Compact Disc Digital Audio (CDDA) Filesystem Module Design Specification

Author: Chris Sarcone
Group: Mass Storage Software
Status: Version 1.0
Date: 11/17/09

Change History

Version	*Section/Page Affected*	*Change Description*	*Author(s)*	*Release Date*
Draft 1	*All pages*	*First version*	*C. Sarcone*	*11/17/09*

1.0 Problem Set

Mac OS X needs to be able to mount Audio CDs as discs in the Finder and allow users to copy the music tracks on the discs to another location, to play the music from the tracks in applications such as iTunes and QuickTimeX, and to view the names of the tracks in the Finder window. All applications shall be able to access the musical tracks while the disc is mounted.

The goal of this project is to create a filesystem, the CDDA FileSystem, which mounts the digital audio portions of a CD and allows the Finder and other applications to read this digital data as they would any other file. The files will be presented as AIFF-C-encoded sound files with the attributes of 16-bit stereo sound and 44.1kHz sampling. This means the Finder can directly copy files from a CDDA filesystem to another filesystem (e.g. HFS+) and the data will be preserved in the copy. The CDDA FileSystem will support all required VFS operations for BSD compatibility which in turn guarantees compatibility with Cocoa and Carbon clients as well.

1.1 Definitions, Acronyms, Abbreviations

Address Space	- The memory allotment for a task (sometimes called a process)
AIFF	- Audio Interchange File Format
API	- Application Programming Interface
ATAPI	- Advanced Technology Attachment Packet Interface
Block	- The smallest amount of data which can be read from a storage device
BlockSize	- The size (in bytes) of a block
BSD	- Berkeley Standard Distribution (a UNIX variant)
CD	- Compact Disc
CD-ROM	- Compact Disc Read Only Memory
CDDA	- Compact Disc Digital Audio (sometimes written as CD-DA)
daemon	- pronounced de-mon, this is an application which is constantly running on a UNIX-like operating system
DVD	- Digital Versatile Disc (sometimes incorrectly called Digital Video Disc)
DVD-RAM	- Digital Versatile Disc Random Access Memory
DVD-ROM	- Digital Versatile Disc Read Only Memory
IEEE-1394	- A high speed serial bus used for data transfer
FireWire	- Apple's implementation of IEEE-1394
Frame	- Approximately 1/75 of a second
HTTP	- Hyper Text Transfer Protocol
KB	- KiloBytes (1024 bytes)
Kernel	- The "guts" of OS X which includes mach, virtual memory, IOKit, and anything else residing in the kernel's address space
Mach	- The Microkernel designed at CMU by Avie Tevanian and colleagues

	The term Mach really only describes the microkernel, but is sometimes colloquially used to mean the entire kernel address space and execution layer
Mach/BSD	- Term used to describe the aggregate foundation on which OS X is built. Some of the foundation comes from BSD, and the core of BSD has been replaced with the Mach microkernel.
MB	- MegaBytes (1024 KB)
MSF	- Minutes, Seconds, Frames
SCSI	- Small Computer Systems Interface
TOC	- Table of Contents - Meta-data associated with a Compact Disc used to identify the number of sessions and tracks on the media
Track	- A song on an Audio CD
TCP/IP	- Transmission Control Protocol / Internet Protocol
UNