas, including hard limits, soft limits, and grace period (UFS or AdvFS)
quotaon	Turns quota enforcement on (UFS or AdvFS)
quotaoff	Turns quota enforcement off (UFS or AdvFS)
chfsets	Displays or changes fileset quotas for AdvFS filesets (AdvFS only)

10.1.1 quot

The `quot` command displays disk space usage and the number of files created by each user on the system. This information is not quota-specific, but allows tracking of fileset usage by user.

10.1.2 quota

The `quota` command displays, by default, disk space usage and limits for the current user (you do not need to have root permissions to use this command). Its various options allow reporting group quotas as well as quotas on all mounted filesets, just filesets listed in `/etc/fstab`, or just filesets listed in `/etc/fstab` which are over their quota limits.

The program gathers a list of mounted file systems (according to the options selected) and uses a `quotactl`(Q_GETQUOTA64) to read the appropriate quota records. If this call is not successful, and we have root permissions, then `quotactl`(Q_QUOTAINFO) is called which determines the size of the quota records (`dqBlk32` or `dqBlk64`). Knowing that, we navigate the quota file via `lseek`() and `read`(), retrieving the records into a temporary buffer and printing the requested information.

10.1.3 quotacheck

This command checks current file system usage against usage recorded in the quota file. If any inconsistencies are detected, the quota file and its in-memory structures are brought to a consistent state. Both user and group quotas are verified. Although this command will check only mounted file systems,

the user must make sure that the file systems being verified are quiescent (no files are open). Any file system activity could yield unreliable results. See the `man` page for options for this command.

On an AdvFS fileset, this utility calls `advfs_tag_stat()` iteratively, retrieving and adding each file's size to the appropriate user and group counters. `advfs_tag_stat()` dispatches into the `tagdir_lookup_next()` routine which will traverse the tag file for the fileset, returning information about the next file in the tag file. It eventually returns ENO_SUCH_TAG when it has reached the end of the tags in the fileset. If the scan-by-tag terminates normally, the `update_advfs()` routine is called to update the user or group quota information as appropriate. This routine opens and reads the quota file, comparing the data from the file with the information accumulated in the previous loop. If there are any discrepancies, the `quotactl`(Q_SETUSE64) is used to update the quota file.

10.1.4 edquota

This command is used to 1) add and modify user and group quota limits, and 2) modify file system quota grace periods. The `quota` command is needed to display these values originally. Only those file systems that are currently mounted and are listed with quota entries in the `/etc/fstab` file may be modified with this utility. Only a user with root privileges can use this utility. See the `man` page for more information on using this command.

10.1.5 repquota

This utility is used to display user, group, and fileset quota information. The user must have root permissions to use this command.

`Repquota` uses `quotactl`(Q_QUOTAINFO) to determine the size of the quota records (`dqBlk32` or `dqBlk64`). Knowing that, we navigate the quota file via `lseek`(), retrieve the records using `quotactl`(Q_GETQUOTA or Q_GETQUOTA64), save them into a temporary buffer, and print the requested information.

10.1.6 quotaon and quotaoff

These commands enable and disable user and qroup quotas. The actual quotas must have been previously established using the `edquota` command. The user must have root privileges to use this command, and the file systems must have quotas specified in the `/etc/fstab` file and be mounted. An example entry in `/etc/fstab` to enable user and group quotas would be

```
test#test        /test1  advfs userquota,groupquota,rw 0 2
```

Remember that AdvFS always maintains user and group file information, and these utilities merely turn quota enforcement on and off. Each time a fileset is unmounted and remounted, the `quotaon` utility must be run to enforce user and group quotas.

Fileset quotas are not maintained with these commands – the `chfsets` utility must be used.

10.1.8 chfsets and showfsets

This command is used for changing attributes of AdvFS filesets. These attributes are written to disk, so their settings are preserved across reboots and fileset unmount/remount activities.

This AdvFS program allows for changing certain characteristics of an AdvFS fileset. This includes setting a fileset quota for soft and hard limits on number of files, and soft and hard limits for number of blocks. Note that the fileset grace period is only set by the `edquota` program.

Fileset quotas can only be enabled using the `chfsets` command. There is no need to turn on quotas via `quotaon`, or to change `/etc/fstab`. Fileset quotas are displayed as part of the output of `showfsets` and `vdf`.

When looking at the showfsets output,

```
% /sbin/showfsets -k test
fset1
        Id           : 3e835a55.0006ae8c.1.8001
        Files        :       21,  SLim=        0,  HLim=        0
        Blocks  (1k) :    75062,  SLim=        0,  HLim=        0
        Quota Status : user=off group=off
        Object Safety: off
        Fragging     : on
        DMAPI        : off
```

The `Quota Status` line provides information on the state of the user and group quotas. However, this has nothing to do with fileset quotas. The the `Slim` and `Hlim` values shown on the `Files` and `Blocks` lines provide the current Soft and Hard limits, respectively, for files and file storage for the fileset. In the example above, fileset limits are not enforced (shown by the limits equal to zero). To turn fileset quotas on for fileset `fset1` and set the file storage hard limit, use `chfsets`:

```
# /sbin/chfsets -b 100000 test fset1

# showfsets -k test fset1
        Id           : 3e835a55.0006ae8c.1.8001
        Files        :       21,  SLim=        0,  HLim=        0
        Blocks  (1k) :    75062,  SLim=        0,  HLim=   100000
        Quota Status : user=off group=off
        Object Safety: off
        Fragging     : on
        DMAPI        : off
```

If a user now tries to allocate storage in the fileset that exceeds the new quota, an error message will be displayed and the allocation will not be allowed:

```
% cp /vmunix /fset1/user1/foo

/fset1: write failed, fileset disk limit reached
cp: /fset1/user1/foo: Disc quota exceeded
```

Note that root is not similarly limited by fileset quotas.

10.2 In-memory Quota Structures

The `dQuot` structure and many of the other quota structures are defined in fs_quota.h. The `dQuot` tracks disk usage for a user or group on a file system. There is one `dQuot` allocated for each current user or group in a file system that is under quota control. The `dQuot` structure contains fields for its hash table link and key, quota flags, a quota type, the number of active references to the structure, the uid or

gid for that entry, a dQ structure containing the usage and quotas, a pointer to the quota's associated filesetNode, and a dqLock to ensure mutual exclusion when modifying the dQuot structure. The dqLock is protected by the dquotMutex. The dynamic dQuot hash table (DqHashTbl) contains the active dQuots. The hash bucket lock protects the individual dQuot reference field (dq_cnt) as well as the dQuot's hash chain pointers.

The fileSetNode structure contains many fields that are used to maintain fileset quotas. The quota-relevant fields pertain to soft and hard block and file limits, the number of quota blocks and files used in the fileset, fileset grace limits, a two-element array (qi[]) of quotaInfoT structures (protected by the fileSetMutex), and a quotaStatus field to track quota state. quotaStatus is set to QSTS_DEFAULT upon fileset creation, and this value changes to QSTS_QUOTA_SYNC only while we are in the process of syncing quotas to disk.

The two-element array of quotaInfoT structures stored in the fileSetNode (qi[] field) is used to track info for each of the quota types. qi[USRQUOTA] is used to access the user quota data (array element 0) and qi[GRPQUOTA] is used to access the group quota data (array element 1). The quotaInfo structure contains pointers to the quota bfAccess structure and quota fsContext structure, fields to store the quota file tag number, the grace periods, flags, number of quota pages, credentials, and the qiLock which serializes adding storage to the quota file. The qiLock is protected by the filesetMutex in the fileSetNode structure. The flags field, qiFlags, is set to QTF_ENFORCED when quotas are being enforced on the fileset.

The constants MAX_FQ_TIME and MAX_DQ_TIME define the default amount of time given to a user before the soft limits are treated as hard limits (usually resulting in an allocation failure). The timer is started when the user crosses their soft limit and is reset any time they drop below their soft limit. These default values are used in the *qi_init()* function where the quotaInfo structures for both user.quota and group.quota are initialized. Both constants are equal to 1 week's time as measured in units of seconds. These grace periods can be reset to any value using the edquota utility.

10.3 Internal Functions for Maintaining Quotas

The following routines relate directly to quota maintenance and enforcement.

attach_quota (struct fsContext *cp, ftxHT ftxH) - Sets up quotas for a file. Setup entails adding pointers to up-to-date dQuot structures (filled in by *dqget()*) to the fsContext structure for each quota type (user and group). The quota limits and usage are reported in the fields dQBlk32 or dQBlk64 which are parts of a union inside the dQuot structure. The dQuot has flags to help determine if the 32 or 64 bit field should be used.

detach_quota (struct fsContext *cp) – Disassociates quotas from a file. *dqrele()* is called on each dQuot pointed to inside the fsContext structure. This function is always called from *vnode_fscontext_deallocate()*, which is called when we are trying to reclaim a vnode (due to vnode recycling or a filesystem unmount).

dqrele (dq) – Macro to call *dqput()*

dqput (struct dQuot *dq, int blkChng, int bfChng, int dirty, ftxHT ftxH) - Releases a reference to a dQuot. If the dQuot is dirty, *dqsync()* is called to flush the data to the appropriate on-disk quota file.

dyn_hash_remove() is then called to remove the `dQuot` from the hash table and its unique lock is terminated.

dqsync (struct dQuot *dq, int blkChng, int bfChng, int update, ftxHT parentFtxH) - Updates the on-disk quota file in the scope of a transaction before freeing a modified in-memory `dQuot`.

dyn_hash_remove (void * hashtable, void * element, int obtain_lock) – Removes a `dQuot` from the dynamic hash table of active quotas.

dyn_hash_insert ( ) – Inserts an in-memory `dQuot` element into a dynamic hash table of active quotas. The hash function is based on the pointer to the associated `quotaInfoT` structure and the quota's `fsContext` structure pointer.

quota_init ( ) - Called from *msfs_init()* (general startup), to register a transaction agent and initialize a dynamic hash table for `dQuot` entries.

quota_files_init (bfAccess *uqbfap, bfAccess *gqbfap, uint32T quotaStatus, ftxHT parentFtxH) - Initializes the quota files for a new fileset. For each quota file (user and group), page 0 is pinned. Then the appropriate `dQBlk32` or `dQBlk64` record structure within page 0 is pinned. The current number of files within the fileset (excluding the quota files) and the number of blocks in use (also excluding the quota files) are initialized within the `dQBlk32/dqBlock64` structure and the pages are unpinned. This is all done within a transaction.

advfs_quota_chown (struct vnode *vp, uid_t new_uid, gid_t new_gid, int flags, struct ucred *cred, ftxHT ftxH) - Transfers usage statistics from one owner to another. It does this with proper locking. The calculated change is performed via the *chk_bf_quota()* and *chk_blk_quota()* routines. The quota amount being transferred is decremented from the original owner first, then the amount is added to the new owner's quotas.

advfs_enforce_on (struct fileSetNode *dnp, int type) - Turns on quotas for a fileset. Called by the `quotaon` user command. Note that `quotaon` must be run after every startup to start quota enforcement for that fileset.

advfs_enforce_off (struct fileSetNode *dnp, int type, ftxHT ftxH) - Turns off quotas. Called by the `quotaoff` user command.

advfs_quota_sync (struct fileSetNode *dnp) – Noop function.

advfs_set_quota (struct fileSetNode *dnp, uint_t id, int type, int size, caddr_t addr, int kernaddr) – Sets fields in a `dQBlk` structure. *dqget()* is called to get the `dQuot`. If there was no soft limit previously or the group or user was under the soft limit, but now a soft limit exists and has already been passed, then the grace period will begin immediately. Once all fields are reset, the `dQuot` is flushed to disk with *dqput()*.

advfs_get_quota_info (struct fileSetNode *dnp, caddr_t addr, int kernaddr) - Returns the value of the `fileSetNode->quotaStatus` field. This field is set to QSTS_DEFAULT on a newly created fileset. The only other relevant status values are QSTS_QUOTA_SYNC (set when we are syncing quotas to disk) and QSTS_LARGE_LIMITS (set if 64 bit quotas are used).

advfs_get_quota (struct fileSetNode *dnp, uint_t id, int type, int size, caddr_t addr, int kernaddr) - Returns current values in a `dQBlk` structure stored at address `addr`. The caller specifies the quota type (USER or GROUP) and the size (32 bit or 64 bit). *dqget()* is called to obtain the `dQuot` from disk, then values found in the on-disk `dQuot->dQBlk` structure are stored in memory in the appropriately sized `dQBlk` structure at address `addr`.

advfs_set_use (struct fileSetNode *dnp, uint_t id, int type, int size, caddr_t addr, int kernaddr) - Called by the *quotactl()* application level function on behalf of a request for Q_SETUSE or Q_SETUSE64. This call is used to set current inode and block usage, which is not typically done except by the quotacheck utility to correct usage values. The caller specifies the type (USER or GROUP), the id (USERID or GROUPID), and the dQBlk size (32 or 64 bit). The function calls *dqget()*, updates the new `dQBlk` values, then calls *dqput()* to flush the new values to disk. During this update, if a soft limit is crossed, the grace period is started. Finally, the fileset quota counts are verified and updated in the filesetNode structures as appropriate, and grace periods are started if necessary.

chk_blk_quota (struct vnode *vp, long change, struct ucred *cred, int flags, ftxHT ftxH) – Verify that a block change to a particular file is within the quota range and update the quotas accordingly. If the requested allocation surpasses quota limits, undo the change. If the fileset does not have quotas turned on, EOK is returned immediately. If quotas are enabled, then calculate the proposed change, adding or subtracting blocks from the current values to come up with the difference between new and old block values. A calculated change which is negative indicates that the quota file is no longer reporting correct values and we may underflow. The incorrect values may be due to an incorrect quota calculation or a corruption, and now the `quotacheck` utility must be run to correct the accounting (the user is given a warning message to run `quotacheck`). In the mean time, the `dQBlk` value is set to zero rather than performing negative accounting with unsigned values, and a flag is set to avoid reporting a warning message twice.

If the change request looks valid, we copy it into the appropriate `dQBlk` field, overwriting the previous value. If the change was positive and neither the `FORCE` nor `ESCAPE_QCHECK` flags were set, then *chk_blk_quota_chg()* is called on the value. If *chk_blk_quota_chg()* returns with a failure then our original change request is backed out and a *dqsync()* is performed. If *chk_blk_quota_chg()* returns successfully, we add or subtract from the `dQBlk` as expected. *dqsync()* is then called, passing in the change so that an undo record can be created. If *dqsync()* fails, an undo is performed on the changes made in the `dQBlk`. If *dqsync()* succeeds, the fileset `blksUsed` and `filesUsed` fields are updated. If during this update an underflow is detected, it is reported, the field is set to 0, and a flag is set to avoid repeating these steps again.

The `FORCE` flag mentioned above is used to allow a deduction to occur, since values may temporarily increase but cause fragging on close, which will then cause a decrease. A decrease should always be allowed to take place. The `ESCAPE_QCHECK` flag is used to allow CFS to finish a cached operation because it has committed to it and can not do otherwise. The `ESCAPE_QCHECK` flag is employed when user or group quotas for a user might be exceeded.

chk_bf_quota (struct vnode *vp, long change, struct ucred *cred, int flags, ftxHT ftxH) - Verify that adding or deleting a file is within the quota range for a specified type and update the quotas accordingly. Identical to *chk_blk_quota()* but refers to file counts rather than block counts.

chk_quota_write (struct fsContext *cp) - Special check to ensure that writing is not being done directly to a `quota.user` or `quota.group` file.

get_clone_usage_stats (struct fileSetNode *dnp) - Since a fileset clone contains a snapshot in time of all files in a fileset, it also needs to have its own quota files. This routine assists in determining in-use blocks and files for the clone fileset.

quota_page_is_mapped (struct bfAccess *bfap, unsigned long id) - Specialized routine to increase performance by avoiding taking locks if not necessary. To synchronize with migrate, check to see if a quota page is mapped by a real page.

Chapter 11: Data Management API (DMAPI)

11.1 Introduction

Today's computing environments are characterized by an ever-increasing demand for data storage capacity. Large amounts of data are stored on UNIX-based servers, and the costs associated with managing the storage subsystems have been significantly higher than the cost of the storage itself. There is an ongoing need for intelligent and efficient storage management.

Over the years, a variety of **data management** applications have been developed, including various hierarchical storage management applications, data migration applications, unattended backup and recovery, and various on-line data compression schemes. We can also include in this category various enhanced data security applications such as automatic data encryption. Throughout this chapter, such data management applications are referred to as **DM applications**.

Chapter 12: Lock Management

12.1 Overview

This chapter is about various types of locks used throughout AdvFS. We will attempt to explain the different types of locks and why they are used, some general rules when using locks, how to detect some common locking problems, and an inventory of the AdvFS locks and what they protect.

In general terms, locks are used to prevent different threads from using or modifying a common resource concurrently. For instance, if you had a multi-threaded application in which a certain routine were run by each of the threads, and you needed to know how many times the routine had been run, you would probably insert a counter into the routine that would be incremented each time the routine was executed. A problem can arise however if two threads attempt to increment the counter at the same time. To solve the problem, you would seize a lock, increment the counter, and then release the lock. If a second thread attempted to seize the lock while it was held by the first thread, then it would block waiting for the lock to be released by the first thread. This would ensure that the counter was incremented exactly one time per execution of the routine.

Before jumping into the locks, there are several terms that need to be explained. The act of getting access to a lock is called **seizing**, **asserting**, or **locking** that lock. In this state, the lock is said to be **seized**, **asserted**, **held**, or **locked**. Releasing the lock is called **releasing** or **unlocking** the lock. Other terms will be introduced as they are needed in the following sections.

12.2 Lock types and their uses.

12.2.1 Simple Locks

Simple locks are also known as **spin locks** and **mutexes**. This is basically just a flag at a known memory location that indicates if the lock is held or not. It can only be held by one thread at a time. If one thread holds the lock, and another tries to seize the lock, the second thread will 'spin' on the processor until it is granted the lock. This is why it is called a spin lock. While the thread is spinning, it is consuming processor cycles.

In the Tru64 UNIX code, the simple lock is a structure of type `simple_lock_t` which is really just an `unsigned long` word. The state of a simple lock can be viewed with the `crash` utility by using the `locksig <lock>` command. This will show the caller and the state of the lock.

The advantage of this type of lock is that it has relatively low overhead; there is not a lot of code needed to seize, wait for, or release the lock. The disadvantage of this kind of lock is that the processor is busy while threads are waiting to seize the lock. It might be more efficient, from a system point of view, if the threads that had to wait for a lock were put to sleep and another thread were allowed to run on the processor. In this case, a complex lock might behave better, as we shall shortly see. Another disadvantage of this lock type is that it should only be held for very short periods of time. This is to prevent other threads from spinning and wasting time on other processors while this lock is held. What is acceptable? Try to limit to the scope of simple locks to several lines of code. More on this later.

12.2.2 Complex Locks

A complex lock is also called a **read/write** lock because it can be seized for reading or writing. There can be many readers, but only one writer. This means that if it is seized for reading, the lock can be shared among readers, but will block any writers. Similarly, if this lock is held for writing, all other readers and writers will block until the lock is released.

This lock, of type `struct lock`, contains a simple lock, plus several additional fields to record state, number of readers, and indexes into a `lockinfo` array. There is also an `l_lastlocker` field that contains the thread that was last granted the lock; this is often useful in debugging.

The advantage of this kind of lock is that a thread attempting to seize it will block (be put to sleep) until the lock is available. This means that processor cycles are not wasted on a thread that is merely waiting for a lock. Another advantage is the fact that the lock can be shared by multiple readers, but will block all other threads if the lock is seized for writing. Unlike the simple lock, this lock can be held for relatively long periods of time. The biggest disadvantage to this type of lock is that there is greater overhead than with simple locks. In fact, there is a simple lock embedded inside the complex lock that guards its internal fields.

12.2.3 Special AdvFS Lock Types

AdvFS designers added several other special lock types. The first three are the state lock (`stateLkT`), the buffer lock (`bufLkT`), and the ftx lock (`ftxLkT`). All are implemented as a common lock header structure (`lkHdrT`), plus some additional state fields. The lock header contains a simple lock to guard the state fields, and a type field to identify the lock. There is also a chain pointer, but it is used only by the ftx lock version of these locks.

These locks can have many different 'states', and the code can determine what to do when the lock is found to be in each of the different states. This makes this type of lock more flexible than a read/write lock, while retaining the relatively low overhead of the simple lock. The disadvantage of this type of lock is that AdvFS must maintain all the code for their use.

As of Tru64 UNIX Version 5.1B, there are 3 state locks (`bfAccessT.stateLk`, `vdT.active`, and `bfSetT.cloneDelState`), one buffer lock (`bsBuf.bufLock`). and 9 ftx locks (`bfAccessT.xtntMap_lk`, `bfAccessT.mcellList_lk`, `bfSetp.dirLock`, `bfSetp.fragLock`, `vdT.del_list_lk`, `vdT.stgMap_lk`, `vdT.mcell_lk`, `vdT.rbmt_mcell_lk`, and `domainT.BfSetTblLock`).

Ftx locks are unlike all other locks in one special way. All other locks tend to be seized and released in the same code path, meaning that most calls to seize a lock will be paired with a call to release the lock. This is not true for the ftx locks. These are seized at some point during a (sub)transaction, but are never explicitly released. They are associated with a given transaction structure, and are released when that (sub)transaction commits or aborts.

Active range locks are another mechanism unique to AdvFS that are used to synchronize threads acting within a specific range of the file. They are used primarily by direct I/O routines to synchronize with threads that tend to bring pages into the buffer cache, as well as to prevent direct I/O reads and writes from acting on the same disk sectors simultaneously (see section 8.12.2). Active ranges have some characteristics of both spin locks and complex locks. They are always seized for exclusive use; but if one thread holds the range, other threads will sleep while waiting to be granted the range. One advantage of active range locks is that they can be seized by one thread and released by another. This must be done when a directIO read/write is done via the AIO interface. One thread seizes the active range, starts the I/O, and then returns to the caller. The I/O is completed and the active range is released in the context of

the I/O completion thread. One disadvantage of the active range locks is the fact that all the code is maintained in AdvFS code space. The routine `insert_actRange_onto_list()` is used to seize an active range, while `remove_actRange_from_list()` is used to release it.

12.2.4 Other Kernel Lock types

There are several other lock types that you may encounter in the kernel. The first is a probe, and this is a special lock type that was used throughout AdvFS for evaluation of testing code coverage. There were tools that allowed the tests to be run, and then evaluate how many times each probe had been passed. From this, it could be determined whether the tests were adequately testing the AdvFS code paths. This mechanism was deemed to be obsolete in the original Version 51B (wildcat) project and was removed. However, the wildcat changes were backed out before the wcalphaos project changes were implemented, and the probes are still in place. We are currently removing the probes on a case by case basis. When changes are made to existing routines, the submitter also removes the probes associated with that routine. Brian Tsao put together a document on the steps necessary to remove a probe. Those guidelines can be found on `anw:/usr/specs/filesystem/advfs/probeRemoval.html`.

The second lock type is the read/write spin lock that was implemented in Tru64 Version 5.0. These have a type of `rws_lock_t`, and are not currently used by AdvFS.

The third lock type is the MCS or queued lock that is used only on NUMA-systems. These are needed on NUMA boxes because of the unfair access to memory. CPUs local to the memory can starve out non-local CPU's if a simple spin lock is used. MCS locks are used for NUMA machines by default, but this can be overridden. The `sysconfig generic` variable called `locktype` is used to control this; if set to 0 will specify spin locks and if set to 1 will specify MCS locks. (A value of 1 is ignored on non-NUMA platforms.)

12.2.5 Locking at the utility/library level

There are quite a few utilities and commands that perform an operation on one particular domain, but only one instance of any of these utilities should be running at one time on the domain. Locking at this level is kept to a minimum so that filesystem access is not limited by administrative maintenance to the system.

The motivation for the synchronization is two-fold:

1. Some of these utilities work towards the same result, either directly or indirectly. For example, doing a `rmvol` and `defragment` simultaneously on the same domain will both move file extents among disks and reduce fragmentation in the domain. Also, it is a waste of machine time to defragment a disk that this being removed from the domain.

2. The lack of synchronization may cause a utility to fail. For example, if `rmvol` is run at the same time as `defragment`, when `rmvol` successfully removes a volume, the defragmenter will fail when it tries to reference that volume. Even more serious, a kernel failure may result if a `vd` pointer is referenced after that volume has been removed and the pointer has become invalid.

For the purpose of synchronization, we divide the utilities and commands into three categories:

Fast	addvol, chfsets, chvol, clonefset, migrate, mkfset, renamefset, rmfdmn, switchlog
Slow	balance, defragment, rmfset, rmvol, verify
Special Case	mkfdmn, mount_advfs, showfdmn, showfsets

The theory behind this categorization is that if a fast utility is running and another utility (X) wants to run, then X should just wait, because the fast utility will be done soon. But, if a slow utility is running and X wants to run, X should print an error message and exit, because the wait would be too long.

The exceptions to these rules are the special case utilities. When executed first, they are treated like fast utilities, in that they will cause subsequently-run utilities (both fast and slow) to block and wait until completion. But, if a slow utility is already running, the special case utility will not wait or exit with a failure – it will execute as normal.

This behavior is implemented by locking the `/etc/fdmns` and `/etc/fdmns/<domain-name>` directories with special protocols. These are explained next.

Fast Utility Locking Protocol

First lock `/etc/fdmns`, by calling `lock_file`. This causes a wait if the lock is already held. Then lock `/etc/fdmns/<domain-name>` by calling `lock_file_nb`. If the lock is already held, the utility will exit with an error message indicating that another utility is claiming exclusive use of the domain.

With the above two locks held, the bulk of the utility work and calls into the kernel are completed. Since this is a fast utility this work should not take a long time. Lastly, the file locks are unlocked explicitly or by process rundown.

Slow Utility Locking Protocol

First lock `/etc/fdmns`, by calling `lock_file`. This causes a wait if the lock is held. Optionally, do a small amount of work. Then lock `/etc/fdmns/<domain-name>` by calling `lock_file_nb`. If the lock is already held, an error message is printed and the utility exits. Optionally, do a little more work. Next unlock `/etc/fdmns` by calling `unlock_file`. Now do the major part of the work (this should be the only step that takes a significant amount of time). Lastly, unlock `/etc/fdmns/<domain-name>` by process rundown.

These protocols ensure that subsequent utilities find `/etc/fdmns/<domain-name>` held when a slow utility is already running, causing the latter utility to exit with an error. However, fast utilities hold both locks for their duration, which will cause a subsequent utility to block on `/etc/fdmns` when it attempts to lock that first.

Special Case Utility Locking Protocol

The special case commands are done differently because it is often convenient to run them (especially showfsets and showfdmn) to monitor the progress of a utility.

First lock /etc/fdmns, by calling lock_file (in most cases with the LOCK_SH flag). Do the small amount of work necessary for this utility. Unlock /etc/fdmns, explicitly or by process rundown. This protocol allows concurrent access across domains, except for the duration that /etc/fdmns is locked, which is very short.

Note that the /etc/fdmns directory lock and its subdirectory domain locks are only advisory: they don't actually prevent any action from occurring on the domains and files they lock, so it's important that all utilities working on a specific domain follow this scheme.

12.3 Good things to know when using locks

There are some simple things to understand when using kernel locks.

- In AdvFS there is a naming convention in which global variables (and locks) have the first letter of the name capitalized. Non-global variables have the first letter in lower case. Therefore, the FtxMutex and BfAccessFreeLock are global locks, while the bufLock and bfIoLock are locks with a smaller scope. One lock that breaks this rule is the domainT.BfSetTblLock. I believe this was a global lock at one point, and the name was not changed to adhere to the convention when it was moved into the scope of the domain.

- The kernel does not like to return from a system call with either simple or complex locks held. There are lock counters in the thread() and pcb() structures, and, if not zero, will cause a panic when returning from kernel code. There is a lock_disown()/lock_adopt_lock() set of routines that can be used to avoid this problem, but AdvFS does not use them. There is one situation where AdvFS does return from the kernel mode with a lock held, and this is in the case of a read/write using directIO and AIO. In this case a file range is locked by the thread that starts the I/O, and the thread that completes the I/O must release the lock. This path avoids the problem by using active range locks instead of simple or complex locks.

- A thread may not hold a simple lock and then sleep or block for any reason. This could cause indefinite holding of a simple lock.

- Similarly, a light-weight context thread cannot block, and therefore cannot use complex locks. There is no real wake-up mechanism for this case. In AdvFS, the I/O completion path through bs_osf_complete() is executed in a light-weight context.

- When a thread must seize more than one lock, it must do so in hierarchy order to prevent **deadlocks**. All kernel locks are declared in the file lockinfo.c, with locks earlier in the file being higher in the hierarchy, and therefore seized first. A deadlock is a situation where 2 or more threads block each other's ability to proceed. Consider the following example. Thread 1 seizes lock A and then attempts to seize lock B. Meanwhile, thread 2 has lock B seized, and attempts to seize lock A. In this situation, both threads are deadlocked and will never be able to proceed. By adhering to the hierarchy, both threads will seize lock A and then lock B. A single thread can deadlock with itself if it holds a lock and attempts to seize that lock again in a non-recursive fashion. When running in lockmode == 4, an error will be generated if a thread seizes locks in an out-of-hierarchy order. If a thread must seize locks in an out-of-hierarchy order, it can do so safely by using the lock try mechanism. For complex locks this includes the lock_try_read() and lock_try_write() routines, and for simple locks, the simple_lock_try() routine. When these routines are called, the lock will be seized if it is not already held, and will not be seized if it is already held.

- There are a variety of other lock routines to help avoiding locking problems. The recursive lock routines allow a single thread to seize a lock multiple times without deadlocking itself. Normally if a thread holds a particular lock, and then attempts to seize the same lock, it will wait forever for that lock to be released, and is said to be deadlocked with itself. The recursive lock routines help to avoid this sometimes-unavoidable situation. If a lock is seized recursively, it must be released as many times as it has been seized. Routines such as `lock_islocked()` tell if a lock is already locked, and `lock_holder()` will tell if the current thread already holds a lock. For complex locks, a lock can be upgraded from a shared lock to an exclusive lock by using `lock_read_to_write()`, and can be downgraded by using `lock_write_to_read()`. Look through `lock.c` for various flavors of these routines.

- In `lockinfo.c`, there are flags of value ORDERED and ORDNEXT that are sometimes used, and bear mentioning. If there is a class of lock for which many locks actually exist and can be held at the same time, then the ORDERED flag is specified, and indicates to the kernel that the application has determined an ordering for seizing locks within this class to prevent deadlocks. For instance, the `logDescT.descLock` is ORDERED because two can be held simultaneously without hierarchy violation, and the code in `lgr_flush_start()` explicitly avoids deadlocking by using the lock try mechanism. A different method for avoiding a deadlock is used for the `bfAccessT.bfaLock` which is also ORDERED. In this case, a thread will seize the `bfAccessT.bfaLock` for the clone file before seizing the same lock for the original file in `bs_access_one()`. It really doesn't matter what mechanism is used to prevent deadlock, just that there is such a mechanism.

- The ORDNEXT flag for a lock class indicates that the lock is at the same hierarchy level as the next lock declared in `lockinfo.c`. This is used to avoid lock hierarchy violations when `lockmode == 4` and there is some external synchronization mechanism between the two lock classes to avoid deadlocks.

12.4 Debugging lock usage

12.4.1 Using the lockinfo command

Some of this information is also available in the Tru64 System Configuration and Tuning Guide.

Without showing all the options available for this command, the command is run as:

```
lockinfo <command> <command-args>
```

When you enter a lockinfo command, the utility first opens the `lockdev` pseudo driver and turns on lock statistics gathering. Then, the utility forks and executes the specified command. After the command completes, the utility turns off lock statistics gathering (closes `lockdev`), collects the data, and sends it to `stdout`. The output data shows the locking done by the operating system during the execution time for command.

To gather statistics with `lockinfo`, follow these steps:

1. Start up a system work load and wait for it to get to a steady state.

2. Start `lockinfo` with `sleep` as the specified command and some number of seconds as the specified `cmd_args`. This causes `lockinfo` to gather statistics for the length of time it takes the `sleep` command to execute.

3. Based on the first set of results, use `lockinfo` again to request more specific information about any lock class that shows results, such as a large percentage of misses, that is likely to cause a system performance problem.

For example, the following command causes `lockinfo` to collect locking statistics for 60 seconds:

```
lockinfo sleep 60
```

The output from this command will look similar to that shown in Table 12.1.

Table 12.1 Sample output from the lockinfo command.

```
hostname:          sysname.node.corp.com
lockmode:          2  (SMP default)
processors:        4
start time:        Wed Jun  9 14:38:05 1999
end time:          Wed Jun  9 14:39:05 1999
command:           sleep 60

tries    reads     trmax     misses    percent   sleeps    waitmax   waitsum
                                       misses              seconds   seconds
bsBuf.bufLock  (S)
5718642  0         45745     194509    3.4       0         0.00007   0.63226
lock.l_lock  (S)
5579643  0         40985     75656     1.4       0         0.00005   0.16531
thread.lock  (S)
1989132  0         24817     21795     1.1       0         0.00003   0.03864
vnode.v_lock  (S)
1578583  0         49207     1527      0.1       0         0.00002   0.00443
.
.
inifaddr_lock  (C)
1        1         1         0         0.0       0         0.00000   0.00000

total simple_locks = 28545191                  percent unknown = 0.0
total rws_locks = 1429                         percent reads = 100.0
total complex_locks = 2764296   percent        percent unknown = 0.0
```

The first six lines of output specify the system, its `lockmode` attribute setting, how many processors it has, the start and end times of the statistics gathering, and the command that was run. Next, a table with statistics data for each lock class is displayed. In this example, the lock class entries in the table are sorted by the number of tries. This is the default sort order, but can be changed using the `-sort` option. (Note that the names of the locks are the strings used to declare each lock in `kern/lockinfo.c`). The data in each column are explained in this table:

Column Heading	Meaning
`tries`	The number of tries for asserting (seizing) the lock.
`reads`	The number of attempts on read.
`trmax`	The maximum number of readers in the critical path (for a C class lock) or the maximum number of cycles spent holding the lock (for an S, RWS, and MCS class lock).
`misses`	The number of lock misses.
`sleeps`	The number of blocks encountered while waiting for the lock.
`waitmax`	The maximum amount of time (in seconds) spent waiting for a lock.
`waitsum`	The total (in seconds) of all times spent waiting for the lock.
`percent misses`	The lock miss percent.

When diagnosing a system problem, certain statistics are more important than others. The following results may indicate a problem:

- A large number of tries. This is simply the number of times the lock was attempted to be seized. If this number is large but the path being tested seizes and releases this lock many times, this may not indicate a bug. However, it would probably be beneficial to consider the locks being taken and whether they are needed or can be seized fewer times.
- A percent misses value that is too high. "Too high" varies somewhat, depending on the lock class. A kernel developer who is testing VM code under development might consider any percentage over 1 percent "too high" for certain kinds of locks. A support representative testing released product software should look for percent misses values that exceed the range of 5 to 7 percent.
- A large waitsum value. This it the total time that threads spent waiting for this lock to be released, so a large value here indicates a lot of inter-thread lock contention. Perhaps the lock can be held for shorter periods of time in certain paths to reduce this contention. Using the `-class` option for `lockinfo` may be useful in determining which paths generate the most contention. This will be discussed next.

In the first example of `lockinfo` output, a few locks show high values in the `tries`, `percent misses`, and `waitsum` columns. The following example uses `lockinfo` to show the code paths where one of these, `bsBuf.bufLock`, is asserted with high frequency:

```
lockinfo -class=bsBuf.bufLock sleep 60
```

```
hostname:       sysname.node.corp.com
lockmode:       2  (SMP default)
processors:     4
start time:     Wed Jun  9 14:38:05 1999
end time:       Wed Jun  9 14:39:05 1999
command:        sleep 60

Locks asserted by PC for lock class:              bsBuf.bufLock

count    miss            caller    line #            return     line #
-------------------------------------------------------------------------
733418   41275  bs_pinpg_one_int:  4494        bs_pinpg_clone:  4125
704579   49182  bs_pinpg_one_int:  4460        bs_pinpg_clone:  4125
697209   0             find_page:  5986      bs_pinpg_one_int:  4368
680910   30359        bs_unpinpg:  5461    log_donerec_nunpin:  3149
544828   12869         bs_q_lazy:  2006             bs_q_list:  1142
496294   15537         find_page:  5670    log_donerec_nunpin:  3149
.
.

tries    reads  trmax    misses    percent  sleeps    waitmax    waitsum
                                   misses             seconds    seconds
bsBuf.bufLock
5322157  0      45745    366848    6.9      0         0.00009    1.77041
.
.
```

For the `bufLock`, we could conclude that there is no contention for this lock when called from *find_page()* in the *bs_pinpg_one_int(),* but that there is some contention (6.3%) when locked in *bs_pinpg_one_int()* directly.

Based on the information returned about high frequency code path assertions for the `bsBuf.bufLock` lock, the kernel developer can then look for ways to reduce the amount of locking for this class. Strategies for reaching this goal might include one or more of the following:

- Changing the code to use a read/write spin lock rather than a simple spin lock.
- Reducing lock hold times by moving some work outside the time that the lock is being held.

- Making more radical changes in kernel algorithms to reduce the frequency of lock assertions or the amount of time that locks are held.

12.4.2 Lock Mode

In order to have kernel lock debugging enabled, the `lockmode` that the system is running under must be set to 4. This is a `sysconfig generic` variable. See the `sys_attrs_generic(5) man` page for more information about the `lockmode`, `locktype`, and `locktimeout` configurable variables when debugging lock issues. The `lockinfo` command will gather lock statistics for `lockmode` values of 2, 3 or 4. Note that the simple lock macro SLOCK_HOLDER() that returns true if the current thread (cpu) holds the simple lock, otherwise false, will only work as expected when running with a lockmode value of 4. For all other lock modes, it acts the same as SLOCK_LOCKED(), returning true if the lock is held by any thread or cpu. Usually this behavior is OK, but beware of this when tracking down problems if there is a question of whether a thread is correctly synchronizing itself using simple locks and this macro on a system in running lower lock modes.

12.4.3 Detecting deadlocks

Deadlock detection is usually straight-forward. The primary symptom is a thread or set of threads that are hung attempting to seize a lock. A single thread can also deadlock with itself if it holds a lock and attempts to seize it again in a non-recursive fashion. To determine if two threads are deadlocked, select one thread and determine which locks it currently holds, and which lock it is attempting to seize. Do the same for the second thread. If the locks that each are attempting to seize are held by the other, then the threads are deadlocked. One code path must be changed either to seize the locks in the correct order, or to use a lock-try type of mechanism.

Determining which locks a thread holds can sometimes be tricky, since there is no linked list of locks per thread. The number of simple locks held by the thread is stored in `pcb.pcb_slock_count`, and the number of complex locks held is kept in `thread.lock_count`. Using these values will help you figure out which set of locks need to be evaluated. Obviously only one thread can hold a simple lock at a time, and simple locks are held for short periods, so if there are simple locks held, it is usually possible to determine these by looking back through the recently executed code path. The same thing can be done for complex locks, but there is a way to verify these. The `lock.l_lastlocker` field will hold the thread that last locked the lock. If there are many readers holding the lock, this may not help a lot, but if the thread holds the lock exclusively, then its thread pointer will be in this field.

Don't forget that deadlocks can occur between any kind of resource, so state locks and active ranges can also contribute to the deadlock. Deadlocks involving state locks may have to be inferred by the code path involved since there is no thread tracking unless the kernel has been compiled with the ADVFS_DEBUG directive enabled. Active ranges contain an `actRange.arState` field that indicates what type of action (migrate, direct I/O, etc.) has taken the range, so this can be helpful for determining whether the range is involved in the deadlock.

12.4.4 Determining if there is excessive lock contention

This is usually determined by large `waitsum` values generated for the lock class by the `lockinfo` command. If there are a number of paths seizing this lock, you may need to run the `lockinfo` command with the `-class` option as shown above, and then examine each of the code paths to determine which are holding the locks for long periods and contributing to the contention.

12.4.5 Determining if a lock is held for excessive time.

This section is focused toward simple locks. A complex lock or the various state locks can be held for an indeterminate amount of time, so that analysis really becomes detection of excessive lock contention which has already been discussed in section 12.4.4. Theoretically, simple locks should be held for short periods of time. If a thread spins for more than a configurable number of seconds while waiting for a simple lock, then the thread holding the lock is violating this rule. Most times when this is encountered it is a coding error and the lock release was omitted in a code path. At other times, it is because the scope of work done while holding the simple lock is too large or indeterminate.

If a thread waits more than 15 seconds for a simple lock, and the kernel is running in `lockmode == 4`, then the message "`simple_lock: time limit exceeded`" is generated and the kernel panics. (Actually, the 15 seconds timeout value is the default value for the `sysconfig` variable `locktimeout`. This value can be changed, but do so with caution; see `sys_attrs_generic(5)` `man` page). The panic gives the Program Counter (pc) for the thread that is attempting the lock, the lock address, and the state of the lock. In this case, the thread that detects the condition is not the thread that is holding the lock. However, the `slock.sl_data` field will contain the low-order 32 bits of the instruction that called the lock routine. This gives you a routine and line number of the code path that seized the spin lock. Searching for a thread that is in this code path usually turns up the thread that is holding the lock. As a bonus, the code path that this thread is in is the path that has held the lock for an excessive amount of time.

To proactively tell what simple locks are being held for long periods of time, you can set up the lock to timeout if it is held too long. To do this, set the `sysconfig generic` variable `lockmaxcycles` to the number of cycles in, for example, one second. Then, if the lock is ever held for more than one second, the lock routines will panic the system, and you will be able to see which lock and path held the simple lock for this amount of time. Use this with caution, and only in a test environment. To determine what value to set this variable to, determine the number of cycles per second for your machine. Use the crash command `rpb -p`. This will show the processor type and speed for each cpu. Set the value of `lockmaxcycles` to the speed of the cpu to get a one-second timeout value. For example, issue the `rpb -p` command:

```
crash> rpb -p

CPU 0:  PROCESSOR: EV56 (21164A) pass 2 (532 MHz)

CPU 1:  PROCESSOR: EV56 (21164A) pass 2 (532 MHz)
```

Then reset `lockmaxcycles` to the number of cycles per second:

```
sysconfig -r generic lockmaxcycles=532000000
```

This would cause a lock panic if any simple lock were subsequently held more than one second. The timeout value you select may vary. Note that the resulting panic will be in the context of the thread that is holding the lock. There is no problem of getting a lock panic generated by one thread for a lock actually held by another thread as we saw previously.

Another way to see how long simple locks are being held is to examine the `trmax` values for a lock generated by the `lockinfo` command. For example, in Table 12.1 the `bsBuf.bufLock` has a `trmax` value of 45745. If we are running on the same machine as we ran the `rbp -p` command, we know that

this is a 532 MHz machine, so divide the `trmax` value by the clock rate in MHz to give the number of microseconds that the lock class was held. For the `bsBuf.bufLock` in Table 21.1, this is (45745/532) or 85 microseconds.

12.5 AdvFS Lock Inventory

The following sections detail the various AdvFS locks in Tru64 UNIX Version 5.1B. For some locks some additional notes have been added. These notes can be expanded as more information becomes available.

12.5.1 Domain Locks

There are a number of locks that work on the `domainT` level. These are summarized in the following table.

Lock	Lock Type	Scope	Comments
DmnTblLock	Complex	System	Synchronizes domain activation / deactivation
DmnTblMutex	Simple	System	Guards domain lookup (along with the DmnTblLock)
rmvolTruncLk	Complex	domainT	Serializes file truncation and volume removal.
ftxSlotLock	Complex	domainT	Checks for hierarchy violations between locks & starting a root ftx. (Debugging Only)
scLock	Complex	domainT	Guards service class table
xidRecoveryLk	Complex	domainT	Guards the domainT.xidRecovery structure which is used by CFS to hold xid recovery status information.
mutex	Simple	domainT	Mutex protecting vd.mcell_lk, vd.stgMap_lk, totalBlks, freeBlks, vd.ddlActiveWaitMCId, bfSetHead, and bfSetList
lsnLock	Simple	domainT	Guards fields related to the lsnList
vdpTblLock	Simple	domainT	Guards the domain's vdpTbl[] array, maxVds, and vdCnt
BfSetTblLock	Ftx Lock	domainT	Guards bfSetT in the domain, particularly during fileset state changes (open, creation, deletion, etc.)
dmnFreezeMutex	Simple	domainT	Guards dmnFreeze* fields.
ssDmnLk	Simple	domainT	SmartStore Lock
ssDmnHotLk	Simple	domainT	SmartStore Lock

There are two domain locks that may bear investigation: the use of the `DmnTblLock` and the `DmnTblMutex` appear to overlap. For instance, in `domain_lookup()` there is a check that either the `DmnTblMutex` or the `DmnTblLock` is held. Why isn't there just one lock for this?

12.5.2 BitfileSet Locks

Lock	Lock Type	Scope	Comments
FilesetLock	Complex	System	Guards FileSetHead list of mounted filesets and the fileSetNodeT.fsNext and fsPrev chain fields.
LookupMutex	Simple	System	Synchronizes threads inserting, deleting, or looking up bfSet descriptors in the bfSet table. Guards the BfsFreeListStats structure.
fragLock	Ftx Lock	bfSetT	Guards the manipulation of the frag file.
dirLock	Ftx Lock	bfSetT	Guards the bitfile set's tag directory
setMutex	Simple	bfSetT	Mutex for guarding the fragLock and dirLock
accessChainLock	Simple	bfSetT	Guards the chain of access structures in the bfSetT structure.
cloneDelStateMutex	Simple	bfSetT	Mutex to guard cloneDelState and the xferThreads field in the bfSetT
cloneDelState	State Lock	bfSetT	State of clone fileset deletion
bfSetMutex	Simple	bfSetT	Guards bfSetT.bfSetFlags field.
filesetMutex	Simple	fileSetNodeT	Guards the quota and fileset statistics fields in the fileSetNodeT structure.

12.5.3 Device Locks

Lock	Lock Type	Scope	Comments
ioQLock	Simple	ioDescHdrT	Protects the I/O queue chain; one lock per I/O queue. There are 24 I/O queues per device.
vdIoLock	Simple	vdT	Guards vdT.syncQIndx and vdT.error* fields
vdStateLock	Simple	vdT	Guards the vdT.vdState and related fields.
ddlActiveLk	Complex	vdT	Synchronizes DDL activity on a vdT.
active	State Lock	vdT	State flag for whether I/Os are outstanding on device. Protected by the vdT.vdIoLock.
mcell_lk	Ftx Lock	vdT	Guards vdT.nextMcellPg
rbmt_mcell_lk	Ftx Lock	vdT	Guards mcell allocation from the rbmt mcell pool.
stgMap_lk	Ftx Lock	vdT	Guards free-space cache (vdT. freeStgLst).
del_list_lk	Ftx Lock	vdT	Guards disk's Deferred Delete List (DDL).
ssVdMsgLk	Simple	vdT	Guards ssVdMsgState field
ssVdMigLk	Simple	vdT	Guards migrate fields in SS volume information
ssFragLk	Simple	vdT	Guards frag list in SS volume information

12.5.4 Logging/Transaction Locks

Lock	Lock Type	Scope	Comments
FtxMutex	Simple	System	Guards domainT.ftxTbld
descLock	Complex	logDescT	Guards most fields in log descriptor and log statistics fields in the domainT.
flushLock	Complex	logDescT	Guards the fields in the log descriptor that govern which log pages are being written to disk.

12.5.5 File Locks

Lock	Lock Type	Scope	Comments
file_lock	Complex	fsContext	Protects the file during various operations; see paragraph below.
kdmLock	Complex	fsContext	Used by DMAPI code to enforce exclusive and shared access rights to the file.
fsContext_mutex	Simple	fsContext	Guards fsContext fields, particularly the statistics and flags fields.
bfIoLock	Simple	bfAccessT	Guards the I/O and buffer cache related fields in the access structure inclunding the dirtyBufList. The bsBuf.accFwd and accBwd fields for buffers chained onto this access structure are also guarded.
bfaLock	Simple	bfAccessT	Guards bfAccess structure fields such as bfVp, bfObj, refCnt, accessCnt, mmapCnt, stateLk, and bfState.
migTruncLk	Complex	bfAccessT	Guards a file from being migrated while storage is being added to, or removed from, the file.
trunc_xfer_lk	Complex	bfAccessT	Prevents a clone from being read while the original file is being truncated and the extents are being transferred to the clone.
clu_clonextnt_lk	Complex	bfAccessT	CFS-only lock protecting a clone.
cow_lk	Complex	bfAccessT	Used to serialize threads that are COWing pages with those attempting to examine or modify the extent maps of the original file.
stateLk	State Lock	bfAccessT	Contains current state of the bfAccessT structure.
xtntMap_lk	Ftx Lock	bfAccessT	Protects extent maps in bfAccessT
mcellList_lk	Ftx Lock	bfAccessT	Protects the on-disk Mcells.
actRangeLock	Active Range	bfAccessT	Locks a range of the file for kernel use. (see below)

The `fsContext.file_lock` is used for many reasons, including the following. It is seized for shared access: 1) while searching a directory, 2) while reading a page in `msfs_getpage()`, and 3) while syncing a file's data. It is seized for exclusive access while 1) adding or removing storage to a file, 2) changing bitfile attributes, 3) while inserting or deleting an entry in a directory, 4) on file write if the file is being extended or opened in IO_APPEND mode, and 5) when changing cache policy for a file (mmap, ADL, or directIO). The use of the `file_lock` in direct I/O paths is discussed in section 8.12.1.

The migTrunc lock protects a file from being migrated while storage is being added to, or removed from, the file and, conversely, it prevents storage allocation to or deallocation from a file during the time that the file is migrated. The migrate code path takes the lock exclusively, so while the file is migrated, there is very little else allowed on the file: it can be read, but that's about all that can be done to it. The storage allocation/deallocation paths take the lock shared, so they lock out migration but not each other. There is an ordering constraint: the migrate code path takes the migTrunc lock on a file and then starts a root transaction, provided the file is not a clone. But if the file is a clone, the migTrunc lock is taken after the root transaction is started. That creates an ordering dependency that has to be obeyed by the rest of the code, otherwise deadlocks might occur. The reasons for this constraint are technical: the clone file being migrated might have storage allocated to it while the migration is going on. Being able to manipulate the lock within a transaction allows bs_cow_pg() to coordinate storage allocation and migration, something that would be tricky (if not impossible) in the other order.

The use of Active Range locks is discussed in Sections 8.12.2 and 8.12.7.

12.5.6 Buffer Cache Locks

Lock	Lock Type	Scope	Comments
bufLock	Simple	bsBuf	Guards most fields within the bsBuf, but primarily the state.lock and the fields that identify the buffer (bfPgNum, tag, bfPgAddr).
lock	Buffer Lock	bsBuf	A state field in the buffer
rangeFlushLock	Simple	bsBuf	Guards rangeFlushT.outstandingIoCount. There may be multiple rangeFlushT's associated with a bsBuf.

12.5.7 Other Locks

Lock	Lock Type	Scope	Comments
InitLock	Complex	System	Used during AdvFS initialization to ensure that racing domain activations only do system-wide initializations one time.
TraceLock	Complex	System	Used for synchronizing trace events (Debug Only).
BfAccessFreeLock	Simple	System	Guards global access lists: ClosedAcc and FreeAcc
LockMgrMutex	Simple	System	Used only during AdvFS initialization when ADVFS_DEBUG is enabled. (Debug Only)
mutex	Simple	msgQT	Guards the message queue contents
dqhash_chain_lock	Simple	DqHashTbl	Guards DqHashTbl hash for dquot structs
dqLock	Complex	dQuot	Guards dQuot struct (quotas)
qiLock	Complex	Quota File	Serializes adding storage to the quota file.
ssListTpool.plock	Simple	System	Guards pool of Smart Store list worker threads
ssWorkTpool.plock	Simple	System	Guards pool of Smart Store general worker threads
ssStoppedMutex	Simple	System	Guards condition variable for shutting down SS.

Chapter 13: AdvFS System Calls and Utilities

13.1 Overview

This chapter is an overview of how some of the AdvFS utilities function. The utilities covered include `vdump`, `vrestore`, `rvdump`, `rvrestore`, `fixfdmn`, `verify`, `salvage`, `vfast`, `freezefs`, `thawfs`, `advscan`, and the `vods` tools. There is additional information on these utilities in the `man` pages and in Chapter 5 of the Tru64 AdvFS Administration Guide.

`Vfast` is a relatively new utility that runs in the background and attempts to maximize the performance within AdvFS domains by defragmentation and balancing loads across disks. `Vdump`, `vrestore`, `rvdump`, `rvrestore`, `freezefs`, and `thawfs` are used for file backup and restore operations. `Fixfdmn` ,`verify, and salvage` are used to check, and optionally fix, the consistency of domain metadata. `Advscan` is used to check the consistency between the `/etc/fdmns` directory and the data on volumes. The `vods` tools are used primarily by engineering to examine on-disk AdvFS data structures.

Sometimes customers wonder which of the tools we provide are most appropriate to use when they suspect or experience domain corruption. Here are a few guidelines.

- The `verify` utility is fairly old, and the use of `'fixfdmn -n'` should generally be used to check the consistency of domains. `Fixfdmn` checks more corruption cases than `verify`.
- If a domain is unmountable, then `fixfdmn` can be used to check and repair the metadata on that domain.
- If a domain is unmountable and cannot be repaired, `salvage` can be used as a method of last-resort to extract data from files in that domain. It does not fix any metadata or make the domain mountable. It is generally recommended to run `fixfdmn` before resorting to the use of `salvage`.

First, let's look at how a call from a typical application makes its way down into the kernel.

13.2 Trace of AdvFS System Call

The following is an example trace of an AdvFS system call (in this case a call from `rmvol`) from user space into the kernel.

`rmvol.c:` *main()*

- *remove_volume (dmnName, volName)*
  - *advfs_remove_volume()*
    - *advfs_syscall(ADVFS_OP_REM_VOLUME)*
      - *msfs_syscall(ADVFS_OP_REM_VOLUME)*
        - *msfs_real_syscall(ADVFS_OP_REM_VOLUME)*

- *msfs_syscall_op_rem_volume()*

Starting in user space, *remove_volume()* is called from *main()* within the `rmvol.c` file. *remove_volume()* is a local function found in `rmvol.c`, and from this routine we call into the libmsfs library by calling *advfs_remove_volume()*. The `libParamsT buf struct` is setup here, which is a union of all the AdvFS structures specific to each kernel call. In this case, we insert our domain and volume info into the `libParamsT.remVol` union structure. *advfs_syscall()* is then called, passing in an operation type of `ADVFS_OP_REM_VOLUME.` This is actually a wrapper function for *msfs_syscall()*. *msfs_syscall()* is called passing the `opType (ADVFS_OP_REM_VOLUME)`, a pointer to the `libParamsT` , and the size of `libParamsT.remVol` .

msfs_syscall() lives in `msfs/osf/msfs_syscalls.c`. If AdvFS is not installed on the machine, the system call will error out here. If AdvFS is installed, `MsfsSyscallp` will have been setup to point to *msfs_real_syscall()* in *bs_kernel_pre_init()*. This is how we get entry into the AdvFS kernel code that will only be followed if AdvFS is installed. `MsfsSyscallp` is defined as a pointer to a function. From here *msfs_real_syscall()* is called with the opType of the kernel call we are seeking, in this case `ADVFS_OP_REM_VOLUME`.

msfs_real_syscall(), located in `msfs/bs/bs_misc.c`, is where the actual call into the kernel occurs, through a huge switch statement in this function. In this function, the opType drops the 'ADVFS' part, so opType becomes `OP_REM_VOLUME`. A big switch statement is then used to match the opType, which then determines the specific kernel function will be called. In this case, the kernel function *msfs_syscall_op_rem_volume()* will be called after appropriate error checks are performed.

msfs_syscall_op_rem_volume() is located in `bs_misc.c`, and starts the kernel work of the original call.

13.3 vdump/vrestore

The `vdump` and `vrestore` utilities provide basic file system backup/restore functionality that is compatible at the functional level with UFS dump and restore.

Key Features of `vdump`:

- Performs full or incremental backups on filesets or clone filesets
- Backs-up any associated extended attributes (ACLs, property lists, etc.)
- Partial fileset backup/restore (directory level backup) is possible with `vdump/vrestore`
- Remote versions of the utility can backup/restore to/from a device connected to a remote machine.
- Can be used to backup other file systems such as UFS, NFS, etc., apart from AdvFS.
- Provides protection against data corruption by XORing the blocks and making it part of the save set.
- Compression of data blocks during backup.

Key features of `vrestore`:

- The backed-up fileset can be restored using a corresponding version of the `vrestore` utility.
- Browsing and selective restoration of data from the saveset is possible.

However, `vdump` and `vrestore` differ from dump and restore in the following ways:

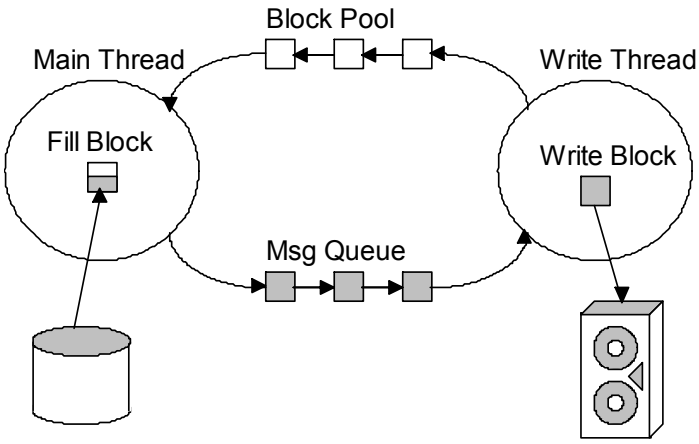
- `Vdump` works at the logical file system level rather than in terms of inodes (which dump does). `Vdump` scans the directories for files and uses regular POSIX file system calls to access directories and files. `Vdump` can also be used to backup non-AdvFS file systems.
- `Vdump` provides a tape format that is incompatible with dump/restore.

13.3.1 Vdump Basic Design

`Vdump` performs two passes through the directory hierarchy of the fileset being backed up. In the first pass, it creates a **saveset** which is a set of data describing the data to be backed up (see Section 13.2), and calculates the number of bytes to be backed up by looking at each file's size. In the second pass, `vdump` backs up the file attributes and actual file data.

In order to improve backup performance, `vdump` implements buffered I/O using two threads. The main thread traverses the directory tree, reading file data blocks and generating saveset blocks. When a saveset block attains the specified block size, which is tunable, the main thread sends a message to a write thread with a pointer to the block to be written. When the write thread receives the message, the saveset block is written to the destination media. In parallel, the main thread allocates a new saveset block and continues to read in data to be backed up. Concurrent read and write operations by these two threads improve the overall performance.

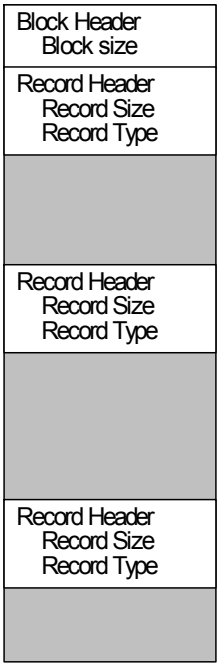
The following figure and description shows how the process operates.



1. Blocks move from the buffer block pool to the main thread where the block is filled with data read from the source (disk).
2. The block is moved to the message queue.
3. The write thread writes the now-full data block to a saveset (tape).
4. The block is put back in the buffer block pool for reuse.

13.3.2 Saveset Format

A saveset is an array of fixed-size blocks. The valid range of block sizes is 2KB to 64KB; the default is 60KB. Each block consists of a block header and one or more variable length records. Each record consists of a fixed length record header and a variable length data section. Record types include: file attributes, file data, directory data, and property lists. A block's unused space is accounted for by a filler (dummy) record so all space in a block is accounted for by its records. These records are illustrated in the following layout of a saveset block. The gray areas represent the variable-length data.



A saveset can span multiple tapes. Likewise, a tape can contain multiple savesets. Savesets on a tape are delimited by file marks, which are written when the saveset is closed by vdump. The saveset consists of three regions:

1. First block - Used to determine the block size, source directory path, and other saveset attributes.
2. Directory blocks - Used to restore the fileset directories into a temporary file. This enables browsing of the saveset and supports interactive/selective restoring of files.
3. File attributes and data blocks - Contains the directory and file attribute records and the actual file data records.

13.3.3 Vrestore Basic Design

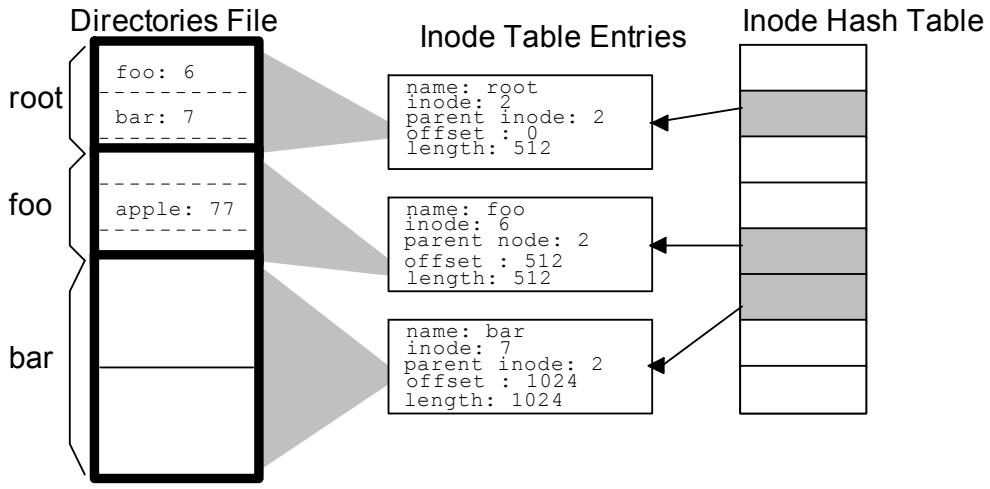
`Vrestore` allows the user to select specific files and directories to be restored. This can be done through an interactive shell or by using the `-x` option on the command line. To support this feature, `vrestore` uses the Directories section of the saveset and an in-memory inode table. Since POSIX defines an **inode** as the unique identifier for a file, we use the term `inode` in `vrestore` rather than the AdvFS tag number. This is done because `vrestore` is a file system-independent program.

`Vrestore` constructs a temporary directory structure of the original fileset that `vdump` backed up. This structure is created from the Directories section of the saveset and it is put in a file (the

Directories File) in the current working directory. The file is unlinked immediately after being created so that it will be cleaned up automatically after the `vrestore` process is done.

As the Directories File is being created, `vrestore` also creates an inode table. The inode table is a hash table (hashed by inode) with chaining for multiple entries that hash to the same slot. Each time a directory is restored into the Directories File, an entry for that directory is added to the inode table. The inode entry is used to describe the directory in the Directories File; it contains information like the directory name, its inode, its parent directory's inode, the byte offset to the start of the directory in the Directories File, and the length of the directory.

The following figure shows a sample Directories File and inode table for a saveset that consists of two subdirectories (foo and bar):



This structure allows `vrestore` to locate any file in these directories. For example, to find "./foo/apple" the following steps are done:

1. "`./`" is equivalent to the root directory whose inode (2) is stored in a global variable.
2. Lookup inode 2 in the inode table and get the location and length of the directory for inode 2 (root).
3. Search the root directory for "foo". The directory entry for "`foo`" will show that it has inode 6.
4. Lookup inode 6 in the inode table and get the location and length of directory "`foo`" (inode 6).
5. Search the "foo" directory for "`apple`". Its directory entry indicates that it has inode 77.

The inode table also allows `vrestore` to construct directory path names given just the last directory's inode number. This is important because `vdump` writes only the file name and its parent directory's inode number into the saveset (in the file attributes record). For example, to

generate the full path name for "apple" and parent inode 6, vrestore will lookup inode 6 in the inode table and find the name "foo" and parent inode 2; so now it has the path "foo/apple". Then it looks up inode 2 and finds it is the root so the full path name is "./foo/apple".

The above discussion only examines directories in the inode table. Vrestore also uses the inode table for individual files that are marked for restoration or marked as "don't restore;" that is, the files are to be examined and not restored. This process is used for the 'add' command in the interactive mode (-i option) and for the -x option, which allows the user to specify which files are to be restored. For example, if the user specifies vrestore -x ./foo/apple, vrestore would use the method described above to find the file "apple." It then adds an entry in the inode table for file "apple" with inode 77 and marks that entry as "restore me." If a directory is selected (like "./foo"), then since the entry already exists, vrestore just finds the entry for "foo" and marks it as "restore me." When a directory is marked "restore me," all files and directories beneath that directory are restored (unless explicitly marked as "don't restore me").

13.3.4 rvdump/rvrestore

The rvdump and rvrestore commands perform backup of a fileset to a storage device connected to a remote machine. For this to happen, the user/root must have an entry in the $HOME/.rhosts file on the remote machine.

Rvdump shares most of the functionality of vdump except when communicating to the remote system to which the storage device is connected. Rvdump uses a special variety of rcmd (a remote command function), called rcmd_af(), to communicate to the remote machine. Rcmd_af() is capable of handling both IPV6 and IPV4 ports.

Rvdump and rvrestore use rcmd to communicate to a rmt server process running on the remote machine. First, rmt is invoked on the remote system using rcmd_af(), which returns a socket file descriptor. This socket is used as an interprocess communication channel to communicate to the rmt server process running on the remote host. Access permission checks are done before the communication line is setup. Commands such as backup device open, read, write, close, and ioctl are executed on the remote host by the rmt process running there. The rmt process accepts commands and responds to the requests in a particular format. Rvdump and rvrestore issue commands to the rmt process and interpret the responses from the remote rmt accordingly. See the rvrestore(8) and rmt(8) pages for details.

13.4 Fixfdmn

Fixfdmn is a new utility capable of fixing a number of common corruptions in on-disk metadata, and is intended to be used if tools such as verify detect on-disk corruption, or if a domain cannot be mounted without a domain panic. Previously, if a domain became unmountable due to corruption in metadata, the only way to fix the domain was either to run salvage or to restore it from backup. Unlike the verify utility, fixfdmn does not need to mount the filesets in the domain, so it can fix classes of errors that would cause a fileset to be unmountable.

Fixfdmn scans on-disk metadata looking for corruption and attempts to correct such problems when located, if there is enough viable data to allow the on-disk metadata to be corrected. If not enough viable metadata is available, then fixfdmn attempts to bypass the corruption by deleting the corrupted metadata and related files. It is not be possible to fix all forms of corruption, and in those cases running salvage or restoring from backup tape is required.

Fixfdmn keeps track of all changes made to the domain, and gives the user an option to undo those changes.

The AdvFS domain locks are used to prevent other AdvFS utilities from accessing the domain at the same time fixfdmn is running. Fixfdmn reads and writes to the raw disks that comprise the domain; it does not access the domain through AdvFS. Fixfdmn will not run on a domain if there are any mounted filesets. It can be run in either multiuser or single user mode.

When fixfdmn is run for a given domain, the block devices specified for the volumes in /etc/fdmns/<specified domain> are opened. When booting from the CDROM to repair a domain, /etc/fdmns and the entries under it for the domain must be created by the user. If the user specifies the -a flag on the command line, fixfdmn attempts to activate the domain after the fix-up to verify that it can be mounted,. Although the domain may be activated, that doesn't guarantee all corruptions have been eliminated.

Fixfdmn does not process the domain's transaction LOG, but the LOG pages are rewritten to give the appearance of an empty LOG. Since completed transactions are not applied to the metadata and incomplete transactions are not 'undone' in the metatdata, fixfdmn must correct any inconsistencies that are caused by not processing the LOG.

The on-disk metadata is used to determine what corruptions exist in the domain. Only metadata will be repaired, as there is no way to check and repair user data. Fixfdmn is primarily concerned with fixing problems that have a limited scope. When a large portion of the domain is corrupted, there is very little fixfdmn can do, so it recommends restoring data from tape or running salvage. If a volume is missing from the domain, fixfdmn will not work, and the user needs to run salvage or restore data from backup.

If the metadata for a specific file can not be recovered, the file may be truncated or deleted, depending on the situation. As much of the file as possible will be saved.

Every page changed is saved to an undo file, so that fixfdmn can restore the domain to its original state upon user request. If the file system containing the undo files runs out of space during the fixfdmn run, the user is prompted how to proceed. The user has the option to continue without the undo files, to make more space available on the file system containing the undo files and to then continue, or to exit.

All corruptions encountered and whether or not they were fixed are reported.

The following paragraphs describe the sequence of steps that fixfdmn uses to process a domain.

Fixfdmn first verifies that the disk(s) specified actually contain an AdvFS domain. This is done by checking if the AdvFS 'Magic Number', which is located on each partition, is correct. If this value is incorrect, additional checks will be performed on the disks to determine if the magic number is corrupt, or if this is not an AdvFS domain. If it is not an AdvFS domain, then fixfdmn exits, informing the user that either the domain is not AdvFS or the domain is so corrupt that they need to run salvage.

Fixfdmn then validates the RBMT (for V3 domains, BMT page 0). The RBMT contains information on where the other important AdvFS on disk structures are stored, as well as general information on how the domain is put together. The RBMT check is applied to each volume in the domain. As part of this check, fixfdmn verifies that each volume's metadata agrees that the same number of volumes are contained in the domain. If this number is different from the number of volumes found in /etc/fdmns, fixfdmn will inform the user that they either have a missing volume that they need to locate (or optionally run salvage), or that one (or more) of the volumes needs to be removed from /etc/fdmns.

If the RBMT is missing or extremely corrupt, fixfdmn cannot continue and issues a message telling the user to run salvage.

Once the RBMT has been checked, fixfdmn starts checking each volume. This check is to make sure that the volumes are in sync with one another. This stage requires fixfdmn to perform multiple checks of metadata stored in each volume's RBMT. Fixfdmn checks that all the volumes have the same domain ids, mount ids, and domain version. If any volume's data differs from what is expected, fixfdmn asks the user to verify that the correct volumes are being used. If the user informs fixfdmn that all volumes are correct, then it selects the most common values between the multiple volumes and assigns this value to those volumes that are incorrect.

The final check in this stage is to verify that each volume's metadata agrees on the state the volumes are in, such as mounted or unmounted. If the volumes are inconsistent, the tool selects a value which makes sense to all volumes, and assigns it to them.

Fixfdmn then collects extent information on the BMT, SBM, log file, and root tag file. These extents are stored in a page-to-LBN array which allows fixfdmn to quickly locate any page in these files.

At this point fixfdmn builds the in-memory data structures which it needs to continue. The first structure built is an in-memory version of the SBM. The SBM allows the file system to know which pages (and blocks) have been allocated to metadata and to user data on each volume. Fixfdmn updates the in-memory version of this structure as it moves through the metadata. Near the end of execution of fixfdmn, the in-memory version will be compared to the version on disk.

Now that the volumes and high level domain information have been validated, the tool checks and clears the domain's log. Fixfdmn does not process the log records and apply the changes to the domain. This is because the tool would need to duplicate a large portion of kernel code, and at the time of implementation this code was not in a form which would be easily usable by

fixfdmn. Also, since the log might contain records which could overwrite pages which fixfdmn has already changed, the log is simply cleared.

After the log has been cleared, fixfdmn checks all the records stored in the BMT in two stages: a page by page check, and a detailed check which follows mcell chains for each file. The first check verifies that data falls within reasonable boundaries, such as confirming that extents do not extend past the end of the volume and that extents do not overlap. Some of these checks are done by checking the in-memory SBM. When each extent is processed, the tool updates the in-memory SBM with the pages that are known to be used.

Fixfdmn then sequentially steps through the extent maps in the BMT, examining each page in the extent. A BMT page's numerical position can be verified because it contains its BMT sequence number. Since these numbers are sequential, fixfdmn knows if the BMT is missing pages, at which point fixfdmn recreates pages as needed. When fixfdmn finds a page so badly corrupted that it can not correct the page, it reinitializes the page, puts the page in the correct position in the BMT, and finally puts the page on the free list. After this, the tool finds all references to this BMT page and terminates any mcell chains linked to this (now free) page.

On each BMT page there are 28 mcells which fixfdmn verifies while it is examining all the pages in the BMT. However, some mcell checks can only be done by following mcell chains, and these checks are done at a later time. Each mcell has a header which contains the file's tag and an mcell chain pointer which may point to another mcell for the same file. After the header, the mcell also has one or more records which contain data about the file it is describing.

After the mcell header is checked, each mcell's records are verified. There are around 25 different types of mcell records to be checked. Each mcell record has two sections: the record header (which has the same format for all records), and the corresponding record metadata. Fixfdmn validates that the record header and metadata are both of the same type. The tool also checks and fixes the metadata for each of the mcell records.

As part of the mcell verification algorithm, fixfdmn maintains a skiplist of in-use mcells in the domain. As each mcell is verified, as described above, it is marked either as in use and containing useful information, or it is marked not in use. Fixfdmn adds an mcell node into the skiplist for each used mcell it finds. This skiplist is used later to verify the contents of the Root Tag File.

After the BMT pass is complete, fixfdmn clears the deferred delete list. The deferred delete list (DDL) is used for both deleting and truncating files. The DDL stores the mcell of the extents we are truncating, or the entire mcell chain of a file we are deleting. The DDL mcells for extents being deleted are preserved until the truncation or deletion operation is complete. If a file is being deleted, it should either: not be in the tags file or be marked as being deleted. If a file is being truncated, it will be in the tag file and not marked as deleted.

The next step is to collect a list of all the filesets in the domain. This information is stored in the Root Tag File. The Root Tag File is a collection of pages, where each page contains a header, followed by an array of up to 1022 bsTMap structures that describe the location of the Tag File for each fileset in the domain.

The Root Tag File header contains the number of used, free, and dead primary mcells contained in the array. This number is verified against the actual number of used, free and dead primary mcells in the array. Each primary mcell in the array is then verified against the mcell skiplist to ensure that it is a valid mcell. If the mcell is not valid or the mcell is missing, then a page by page search of the BMT is performed later to locate the fileset's primary mcell.

Now that `fixfdmn` has a list of all the filesets, it creates a skiplist structure for each fileset which will contain information about all the tags in that fileset. The information needed to create these skiplist structures is stored in the fileset's tag file.

NOTE: Some of the BMT mcell and record checks described above are not done until the tag files have been checked. So, another walk through the BMT is done here.

One of the checks `fixfdmn` performs is looking for loops in file mcell chains. As each mcell should only be pointed at by one other mcell, the tool can use this information to find problem mcell chains. This is done by using a status bit in each record in the mcell skiplist. This status bit will let `fixfdmn` know if some other mcell already points to this mcell. Now as `fixfdmn` goes through the mcells on each BMT page, it first checks to see if the chain is pointing to a valid mcell. If the mcell is valid, then `fixfdmn` will check that the status bit in the mcell skiplist has not been set. If the bit is not set, it will be set. If the tool finds the status bit already set, it knows it has a problematic mcell chain, and flags it for further testing.

The tool then checks that all pages with free mcells are on the free mcell list. If any pages with free mcells are found that are not on this list, they are added to the list. If there are any pages on the free list that do not contain free mcells, they are removed from the list. Each page in the free mcell list contains a header storing the current number of free mcells on that page. This number is verified with the actual number of free mcells on that page and corrected if necessary.

Although the verification of the BMT chains and the verification of the frag file chains are done simultaneously, we are describing them separately for simplicity and clarity. Tag 1 represents the frag file, so `fixfdmn` steps through the extents describing tag 1 in order to locate all the frag file data.

The frag file is logically separated into frag groups, one for each frag size (1k, 2k, etc.). Each frag group has a header describing how the group is formatted. If the frag header is corrupt, then `fixfdmn` repairs it. `Fixfdmn` also verifies that there are no loops in the free group list or in the free frag list for each group. Later, each tag is checked that it points to a valid frag, if it has one.

The next stage in `fixfdmn` execution involves checking each tag against a number of different error conditions. To do this, `fixfdmn` steps through each tag's mcell chain, and checks that the data in the chain is consistent. `Fixfdmn` has already checked for possible mcell loops so it does not need to check for them again during this stage. The following lists some error situations that are checked and fixed:

- Tags with missing extents
- Tags which have overlapping extents
- Tags which have a frag overlapping an extent

- Tags which have mcells which belong to a different tag
- Verify that tags point to valid frags
- Verify that tags with symlinks are valid
- Verify that tags with property lists are correct
- Verify tags with trashcans are correct

After that, the directories are checked to ensure that all tags are in at least one directory and that no directory is in more than one other directory. `Fixfdmn` also ensures that there are no corruptions in the directories themselves, and that the directory indexes match the directories. If a directory index is corrupt, or does not match its full directory, that index is deleted (the kernel can rebuild the index).

Then the user and group quota files are checked to ensure that the file size of the quota file matches the last extent of actual storage for the file. The data in the quota files is not checked, as it can not prevent mounting a fileset. Quota data validation can be done by running the quotacheck utility.

One of the last checks that `fixfdmn` does is compare the in-memory SBM to the on-disk SBM. By this point we have parsed all the extents in the domain and have marked them in the in-memory SBM. Now `fixfdmn` can compare the checksum of the in-memory SBM against the on-disk SBM, and if the numbers are different then the in-memory version will be written to disk.

After all the checks have been made, `fixfdmn` writes out the undo files. Only after the undo files are closed and written to disk are the changes to the domain volumes written out.

13.5 Verify

The `verify` utility mounts all the filesets in a domain at temporary mount points to process them and unmounts them when finished. It checks on-disk structures such as the BMT, the storage bitmaps, the tag directory, and the frag file for each fileset. It verifies that the directory structure is correct, that all directory entries reference a valid file, and that all files have a directory entry. If an inconsistency is found, it attempts to fix the problem in-memory so that the mcell is consistent.

`Verify` checks the storage bitmap for double allocations and missing storage. It checks that all mcells in use belong to a file and that all files have all of their mcells.

The consistency of free lists for mcells and tag directories is checked. `Verify` checks that the mcells pointed to by tags in the tag directory are indeed the mcells associated with that tag number.

For each fileset in the specified file domain, `verify` checks the frag file headers for consistency. For each file that has a fragment, the frag file is checked to ensure that the frag is marked as in use.

`Verify` creates a tag hash table to check that all in-use mcells are included in a file's metadata mcell chain. It checks every mcell on every volume in the domain and if an mcell is marked as

in-use or marked as being bad, it checks that the mcell position field is non-zero. A zero value in the position field indicates that the mcell is not linked into an mcell chain for any file.

Each fileset's tag directory is then verified. The tag for each fileset is found in the root tag directory. Each in-memory tag is then verified to be valid by finding the tag in the on-disk tag directory for the corresponding fileset.

`Verify` checks the consistency of directories, files, and the BMT for each fileset. Each file's frag is also checked for consistency. `Verify` does not attempt any repairs on clone filesets, since they are read-only.

13.6 Salvage

13.6.1 Overview

One problem with AdvFS file systems is there is no way to recover file data when a domain becomes unmountable due to an error in metadata, accidental removal of a volume, or other errors causing a domain panic. `Salvage` is able to recover file data from corrupt domains or domains with volumes missing. Since domains tend to be large, `salvage` works primarily on the fileset level.

`Salvage` uses on-disk structures to obtain file data. It uses as much viable data as is available in the on-disk structures in the case where corruption has occurred. It recovers as much file data as it can find. However, it may not be possible to maintain directory structures with some forms of corruption. `Salvage` is not able to overcome all possible disk corruptions. If no corruption exists in the on-disk structures, recovery of all files in the fileset will occur. Although `salvage` may recover all pages of a file, it cannot guarantee that the file data is correct. Because it is reading the raw device, `salvage` may recover out-of-date data from the device if on-disk structures have not been updated from the buffer cache.

`Salvage` opens only raw block devices, and only issues read requests. It uses the AdvFS domain locks to prevent other AdvFS utilities from accessing the domain, and is able to run in single user mode

`Salvage` reports any problems it encounters when trying to recover files. It attempts to find all disk blocks associated with a file. However, due to corrupt metadata, it may recover only a portion of a file. This may cause a file to be truncated or to have missing pages in the middle of the file. The file and the page ranges which were recovered are listed in the log file.

If it is not possible to recover the file names, names are created for the files incorporating their tag numbers, and the files will be put in the fileset "`lost and found`" directory under the recovery directory. If it is not possible to recover the file attributes, the `uid` will be set to `root`, the `gid` will be set to `system`, and the protection will be set to "read by owner".

`Salvage` does not recover clone filesets. The filesets from which the clones were made will be recovered.

`Salvage` is capable of processing AdvFS filesets on LSM volumes.

`Salvage` is intended to be used as the method of last resort for restoring files from corrupted domains. It does not attempt to fix any on-disk structures or to make the domain mountable. Therefore, it is recommended that `fixfdmn` is run before `salvage` is run.

13.6.2 Actions Taken During Recovery Processing

`Salvage` has several distinct steps it performs in order to recover data from a corrupted domain:

1. Get options from the command line and initialize data structures
2. Build the directory tree and fill in the tag pointer array
3. Fill in filenames as found in the directory data and validate the directory tree
4. Walk the directory tree and restore the files

By using this design, a tradeoff is made to use memory rather than disk space to store data structures. The primary reason is to allow for an easy transition to tape output.

In general, `salvage` tries to overcome errors encountered due to disk corruption. `Salvage` also attempts to keep processing even after system call errors, if at all possible. In these cases, `salvage` most likely returns a status of partial recovery. However, in the case of an error that prevents continuation, such as memory exhaustion, it exits. `Salvage` also exits on errors occurring before the start of disk processing (i.e., invalid command line options, error opening the log file, etc.).

Each volume is checked for a valid AdvFS magic number or for valid BMT extent records located in the RBMT (or BMT page 0 in DVN 3 domains).

Then the BMT LBN (Logical Block Number) array is built for each volume. This array maps each BMT page to a block on the disk. This allows an easy lookup of the on-disk LBN for each BMT page. In order to determine the size of this array, the BMT extent chain is followed to find the last extent. The number of pages found while following the extent chain is used as the size of the BMT LBN array.

In DVN 4, the BMT extents are found in the RBMT file. The RBMT file can be several pages long. Before we can find all the BMT extent mcells in the RBMT file we need to find all the RBMT pages. Each RBMT page reserves mcell 27 to contain the extent of the next RBMT page. In DVN 4, this requires following the RBMT next pointer to find all possible RBMT pages, but in DVN 3 all of the extents can be obtained by reading only BMT page 0.

Next we need to confirm that the volumes are in the same domain and obtain the real volume index for each volume. This is done by following a next pointer from mcell 0 on page 0 of the RBMT (DVN4) or BMT (DVN3) and checking the vdi for each volume. In DVN 4, the pointer usually points to mcell 6 of the RBMT (in DVN 3, the pointer is to mcell 4 of BMT page 0). The volume information is then resorted by volume index to make it easy to access volume information.

Mcell 2 of the RBMT (BMT0) is checked on each volume for the root tag file. This file provides the locations for the fileset tag files. A fileset information entry is created in-memory for each fileset and linked onto a fileset list for the domain.

Each fileset tag file is searched to find the number of tags in use, which is used to determine the size of the tag pointer array to allocate. The first 2 nodes of the file directory tree are created for the top level directory (`<fileset>`) and the lost and found directory (`<fileset>.lost+found`).

The primary mcell for the frag file (tag 1) is read and the extents for the frag file are obtained from the extents chain. The frag file LBN array is then built and filled in using this information.

Building the Directory Tree and Filling in the Tag Pointer Array

Metadata is read from the on-disk structures, and the in-memory representation of this data is constructed by `salvage`. The collection of the metadata may need to be done in multiple passes, where each pass consists of a different strategy to access the on-disk data. Based on what we have seen of fileset corruption, most of the fileset directory tree should be built in the first pass. Salvage only needs multiple passes if it finds data corruption that it can not work around.

Pass 1 - This is the main pass and it uses the fileset tag file to access the BMT to get mcell information for all files. The pointers to next and chain mcells in the BMT are followed. A fileset directory tree node is created for each file and the fileset tag pointer array entry is filled in. If the parent entry has not been created yet, the node is linked to the lost and found node and the parent tag number is stored in the node (it is possible to find a file before we find the directory which contains it).

In addition, there are 3 special cases to pass 1. These are performed if the user specifies that only a portion of the fileset should be recovered on the command line. There is an option of selecting recovery by tag, path and/or date.

> **Recover by Tag** - After creating the top node of the tree, if a tag was specified, the metadata for this tag is read and the entry in the fileset directory tree is created. One of the fields in the metadata is the parent tag. Using the current tag's parent, `salvage` reads the metadata of the parent and creates its entry in the tree. `Salvage` continues to follow the parent tags and create nodes until it reaches the top of the directory tree. To save memory, we now want to prune the tree so that we only recover files which are in the same path. At each level in the tree, the directory data for the tag's parent is read and all the tags in this directory are set to “ignore” except for the current tag.
>
> **Recover by Path** - After creating the top node of the tree, if a path was specified, the directory data for the fileset’s root directory (tag 2) is read. Each entry in the directory is compared to the first portion of the path specified. For each filename that does not match our current path directory name, the tag is set to “ignore”. This prunes the tree at the top level to save memory when recovering only a path and leaves exactly one node which matches the path. We keep following the path in this manner until we have found all tags which create the full pathname we are recovering.

Recover by Date - This case is processed when stepping through the fileset tag directory. If a date was specified on the command line and the file for a tag is not a directory and has not been changed since the date specified, the tag is marked as "ignore".

Recover all tags not marked ignore - This step goes through the fileset tag directory for each tag. If the tag is not already marked as "ignore" (i.e. from recover by tag, path, or date), and its parent is not marked as "ignore", then fill in the fileset tag pointer array entry and create a tree node for it. If a tag is not marked as "ignore", but its parent is marked "ignore", then mark the tag as "ignore" in the tag array.

Relink lost and found - As it is possible to recover a child before a parent, at the end of pass 1, entries may exist under `lost+found` which actually do have a parent in the fileset directory tree. For each child under the `lost+found` node, if its parent exists and is not marked as "ignore", relink the node under its parent. If the parent is marked "ignore", free this node and everything linked under it.

Pass 2 - This pass is performed when we can't find the root tag file, any fileset tag files, or have partially recovered files. This pass searches all the mcells on RBMT pages (BMT page 0 in DVN 3) looking for mcells that describe the BMT file.

Next a sequential search of the BMT is performed, page by page, mcell by mcell. When the tool finds an mcell which belongs on the free list, this mcell is ignored. If the tool can tell that an mcell belongs to the deferred delete list, it is ignored. In the case that the tool can not tell if the mcell is on the deferred delete list and the file being restored contains more extents than expected, a warning is printed in the log.

None of the pointers to next or chain mcells are followed in this pass. The mcells for files for which all extents have been found can be ignored. If Pass 1 was not successful and if the root tag file or a fileset tag file is found in this pass, then we can use parts of Pass 1 to process them.

If `salvage` finds data in later passes that conflict with data found in earlier passes, the tool assumes that the data found in the prior passes is correct. This is done because the data collected in earlier passes is more closely aligned with the way AdvFS would collect the data under normal operating conditions. The later passes have a greater possibility of finding invalid data, and we have no way to detect this.

Pass 3 - This pass is performed only if a special flag is used on the command line. This pass is useful when a `mkfdmn` has been performed by accident on top of a valid domain, or if the RBMT (BMT page 0) can not be found. A sequential search of the disk is performed, page by page, looking for pattern match for BMT pages.

Filename Recovery and Directory Tree Validation

After the tag pointer array and fileset directory tree data structures are built, the tag array is scanned, and the actual filenames for the tag structures which have been found are retrieved from the directory files and inserted into their respective positions in the fileset directory tree.

In the event that a conditional salvage operation is specified which is based on a specific date range, the directory tree is first scanned to eliminate superfluous empty directories. The tree is

pruned of these empty directories and the status of these files is marked as “ignore” in the tag pointer array.

To retrieve the filename data, the tag pointer array is scanned in tag order. If a tag's file type indicates that the tag is a directory, and the status of the tag in the tag pointer array indicates the directory should not be ignored, then the following steps are taken:

> **Read the data:** Initialize the `fsDirData` structure. This structure includes a pointer to a data buffer used to contain the raw data from the directory file and fields to track the current relative position within the data buffer as it is sequentially parsed. The directory file is then read from disk according to the tag's extent map, one page at a time, and stored in the buffer within the `fsDirData` structure.
>
> **Scan the buffer:** The directory entries in each page are sequentially parsed. Each entry's state is checked, and if valid, the tag number and filename data are extracted.
>
> **Find the node:** The file's tag entry is looked up in the fileset directory tree, and the filename is inserted into the name field of the file's node in the tree.
>
> **Check for hard links:** The relationship between the tag's parent tag (known from the tag's metadata) and the tag's position in the tree (i.e., its "positional parent") is checked. Anomalies indicate the tag has multiple hard links, and these conditions are resolved by creating additional nodes in the tree as required.
>
> **Validate tree data:** All tags are scanned, and the `linksFound` attribute and the `numLinks` attribute are compared. This check corrects for a particular condition which occurs when multiple hard links for a file exist, and the original link was deleted. Otherwise, anomalies are noted and logged. In addition, all tags are scanned to verify that they have been properly named. If tags exist which do not have names (possibly due to missing data in the directory files), special names based on the file's tag number are generated and inserted. In addition, if directory entries are found for which there are no valid tag entries, "empty shell" nodes for these entries are created in the fileset directory tree. These nodes serve as markers so that their existence can be logged during file recovery.

After the fileset directory tree has been checked and made consistent, a final check for extent allocation is performed using the SBM. The bits in the SBM are checked against the disk blocks which comprise each page of each extent in each file. If all of the bits are set to one for the corresponding disk blocks, then they are allocated to a file (mostly likely the file we are checking, but there is no guarantee). If all of the bits are set to zero, then the corresponding disk blocks are not considered to be allocated to a file. If this is true for all pages in the file, then most likely the file was deleted, but the metadata had not been overwritten. This case could also indicate corruption in the SBM as would the case when some bits are one and some bits are zero for a file. In all cases of inconsistency, an error message is displayed.

Restoring Files in the Directory Tree

The fileset directory tree is processed from the top node down (depth first with directories being processed before children) and the files are recovered based on their file type and status. Any

files that are marked as “ignore” or “dead” will be skipped as well as files that are only partially recovered (this occurs only if the `-x` option was specified).

By the time this point is reached in the code, the fileset directory tree will have been pruned to eliminate files not changed since the specified date and/or files not within a specified path. In addition, the extents for each file will have been sorted in file page order.

Before writing out the file data, a check is made to see if the file already exists, and if so, whether or not it can be overwritten. This is done in anticipation of future check-pointing within `salvage` where files may be recovered in stages. The suffix “`.partial`” will be added to the filename if the command line option to do so was chosen.

The following file types are handled: directories, regular files, symbolic links, hard links, fifos, and device special files. Where appropriate, file attributes and property lists are restored.

For regular files, extents are read from the disk page by page and written to the restored file. This continues extent by extent until the entire file is written. Where holes occur in sparse files, `lseek` is used to skip over them in the restored file. Depending on command line options, messages may be written into the log file.

After all the files under the top level directory are processed, the files in the `<fileset>.lost+found` directory are restored in the same manner.

13.7 Vfast

The `vfast` utility automates the tasks that are manually performed using the `balance` and `defragment` utilities. This tool performs file defragmentation, executes volume capacity balancing, and migrates user files to eliminate device I/O bottlenecks. Administrators are freed from mundane tasks and 24x7 operations aren't slowed down by periodic utility operations. `Vfast` was added to Tru64 version 5.1B, and is optionally started as a background daemon when the system is booted. It automatically runs during periods of low system demand to minimize any performance impact to no normal filesystem operations.

The primary goal of `vfast` is to provide a file system that requires no maintenance beyond adding and subtracting storage. `Vfast` performs the following maintenance duties without interaction from system administrators:

- defragmentation and free space consolidation within the domain
- balancing of the free space over domain volumes
- monitoring to identify "hot" files, then distributing them across AdvFS volumes to improve file-access performance

`Vfast` does not require any ramp-up time to start processing files, although as time goes on `vfast` becomes better at predicting which files are used more often and therefore should be fixed sooner. During normal operations, `vfast` avoids any significant analysis before performing useful defragmentation, as is the case with the standard `defragment` utility.

Some utilities, such as `umount`, `rmvol`, and `rmfset`, suspend `vfast` operations temporarily while they run. When the utilities finish, `vfast` is returned to its prior state. Also, because

vfast defragments and balances domains, the traditional AdvFS defragment and balance utilities cannot be used if vfast is activated with the -o defragment, -o balance, or -o topIObalance operations enabled. The Tru64 AdvFS Administration Guide has a very complete section on using vfast.

13.7.1 Balancing free space across volumes

Vfast, balance, and defragment utilities perform similar work because they move extents in order to provide better file-access times, but vfast automates this procedure so that it happens automatically, requiring no interaction from system administrators. Balancing free space among volumes improves subsequent write performance so that newly created files are distributed evenly over the drives. Newly created files have a greater potential for becoming the files with the highest sustained IO count, so they should be evenly distributed over the disk volumes at creation time. Volume capacities are maintained within 10% of each other by vfast.

13.7.2 Defragmentation of a volume

Defragmenting, whether done via vfast or defragment, has two primary functions:

1. Reduce the number of file extents.
2. Reduce the number of free-space fragments and create larger free-space segments.

When a file consists of many discontiguous file extents, the file is considered to be fragmented. File fragmentation reduces read/write performance because more I/O operations are required to access the file. Likewise, when a volume's free space is scattered over many discontiguous extents, the volume free space is fragmented, increasing the chance that new files will have discontiguous file extents.

The vfast utility attempts to reduce the number of file extents in a domain by making files more contiguous. Vfast also attempts to make the free space on a disk more contiguous, resulting in less fragmented file allocations in the future.

Vfast runs most efficiently on volumes that have more than 20% free space. The more free space in the domain, the more efficient defragmenting and I/O balancing becomes.

13.7.3 Frequently Accessed File I/O Distribution

Vfast is a tool to automatically distribute the disk I/O across the available disks. This helps to prevent a single disk from becoming an I/O bottleneck and decreasing performance. Vfast can also determine the files with the highest amount of disk I/O in the domain and distribute them evenly across the disk volumes.

Each AdvFS file domain has a transaction log that tracks activity for all filesets in the domain and ensures AdvFS metadata consistency if a crash occurs. The domain's transaction log may become a bottleneck if the log resides on a congested disk or bus, or if the domain contains many filesets.

To prevent the log from becoming a bottleneck, vfast includes the log, as well as the other AdvFS reserved files, in its I/O distribution algorithm. The transaction log's special I/O

characteristics are considered when determining where to place the frequently accessed normal files.

Statistics gathered by the AdvFS kernel (such as buffer cache misses, file writes, and file access times) are used to produce file usage profiles that are then used to intelligently distribute the I/O load over domain volumes.

The performance benefit of I/O distribution depends on the number of disks in the domain, the location of the disks, and how applications perform I/O. Under ideal conditions, I/O performance improves linearly as the number of disks in a domain increases. All files redistributed by `vfast` are also defragmented during the placement operation.

In order to determine which files have the highest disk I/O, two parameters are used. The first is an I/O counter in the file's access structure. This counter is incremented each time a write or read from disk is performed for this file. This counter determines which files have the most potential for consistent I/O. The use of this counter does not significantly affect overall I/O performance for AdvFS.

The second parameter is a field used to store the last file I/O time. This parameter is used by `vfast` to determine which frequently-accessed files have been accessed recently. Without this parameter, the I/O counts could become stale after an initial burst of file activity and `vfast` would not detect this condition. As file access times grow older, files are less likely to be considered frequently accessed.

The contents of the hot file list can be displayed with the command:

```
vfast -l hotfiles dmnName_x
```

The `vfast topIObalance` option distributes hot file I/O over volumes to perform I/O load balancing. If `topIObalance` is used on a domain, and then disabled, the fragmented files are defragmented without regard for previous file placement. This means files will be defragmented without regard for I/O distribution across volumes and may be moved from their present location on disk.

`Vfast` will create new fragment lists over time as files are accessed. This list is needed to track file extent segments in real-time so that `vfast` does not have to continuously scan the domain. The time required to create this list will vary according to the number of files in the domain. This list creation allows normal file system operations because `vfast` is still subject to the `-o percent_ios_when_busy=percent` option. The list for high disk I/O files is created after an initial ramp up time without affecting current operations.

13.7.4 Dealing with cloned and striped files

Within a clone file, the non-holey ranges of pages will be defragmented when possible.

Clone files are distributed across volumes according to the number of times that their extents are accessed. Clone file hits are counted according to the following rules:

1. If the original of a clone has not been modified, the original receives credit for file hits on the clone.

2. If some changes have been made to the original of a clone, both the clone and the original receive credit for clone file hits.

3. If the original of a clone has been modified so that no more pages need to be cowed, only the clone receives credit for clone file hits.

Striped files are not defragmented by `vfast`.

When the `rmvol` command is used to remove a volume from a domain, `vfast` is aware of the change and no longer moves files to the device being removed.

13.8 Freeze/Thaw

AdvFS's multi-volume domain functionality can cause problems when utilizing some of the features inherent in today's hardware storage products. For example, hardware snapshotting (similar to AdvFS's cloning) is in widespread use by customers. If a multi-volume AdvFS domain made up of multiple hardware RAID LUNs is snapped, metadata inconsistency in the snapshot is likely. This metadata inconsistency stems from not being able to snap all of the LUNs in the domain simultaneously. Since there is a time differential between snapping each LUN, the view of the metadata (which is spread across all of the LUNs) becomes inconsistent in the snapshot.

To allow coherent hardware snapshots, AdvFS provides a method to freeze (or quiesce), and later thaw, its metadata domain-wide. This means that even though all LUNs are not simultaneously snapped, there is no activity in the domain, and the metadata remains in a consistent state so long as the domain remains frozen.

A distinction between two types of freeze is necessary. The term **meta-freeze** describes a freeze of metadata only, and the term **full-freeze** is used for a complete filesystem/domain-wide freeze.

For meta-freeze, only metadata is frozen and flushed to disk (possibly with whatever user data is currently in memory). Subsequent operations on the filesystem are allowed as long as no metadata changes are needed. Since all metadata updates are under transaction control, all operations that require metadata updates are blocked on the start of the transaction.

For full-freeze, all filesystem activity is stopped and current data in memory is flushed to disk.

13.8.1 Overview

For meta-freeze, the premise is to use an exclusive transaction in AdvFS to finish existing metadata changes and to block new transactions. Since all AdvFS metadata operations are done under transaction control, blocking new transactions effectively blocks other metadata changes. The `freezefs` API sends a message to a background thread which starts an exclusive transaction, flushes all existing metadata buffers, and then signals the API. The API then returns. The `thawfs` API sends a different message to the background thread which then completes (or aborts) the exclusive transaction.

13.8.2 freezefs and thawfs utilities

The `freezefs` and `thawfs` utilities provide metadata freeze/thaw in support of multi-volume hardware snapshots. The freeze is domain-wide, so all filesets in the domain are affected by the operation. These utilities are actually a single binary module whose names are

hard-linked together. The name by which it is invoked is parsed and determines whether the domain will be frozen or thawed. Operations on the domain which require transaction control (metadata updates) will block while the domain is frozen, I/O that would not cause AdvFS metadata to change is allowed to proceed. For freeze, any in-process metadata updates are allowed to finish, and new ones are blocked. For thaw, normal operation is resumed on the specified domain.

The normal sequence of events involving freeze/thaw is as follows: freeze the fileset, perform the hardware snapshot, and thaw the fileset. (Even though a mount point or fileset is specified to the utilities, all filesets in the domain are affected). A timeout parameter can be entered on the command line to specify the maximum time that the domain is to be frozen. If the domain is not explicitly thawed before the timeout expires, it will thaw on its own. A timeout value of zero specifies that the default timeout value should be used, and a negative timeout value specifies that no timeout is requested and the domain should stay frozen until the thaw command is given.

13.9 Advscan

In order to mount an AdvFS fileset, the domain that contains the fileset must be consistent. An AdvFS domain is consistent when the number of physical partitions or volumes with the correct domain ID are equal to both the domain volume count (which is a number stored in the domain) and the number of links to the partitions that are in the `/etc/fdmns` directory.

If you attempt to mount an inconsistent domain, a message similar to the following will appear on the console:

```
# Volume count mismatch for domain dmnz.
dmnz expects 2 volumes, /etc/fdmns/dmnz has 1 links.
```

The `advscan` utility attempts to fix domain inconsistencies such as these so that you can mount filesets in a domain. The `advscan` command locates AdvFS volumes (disk partitions or LSM volumes) that are in AdvFS domains. Given the AdvFS volumes, it can re-create or fix the `/etc/fdmns` directory of a named domain or LSM disk group. For example, if you have moved disks to a new system, moved disks around in a way that has changed device numbers, or lost track of where the AdvFS domains are, you can use this command to locate them.

Another use of `advscan` is to repair AdvFS domains when you have broken them. For example, if you mistakenly delete the `/etc/fdmns` directory, delete a domain directory in the `/etc/fdmns` directory, or delete links from a subdirectory under the `/etc/fdmns` directory, you can use `advscan` to fix the problem.

The `advscan` command accepts a list of disk device names and/or LSM disk group names and searches all the disk partitions to determine which partitions are part of an AdvFS domain.

You can run `advscan` to automatically rebuild all or part of the `/etc/fdmns` directory, or you can rebuild it manually by supplying all the names of the volumes in a domain. Run `advscan` with the `-r` option set to re-create missing domains from the `/etc/fdmns` directory, missing links, or the entire `/etc/fdmns` directory.

To determine if a disk partition is part of an AdvFS domain, `advscan` performs the following functions:

- Reads the first two pages of a partition to determine if it is an AdvFS volume and to find the domain information.
- Reads the disk label to sort out overlapping partitions. The size of overlapping partitions are examined and compared to the domain information to determine which partitions are in the domain. These partitions are reported in the output.
- Reads the boot block to determine if the partition is AdvFS bootable.

The `advscan` command displays the date the domain was created, the on-disk structure version, and the last known or current state of the volume.

See `advscan(8)` for examples of `advscan` output and options available.

13.10 Vods Tools

The vods (View On-Disk Structure) tools allow you to look at the on-disk metadata in a domain. These tools also allow you to save on-disk metadata to a file and then to look at the metadata at a later time using these tools. This is extremely helpful when debugging a customer problem that may involve inconsistencies between on-disk and in-memory data. A customer can run a subset of these tools to save on-disk metadata structures into files, and then you can analyze the on-disk state of the domain by viewing the metadata structures stored in the files sent to you, using these tools. There is a vods tool for each AdvFS metadata type:

2. `nvbmtpg` views the volume BMT and RBMT pages.
3. `nvfragpg` views the fileset frag group headers.
4. `nvlogpg` views the domain LOG file.
5. `nvtagpg` views the root and fileset tag files.
6. `vfilepg` views, as hex, any file or block in the domain.
7. `vsbmpg` views the volume SBM pages.
8. `savemeta` is a script that saves the metadata of a domain.

The vods tools read the disk directly and never write to the disk. They do not operate through the file system, so they are suitable for looking at an inactive domain with no mounted filesets, and this method is recommended. The vods tools are able to look at metadata on active domains, but because they do not coordinate with the file system, inconsistent data may be provided by the tools as AdvFS changes metadata while the tool is being run or between runs. Also, AdvFS has access to cached dirty buffers; the vods tools do not. Running vods tools on an active domain may result in reports of errors where there are none.

When a domain is specified in the argument list for a vods tool, the tool will open the block device(s) linked from the `/etc/fdmns/<domain_name>/` directory. Because block devices can be opened by only one application and mounting a fileset also opens the block devices of the

domain, the vods tools normally fail to work on a domain with one or more mounted filesets. The `-r` option will cause these tools to open the character devices that correspond to block devices of the domain.

If the source argument is a device special file (a volume), you can specify whether the tool should use the block device or the character device. Specifying the block device protects the file system from changing the metadata during the time the vods tool is reading the metadata. The character device allows the vods tool to work even when the file system has opened the block device (i.e., when a fileset in the domain is mounted). Using an abbreviated device name resolves to the block device (e.g. `dsk5c` resolves to `/dev/disk/dsk5c`).

13.10.1 Argument order

The vods tools arguments are positionally sensitive. There are a few option style arguments (for example, -v) that precede all other arguments, then the remaining arguments sequentially refine where in the domain to look for metadata. For example, the `nvbmtpg` command requires the domain name first, then the volume id, then the page. Alternatively, some commands require the domain name first, then the fileset within the domain, then a file within the fileset.

Each additional argument either refines the search further (for example, specify `-f` to show the free list on the page) or tells what to search for next (specify `-c` to follow a chain of mcells).

The way you specify the location of the metadata depends upon the file you would like to view. For some metadata files, like the BMT, there is a file on each volume. So to view pages of the BMT (using `nvbmtpg`), you must specify the volume to indicate which BMT file to display. The volume can be specified either by a domain name and volume id pair, or simply by special device name. For example, on a two volume domain, called test_dmn, composed of `/dev/disk/dsk2c` (volume 1) and `/dev/disk/dsk8c` (volume 2), a volume could be specified in one of two ways:

- domain and volume pair (i.e., `test_dmn 2`)
- special device name (i.e., `/dev/disk/dsk8c` or `dsk8c`)

Some metadata files, like the log, occur only once in a domain. To view the log (using `nvlogpg`), you only have to specify the domain. The utility finds the volume that contains the log. However, if you already know what volume contains the log, you can specify the volume identification instead. In that case, the utility looks only on that volume to find the log.

All the vods tools can display any page (sixteen 512-byte blocks on a sixteen block boundary) on any disk, assuming you have permission to read the disk. In addition, the vods tools can display the entire metadata file. This can be used with `grep` or an editor to find a specific part of a metadata file. Most of the vods tools display a summary of their metadata file if no page is specified. For example, the command

```
nvbmtpg test_domain 1
```

displays information about the BMT on volume 1 in `test_domain`. The command

```
vsbmpg test_domain
```

displays general information about the SBM on each volume in `test_domain`.

Some metadata files (i.e., BMT, tag file) contain information that can be identified as belonging to one user file. The vods tools for these metadata files have arguments that allow the user to display only the parts of the metadata file that control or belong to the specified user file.

The vods tools can read an entire metadata file and save that data to a specified file. The tools can then process the saved file with most of the same capabilities as they have when working on a live system. This is useful when saving information from one system and analyzing it on another system.

13.10.2 Using the vods tools

The reference pages for the vods tools show many examples of how to use them. It is worthwhile to review these `man` pages to see of all the options available for viewing different types of metadata with each tool.

The vods tools can be used to manually follow the related metadata from one file to another. The following shows an example of how to look at saved metadata files from a domain:

Suppose you have saved metadata including saved directories from a domain, and you want to view the primary mcell of a file known by its domain, fileset and path.

1. Use `vfilepg` to view the saved directory that contains the file.
    a. `vfilepg saved_dir –a –f d`
    b. `-a` formats and displays the entire saved directory
    c. `-f d` formats the saved file as a directory
    d. Find the desired file in the directory and note its tag number.

2. Knowing the fileset and the tag number, use `nvtagpg` to find the primary mcell. There are 1022 tags on a tag page. Divide the tag number by 1022 to determine the tag page number.
    a. `nvtagpg saved_fileset page_#`
    b. Find the required tag and note the primary mcell (volume, page, mcell).

3. Use `nvbmtpg` to display the primary mcell. Use the volume from step 2 to select the saved BMT file (one per volume in the domain).
    a. `nvbmtpg saved_bmt page cell`
    b. This displays the metadata records in the file's primary mcell.

In an active domain (vs. saved files) you would not have to do these steps manually. The vods tools look at all the metadata to find the requested information for you. For example, to display the primary mcell, as above, of a file, known by domain, fileset and path within the fileset, type:

```
nvbmtpg domain fileset dir/file
```

The vods tools look at the BMT, tag file, and directory files to find and display the requested mcell.

Chapter 14: Interactions with Other Layers

14.1 VFS

One of the components of a process is the file table. When a process opens a file, an entry is made into this table and an index (the file descriptor) is returned to the user side of the process, through which the process can access the file.

The process' open file table entry contains a pointer to a **system open file table entry**. The main purpose of the **per-process open file table** is to hold the process' offset into the file (this offset is what is manipulated by the *lseek()* system call). Before the introduction of VFS, the system open file table entry contained a pointer to the in-memory copy of the file's inode. The in-memory inode is shared between processes (one purpose of the `v_usecount` field is to to keep track of this sharing).

The reason for the intermediate process table is flexibility in sharing (or not) the file offset. The offset cannot be stored in the in-memory inode table, since a process that opens this file shares this inode with every other process that has opened this file - but they all have different offsets into it. Storing the offset in separate per-process tables allows sharing the offset. The *dup()* system call will create a second entry in the per-process open file table pointing to the same system open file table entry. Similarly, *fork()* will copy the whole per-process open file table into the new process, so the new process will share offsets with the old process. If two processes open the same file (or a process opens the same file twice), each per-process open file table entry points to a different system open file entry (although in the same table), so the offsets are not shared.

In terms of implementation, the file descriptor that is passed to the kernel through a system call is used as an index into the per-process open file table, which yields a pointer to the system open file table. This translation is done by *getf()* in `kern_descrip.c`. The resulting system file table entry pointer is used by the system call to perform the required operation. The only thing that changes with the introduction of VFS is that instead of the system open file table containing pointers to in-memory inodes, it contains pointers to vnodes. In short, from a file descriptor, we get to a per-process open file table entry (`struct ufile_entry *`), then to a system open file table entry (`struct file *`), and from there we get to a `vnode`. (There is some information on debugging these structures in Section 4.7.2).

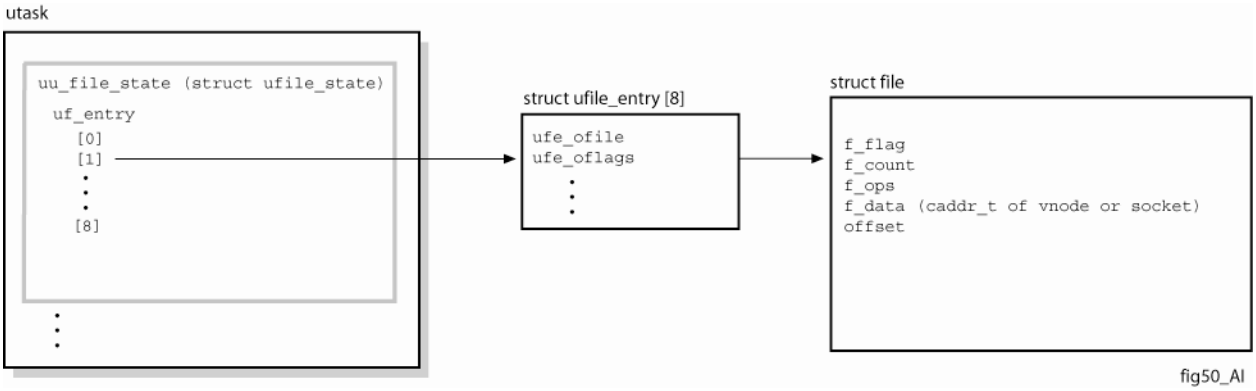


Figure 42: Per-task in-memory structures

14.1.1 Overview of VFS

AdvFS fits into the OS through the VFS/vnode mechanism. **VFS** (Virtual File System) adds a layer of indirection that allows a user process to access multiple underlying file systems transparently (i.e., NFS, MFS, CFS, FFM, AdvFS, UFS). In this section, we examine the VFS mechanism in more detail and examine how AdvFS fits into the scheme.

VFS was created in order to define the abstract data types "file system" and "file". Each type has its own data structure and set of abstract operations (a "class" in OOP terms). The set of operations is represented as a vector of function pointers. Each vector is implemented as a `struct`, rather than an array, so named fields (rather than numeric indices) can be used to index into it. The file system type is described in terms of a `struct mount` that looks like this:

```
struct mount {
        ...
        struct mount    *m_nxt;                 /* next in mount list */
        struct mount    *m_prev;                /* prev in mount list */
        struct vfsops   *m_op;                  /* operations on fs */
        struct vnode    *m_vnodecovered;        /* vnode we mounted on */
        struct vnode    *m_mounth;              /* linked-list of vnodes
                                                    on this mount point */
        ...
        qaddr_t         m_info;                 /* private data */
        ...
};
```

and a file is described in terms of a `struct vnode`:

```
struct vnode {
        u_int           v_flag;                 /* vnode flags */
        uint_t          v_usecount;             /* ref count of users */
        ...
        enum vtype      v_type;                 /* vnode type */
        enum vtagtype   v_tag;                  /* underlying data type */
        struct mount    *v_mount;               /* ptr to vfs we are in */
        struct mount    *v_mountedhere;         /* ptr to mounted vfs */
        struct vnodeops *v_op;                  /* vnode operations */
        struct vnode    *v_freef;               /* vnode freelist forwd */
        struct vnode    **v_freeb;              /* vnode freelist back */
        struct vnode    *v_mountf;              /* vnode mountlist forwd */
        struct vnode    **v_mountb;             /* vnode mountlist back */
        ...
        union {
              struct socket   *vu_socket;    /* unix ipc (VSOCK) */
              struct specinfo *vu_specinfo;  /* device specinfo struct */
              struct fifonode *vu_fifonode;  /* pipe and fifo struct */
        } v_un;
        struct vm_ubc_object *v_object;         /* VM object for vnode */
        ...
        char v_data[1];                         /* placeholder, private
                                                    data */
};
```

We will examine only a few fields from these structures. First, recall that a file system is mounted on an existing directory. That directory, like any file, is described by its own vnode. When you mount a file system, its mount structure is initialized and linked into a list of the system's mounted file systems through the `m_nxt` and `m_prev` pointers. The vnode of the mounted-on directory is stored in the `m_vnodecovered` field, and the `m_op` field is initialized to point to the (fixed) vector of operations for this type of file system (e.g. UFS, NFS, AdvFS, CFS, DFS, etc.).

Similarly, any open files and some closed files within this file system are described by vnodes, whose `v_type` field specifies the type of file system that the file belongs to. The `v_mount` field points to the mount structure of this file system. The vnodes of a given file system are linked together on the vnode mount list, maintained by the `v_mountf` and `v_mountb` fields. When a file is closed, its vnode is added to the vnode free list, but also stays on the vnode mount list. The free list is managed through the `v_freef` and `v_freeb` fields in each vnode. If the file is reopened quickly enough, we have a ready

vnode for it. Otherwise, the vnode is eventually recycled, at which point it is also removed from the vnode mount list in *vgone()*.

If the file is a mounted-on directory, then the `v_mountedhere` field points to the mount structure of the file system, allowing us to proceed downward through the hierarchy.

14.1.2 File System and File Operation Vectors

There are two more fields in each of the `mount` and `vnode` structures that remain to be examined. The first pair is the `m_op` field in the mount structure and the `v_op` field in the `vnode` structure. These are pointers to the corresponding vectors of abstract operations for the underlying file system. The second pair is the `m_info` field in the mount structure and the `v_data` field in the `vnode`. We will now take a closer look at these.

The following operations vector describes the file system set of abstract operations:

```
/*
 * Operations supported on mounted file system.
 */
struct vfsops {
        int     (*vfs_mount) __((struct mount *, char *, caddr_t, struct nameidata
*));
        int     (*vfs_start) __((struct mount *, int flags));
        int     (*vfs_unmount) __((struct mount *, int));
        int     (*vfs_root) __((struct mount *, struct vnode **));
        int     (*vfs_quotactl) __((struct mount *, int, uid_t, caddr_t));
        int     (*vfs_statfs) __((struct mount *));
        int     (*vfs_sync) __((struct mount *, int));
        int     (*vfs_fhtovp) __((struct mount *, struct fid *, struct vnode **));
        int     (*vfs_vptofh) __((struct vnode *, struct fid *));
        int     (*vfs_init) ();
        int     (*vfs_mountroot) __((struct mount *, struct vnode **));
        int     (*vfs_swapvp) ();
        int     (*vfs_smoothsync) __((struct mount *, u_int, u_int));
};
```

Some of the `vnode` abstract operations are shown below (see `usr/include/sys/vnode.h` for a complete list):

```
struct vnodeops {
```

```
        int     (*vn_lookup) __((struct vnode *, struct nameidata *));
        int     (*vn_create) __((struct nameidata *, struct vattr *));
        ...
        int     (*vn_open) __((struct vnode **, int, struct ucred *));
        int     (*vn_close) __((struct vnode *, int, struct ucred *));
        ...
        int     (*vn_read) __((struct vnode *, struct uio *, int, struct ucred *));
        int     (*vn_write) __((struct vnode *, struct uio *, int, struct ucred *));
        int     (*vn_ioctl) __((struct vnode *, int, caddr_t, int, struct ucred *, int *));
        ...
        int     (*vn_mmap) __((struct vnode *, vm_offset_t, struct vm_map *, vm_offset_t *,
                               vm_size_t, vm_prot_t, vm_prot_t, int, struct ucred *));
        int     (*vn_fsync) __((struct vnode *, int, struct ucred *, int));
        int     (*vn_seek) __((struct vnode *, off_t, off_t, struct ucred *));
        int     (*vn_remove) __((struct nameidata *));
        int     (*vn_link) __((struct vnode *, struct nameidata *));
        int     (*vn_rename) __((struct nameidata *, struct nameidata *));
        ...
        int     (*vn_inactive) __((struct vnode *));
        int     (*vn_reclaim) __((struct vnode *, int));
        ...
};
```

The common operations are used to open, close, read, and write files and directories; perform seeks and ioctls on files; get and set file attributes; and rename or remove files and directories. Some more-specialized operations are *vn_lookup()* to translate a pathname to a `vnode`, *vn_mmap()* to memory map a file, and *vn_inactive()* and *vn_reclaim(),* operations that mark a `vnode` as inactive or reclaim it. More on these later.

The `m_info` field in the mount structure contains data specific to the type of the underlying file system; different file system types maintain different kinds of data in this field. Similarly, the `v_data` field in the vnode usually contains data specific to the underlying file (AF_UNIX sockets, special files, and pipes use a different vnode field). Again, the data maintained here depends on the file system type and on the specific file. These fields and the operations vectors tie the abstract VFS/vnode layer to the specific physical file system.

The system finds these structures through a few global variables: `rootfs` points to the mount structure of the root file system; any other mounted file systems come afterwards in this list. There is also `rootvp` and `rootdir`, which point to the `vnodes` of the root device and the root directory respectively. Recall that the open file tables of many processes may point to vnodes. `Vnodes` cannot be reused while there is at least one reference to them. There is a reference count called `v_usecount` that implements this

policy. Note that the set of operations for a particular file system type is the same for any member of the given class. For example, every `vnode` of a UFS file system uses the same set of `vnode` operations. In general, each file system implementation defines a vector of operations as a static structure containing pointers to the functions that implement the operations. When this vector is initialized, a pointer to the vector is placed in the appropriate `vnode` or mount structure.

The illustration below shows what the edifice looks like for three mounted file systems and several open files. The illustration also includes a few global variables that the system uses:

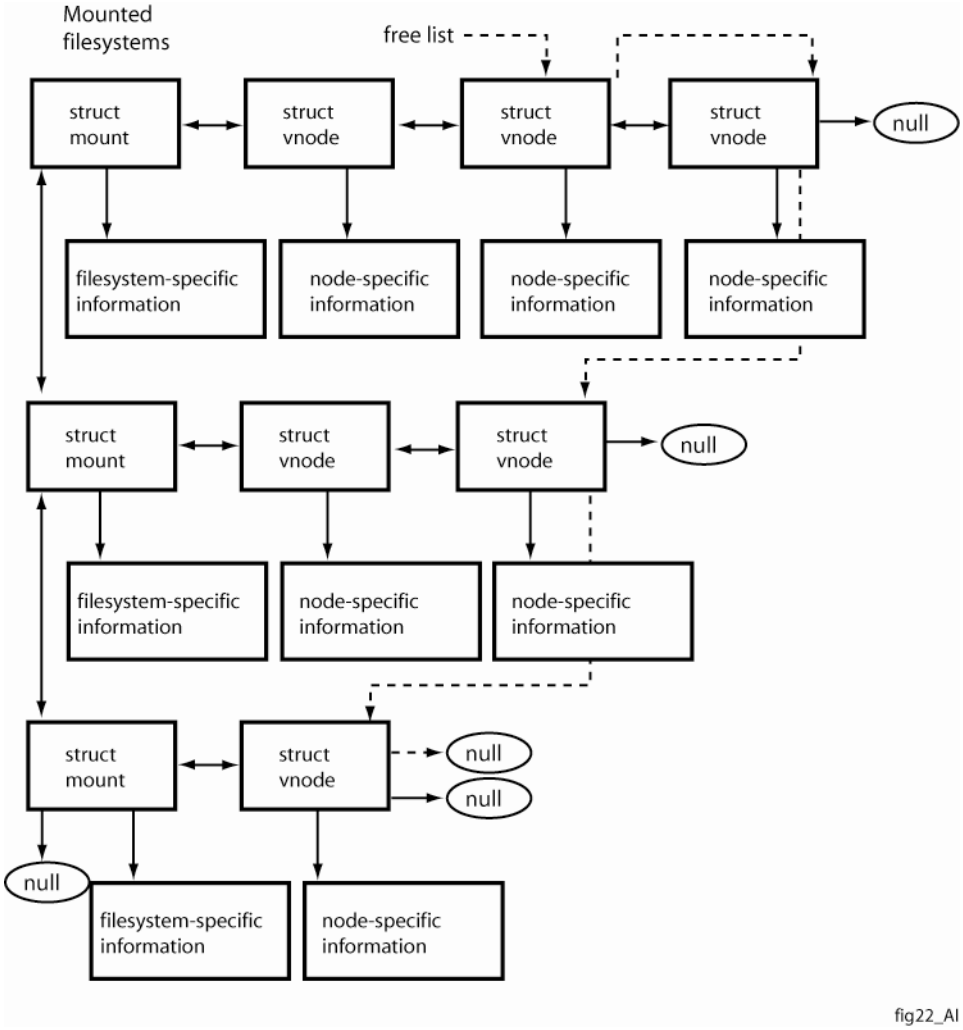


Figure 43: VFS structure relationships

14.1.3 How AdvFS Fits In

AdvFS must supply four pieces to conform to the above conventions: AdvFS-specific data will be stored in the `mount` and `vnode` structures, and implementations for the abstract VFS operations and vnode operations will be declared.

Let us first discuss the mount structure. The `m_info` field is a pointer to file system-specific data. In the case of AdvFS, the `m_info` field is a `fileSetNode` pointer. This structure contains information about the mounted AdvFS fileset. We are now firmly within the purview of AdvFS: the mounted filesets of this domain are linked together through the `fsNext` and `fsPrev` pointers in the `fileSetNode` structure. The `fileSetNode` structure contains a back pointer to the mount structure and pointers to the access structure and the vnode of the root directory of the fileset (see section 3.3.2 for complete details on the `fileSetNode` structure).

Similarly, the vnode `v_data` field contains file system-specific information about the particular file. In this case, however, the information is contained within the vnode, rather than being pointed to by a pointer. For AdvFS, this is a `bfNode` structure (see section 4.3.2).

The two operations vectors, one for the VFS functions and one for the vnode functions, are statically declared in `msfs_vfsops.c` and `msfs_vnodeops.c` and initialized with pointers to the functions that implement the corresponding abstract operations.

There are two classes of system calls for AdvFS. The first class contains VFS/vnode-related system calls which are dispatched normally. For example, a *read()* in user space ends up calling the `read` system call in `bsd/sys_generic.c`, which in turn calls `rwuio`. `rwuio` calls the macro `FOP_READ` to dispatch to the read operation of the file operations vector, which in turn calls `vn_read` in `vfs/vfs_vnops.c`. `vn_read` extracts the vnode pointer from the file structure, and calls the macro `VOP_READ` passing the vnode pointer as an argument. This macro then selects out of the vnode's operations vector the file system's "read" function pointer (which in our case is a pointer to the AdvFS function `msfs_read`) and calls it. At this point, we are within AdvFS territory: `msfs_read` gets a pointer to the access structure from the `bfNode` structure in the vnode and uses it to call lower-level functions within AdvFS to process the read.

The second class of system calls is used by AdvFS utilities to perform operations that are specific to AdvFS, like migrating a file from one volume to another or adding a volume to a domain. These system calls have nothing to do with VFS. They are dispatched through one giant switch statement in *msfs_real_syscall()* in `bs_misc.c`. They are mentioned here for completeness, but do not interact with the VFS/vnode layer. See section 13.1 for an AdvFS-specific system call stack trace.

14.1.4 Namei Cache

"Lookup" is the translator between names and vnodes. When a file is opened, we pass the pathname of the file to the kernel. The pathname is traversed component by component and translated to an open file whose vnode is the result of the lookup operation. This is done by *namei()* which starts at some directory whose vnode is known (either the root directory of the process or the current directory of the process) and keeps going down the pathname, looking up each component in the context of its containing directory. It also takes care of the detours that mount points, symbolic links and ".." represent. Each intermediate lookup returns a new vnode, the vnode of the directory that corresponds to the pathname that was resolved so far, which is then used in the next iteration in *namei()* to resolve the next component of the pathname. When *namei()* runs out of pathname, the lookup is done.

Namei() is file system-independent, but the lookup of a name in a directory is not, so it is implemented as a vnode operation. The filename is looked up in the name cache, and if it is found or knows it does not exist, the result is returned. If the filename is not found in the cache, the directory is searched (either sequentially or using the index file, if one exists). The result (positive or negative) is added to the cache and if positive, the resulting `vnode` is returned.

This is also the place where the `.tags` magic happens: if the name to be resolved is `.tags/something`, then the "something" is used directly as a tag to open the file (see section 4.3).

14.1.5 Vnode Recycling

Opening a file through the VFS layer (as opposed to internal opens) increments a reference counter in the `vnode` structure. When a file is closed, the reference counter is decremented. When the counter reaches zero (the last close), the vnode is made inactive by calling the `VOP_INACTIVE` `vnode` op and freed. Freeing the vnode puts it on the tail of a list of free vnodes which are recycled on a least-recently-used basis. All this is done by *vrele()*. When the system needs a vnode, it grabs one from the head of the vnode free list. While the vnode is on the free list, it still contains valid information. The vnode stays on the file system's open vnode list until the vnode is needed for another file (this means the vnode can be on two lists at once – the vnode free list and the file system's vnode list). If the file to which the vnode referred in its previous life is reopened before the vnode is recycled, the vnode is found on the free list, unlinked, and put into service again. The search is accomplished by a hashing mechanism that uses a combination of inode number/tag and device number to identify the correct vnode, so it is fast.

When the vnode finally arrives at the head of the free list and is grabbed by the system for a different file, the connection between the vnode and the file that it previously referred to has to be broken. In particular, the underlying file system needs to be told about the severance, in order to enable it to do file system-specific cleanup.

A new vnode is requested through a call to *getnewvnode()*, which decides whether to allocate a new vnode or to get one from the free list. If the latter, it unlinks the vnode from the free list and calls *vgone()* on it. *vgone()* checks for races (some other thread may have called *vgone()* on this vnode already), resets the vnode ops to `dead_vnodeops` (for catching and avoiding errors: any vnode ops that are called on the vnode will return error values and/or print error messages on the console) and calls *vclean()* on the vnode. *vclean()* disassociates the vnode from the underlying file system: it flushes any associated pages from the buffer cache and invalidates them. If the vnode is active, it has to be closed (`VOP_CLOSE`) and inactivated (`VOP_INACTIVE`); then it can be reclaimed.

The `VOP_CLOSE` vnode op for AdvFS does not do much: it just updates file stats. The `VOP_INACTIVE` vnode op breaks the pointer from the vnode's `v_data` field to the underlying AdvFS access structure and a `VREG` vnode type is changed to `VNON`. Calling the `VOP_RECLAIM` vnode op does the reclaim. In the case of AdvFS, this ends up calling *msfs_reclaim()* which checks for races, flushes any dirty pages of the VM object associated with the access structures and invalidates them in the UBC, arranges for dirty stats to be flushed to disk, and finally breaks the connection between the access structure and the vnode that it pointed to. There is a little more cleanup to do: *vnode_fscontext_deallocate()* cleans out the file context area (e.g. ensures quotas are detached), and clears out the VM object pointer in the vnode. At this point, the vnode is ready to be reused.

14.2 AIO Interface

The **Asynchronous I/O Interface (AIO)** is a code layer between the application and the kernel that allows synchronous I/O calls to be treated asynchronously by the application. This is done by having the application make special calls to the kernel (*aio_read()* instead of *read(); aio_write()* instead of *write()*). The AIO and kernel routines treat these I/O calls specially to allow the application to start each I/O (or series of I/Os) and then check for their completion later. This capability is particularly useful when an application is using raw or directIO (see section 8.12) because I/O in these modes is essentially synchronous.

AIO code handles AdvFS I/O starts in two steps.

1. The first I/O request initializes the vector of 64 result block pointers (rpb's) in the proc structure. These structures track the I/Os started with the AIO interface by each process. If the process exits before calling the I/O completion routines, the AIO subsystem delays the exit until all outstanding AIO-related I/Os are complete.
2. The application's calls to both *aio_read()* and *aio_write()* filter down through the *aio_rw()* routine in the kernel. Here an `rpb` in the `proc` structure is set up, the user's pages for this transfer are wired, and the appropriate kernel routine is called. For an AdvFS fileset, this routine is *msfs_strategy()*. The *aio_rw()* routine passes a `struct buf` to *msfs_strategy(),* which uses this information to set up a `uio` structure to pass to the AdvFS read and write routines.

The AdvFS read and write routines do little special processing for AIO calls. One special check is to ensure that after the I/O request has been handed off to the device driver, the kernel will return to the application without waiting for I/O completion. Another special step is for the I/O completion path to call the AIO completion routine for each I/O started through the AIO interface.

When a process using the AIO interface exits, it automatically calls *aio_exit()* to check if there are any outstanding AIO-induced I/Os. If there are, it will wait for them to complete and then do the appropriate cleanup. I/O completion for AIO happens in two distinct thread contexts.

1. For any I/O, the driver signals I/O completion and invokes the AdvFS I/O completion routine, *bs_osf_complete()*. If the I/O is determined to be associated with an AIO call, then the AIO completion routine (*aio_driver_done()*) is called. *aio_driver_done()* saves some information from the incoming `struct buf` and then casts that structure (which was the same `struct buf` that was passed into *msfs_strategy()* originally) to be an `aio_completion_qentry structure (acq)`. (This was done for performance reasons, and dictates that AdvFS be careful of how this structure is modified. See the final paragraph of this section.) This `acq` structure is then placed onto a per-Resource Affinity Domain (RAD) queue (`aio_completion_queue`) for later processing.

2. Another context in which I/O is completed takes place in the *aio_completion_thread()*. There is one such thread per RAD, and each is responsible for removing entries from the `rad->aio_completion_queue`, doing some cleanup, and then moving this `acq` structure onto the process' `p_aio_completion` queue for final cleanup.

From the AdvFS perspective, it is important to understand that the `struct buf` passed into *msfs_strategy()* is used for a completion queue entry after the I/O has completed. This means that the AdvFS code must not overwrite any part of this structure after an asynchronous I/O has been started. Conversely, if an asynchronous I/O fails to be started, the error code must be posted in the `struct buf` so that the caller knows that no I/O has been started.

14.3 LSM

The Logical Storage Manager (LSM) is a volume manager used to help manage data storage, and is a layer below AdvFS. LSM uses Redundant Arrays of Independent Disks (RAID) technology to enable configuration of storage devices into a virtual pool of storage from which LSM volumes can be created. New and existing AdvFS file systems, databases, and applications can be configured to use LSM volumes. LSM volumes can also be created on top of RAID storage sets. The benefits of using an LSM volume instead of a single disk partition include the following:

- Data loss protection: LSM can automatically store and maintain multiple copies (mirrors) of data or data and parity information. If a storage device fails, LSM continues operating using either the

remaining mirrors or the remaining data and parity information, without disrupting users or applications, shutting down the system, or backing up and restoring data. LSM can automatically transfer the data from the failed storage device to a designated spare disk, or to free disk space, and send you mail about the relocation.

- Maximized disk usage: LSM can be configured to seamlessly join storage devices to appear as a single storage device to users and applications.
- Performance improvements: LSM can be configured to separate data into units of equal size, then stripe the data to two or more storage devices over different disks and different buses.

14.3.1 LSM Terminology

LSM is used to manage storage space, while AdvFS is used to manage files and file systems. Applications, databases, and file systems access logical volumes, and LSM manages the composition of these logical volumes. A **LSM volume** is a virtual disk device that looks to applications and file systems like a regular disk-partition device and is formed by combining portions of several physical disks. The data contained on each volume can then be duplicated to provide redundancy.

A **physical disk**, or partition, is defined as a group of contiguous data blocks on a disk device, (SCSI disks, DSA disks, etc.) and is referenced in a form similar to `/dev/disk/dsk3d`. A **subdisk** is a logical representation of a set of contiguous disk blocks on a physical disk. Subdisks are the basic components of LSM volumes. Concatenation of subdisks gives LSM the ability to create enormous LSM volumes for use by file systems and enables LSM volume creation of small leftover pieces of disks.

A **plex** is a collection of one or more subdisks that compose one copy of the volume data. A plex is an instance of the volume data. A mirrored volume is comprised of two or more plexes, and data contained at any given point on each plex is identical, although the subdisk arrangement may differ. Plexes have either a striped or a concatenated organization. Volume mirroring using plexes improves data integrity. Any write operation is written to all the plexes, and any read operation can be satisfied by any of the plexes.

A **disk group** is a collection of disks that share the same LSM configuration database (LSM disks, subdisks, plexes and volumes). The root disk group, `rootdg`, is a special, private disk group that is created upon LSM initialization. `rootdg` contains information on all the other disk groups in the system's LSM configuration. Disk groups provide a way to partition the configuration database and enable disks to move between systems.

14.3.2 AdvFS and LSM Interactions

AdvFS's primary point of interaction with LSM is in the volume name space. User space utilities such as `addvol, rmvol, migrate`, and `mkfdmn` must accept traditional AdvFS volume names as well as LSM volume names:

```
rmvol /dev/disk/dsk3c test_dmn        AdvFS volume name
rmvol /dev/vol/rootdg/vol4            LSM rootdg group volume
rmvol /dev/vol/testdg/vol1            LSM testdg group volume
```

Internal kernel migration routines must be able to handle LSM volume names, as well. Other than the LSM volume name awareness, AdvFS functions independently of the LSM layer.

Both AdvFS and LSM support striping of data, but it is recommended that only one method of striping be used. For instance, configuring an AdvFS striped file that resides on an LSM striped volume is usually not recommended. See the Tru64 UNIX System Configuration and Tuning Guide for more information.

See the References section for pointers to more detailed documents about LSM.

References

Glossary

This is a list of some items that might be included in a glossary.

access rights	Portable abstraction of the operating system locks
access structure	Memory-resident structures that hold file information.
asynchronous I/O	A non-blocking I/O scheme where data is written to the cache and may return control to the calling application before the data is written to disk.
Atomic-write data logging	Guarantees that either all data in a write system call (up to 8 kilobytes) is written to the disk, or that none of the data is written to the disk.
Bitfile Access Subsystem (BAS)	The BAS is responsible for managing the filesystem's physical storage representation. This subsystem provides shared storage support of a domain's file systems, storage migration, and storage expansion.
bitfile	A set of pages that AdvFS views as one entity. Reserved files and user files are bitfiles.
Blocking queue	An I/O queue used primarily for reads and for AdvFS-generated synchronous write requests (e.g. log flushes).
BMT	Bitfile Metadata Table. An array of mcells that define and maps a file's data storage. All bitfiles are mapped by the BMT, including the BMT itself. Page 0 of the BMP maps ther domain's reserved bitfiles.
bfAccess structure	Contains bitfile attributes, the extent maps, a dirty buffer list and pointers to the domain structure and bitfile set descriptor for a file.
bsBuf structure	Headers associated with pages in memory that help control the I/O for those pages.
Buffer cache	A set of memory pages that contains data read from and/or waiting to be written to disk.
Clone fileset	A read-only copy of the original fileset's tag file that is maintained as it existed at the time of the clone.
Consol queue	A lazy I/O queue that holds a sorted list of dirty buffers waiting to be written.
Context structure	

continuation transaction	
COW	Copy On Write. The process by which original information is saved in a clone fileset when data in the original file is changed.
DDL	Deferred Delete List
Data management	
descriptor	A file page and corresponding starting disk block pair.
device queue	
direct I/O	An I/O scheme that synchronously reads and writes data from a file without copying it into the buffer cache.
DM application	A Data Management application is any application that uses the DMAPI.
DMAPI	Data Management Application Programming Interface. The term that refers to the interface defined by this XDSM Specification.
DMAPI implementation	The services in the host Operating System that act as the XDSM API provider
Directory	A collection of files that are logically related.
Disposition	Attributes describing a DMAPI session.
Domain	A named pool of storage that contains one or more volumes.
event	A notification from the operating system to a DM application about an operation on a file. For example, a DM application can arrange to be notified about attempts to read a particular file.
Extent	Contiguous area of disk space allocated to a file. A file may have zero or more extents.
Extent map	An in-memory or on-disk map of file pages to disk blocks. A subextent map is part of an extent map that will fit in one mcell.
Fast recovery	
File	An entity that stores information.
File Access Subsystem (FAS)	The subsystem responsible for managing the logical file hierarchy. This is the AdvFS representation of standard UNIX directories, files, and file systems.
file directory	A directory that associates a file name with its metadata.

Fileset	A hierarchy of directory and files. A fileset represents a mountable portion of the directory hierarchy of the AdvFS file system.
Fileset tag	
flush queue	An I/O queue that holds pages waiting to be flushed to disk from an explicit flush request such as fsync().
frag file	A special file that stores the last partial page of a file in a fileset. Using frag files reduces the amount of wasted disk space.
fragment	File storage of less than 8K.
fragment bitfile	A collection of fragment groups.
fragment group	An array of fragments of uniform size.
free-space cache	An in-memory cache of disk space available for each volume in each domain.
fsContext structure	Contains file attributes and disk quota information.
Hole	An area of a file that consists entirely of bytes of zeros. Some file system implementations may not need to consume any media resources to maintain such an area.
inode	A numeric file identifier.
I/O transfer size	A volume-specific value that indicates the amount of data that can be processed in a single I/O request.
I/O consolidation	A performance optimization that combines the I/O for several discrete buffers into a single I/O request to the device driver
lazy queue	A logical series of queues in which asynchronous write requests are cached.
LBN	Logical Block Number . The address on the storage device at which a given page of information is stored.
logical file hierarchy layer	The software that implements the file-naming scheme and POSIX-compliant functions for AdvFS.
logical sequence number	A unique identifier of a transaction log entry.
magic number	A unique number (the time since Chris Sather's birthday) that identifiies a domain as AdvFS at mount time.
managed file	A file that is being monitored by a DM application for events.
managed region	A contiguous span of a file given as an (offset, length) pair, together with an associated event generation specification.
Mcell	Metadata cells that contain records of file structure.

mcell address tuple The address of an mcell; consists of the volume index, BMT page number, and the mcell's index within the BMT page.

metadata File structure information such as file attributes, extent maps, and fileset attributes.

miscellaneous bitfile Maps areas of the volume that do not represent AdvFS metadata, such as the disk label and boot blocks.

namei cache

non-reserved file

NUMA Non-Uniform Memory Access. A memory access system in a cluster that gives preference to certain processors.

object reuse The reuse of storage in such a way that data from previous objects are revealed in the current object.

object safety Forces newly-allocated storage to be zero-filled and written to disk before it is made available to the file so there can be no object reuse.

opaque An opaque variable is one that should not be interpreted by an application. It should be thought of as a block box.

opaque extended attributes

outstanding Pending or waiting for action.

page An allocation of 8 KB of contiguous disk space (16 blocks).

physical storage layer The software that implements the logging, caching, file storage allocation, file migration, and physical disk I/O functions for AdvFS.

punching holes An action taken by a DM application to release on-disk blocks of a file to free up space on a local file system.

preferred transfer size The optimum number of pages in an I/O request for a given volume.

prefetch An algorithm to read and cache more than one page during a single read request.

primary extent map record

primary mcell

RAD Resource Affiity Domain.

RBMT Reserved bitfile metadata table. Contains the disklabel, BMP, boot block, RTD, SBM, and transaction log,

read-ahead	An algorithm to read and cache pages before an application requests them. Used during sequential reading of a file.
ready queue	A lazy I/O queue that contains dirty buffers that are sorted by LBN.
record	
reserved metadata file	
RTD	Root tag directory. A domain reserved bitfile used to locate file systems (for example, as a mount point).
root tag file	A directory that defines the location of all filesets in a domain. Each file domain has one.
saveset	The collection of blocks created by the vdump utility.
SBM	Storage Bitmap. Keeps track of allocated disk space in a volume.
secondary mcell	
service class	Determines which volumes within a domain can allocate file storage for new data or data migration.
Session	Communication channels used between a DM application and the kernel component of DMAPI
smoothsync	A mechanism whereby the dirty buffers are drained to disk at a constant, time-based rate.
smoothsync queue	A lazy I/O queue that contains dirty buffers that have not aged sufficiently to be flushed to disk. It is only used when smoothsync is enabled.
striping	
synchronous I/O	An I/O scheme where data is written both to the cache and to the disk before the write request returns to the calling application.
tag	A unique identifier for an AdvFS file within a fileset. It locates the initial link to the on-disk file metadata.
tag directory	An array of tag entries that identify the volume, page, and mcell location for a file. [correct?]
tertiary storage	
token	A reference to state associated with a synchronous event message.
transaction log	The log that records changes to metadata before the changes are written to disk. These changes are written to disk at regular

	intervals.
transactionally consistent storage deallocation	Requires that all transaction records in the log associated with a storage deallocation transaction must be committed to the disk log before the deallocated space can be reallocated
trashcan	A directory that contains the most recently deleted files from and attached directory. Trashcan directories can be set up by each user for user files.
UBC	Unified Buffer Cache. The dynamically allocated system buffer cache that holds file data and AdvFS metadata.
ubcreq queue	An I/O queue that caches synchronous write requests that come from the UBC..
virtual memory subsystem	
wait queue	A lazy I/O queue that contains metadata pages that cannot be flushed until their log entries are flushed to disk.
volume	For AdvFS, anything that behaves like a UNIX block device. This can be a discrete disk, an LSM volume group, or an HSG80 with many disks.
volume descriptor	Describes the disks in a domain.
volume descriptor array	Contains pointers to active volume descriptors.
volume index	
write-ahead logging	The process by which the modifications to the file metadata are completely written to a transaction log file before the actual changes are written to disk.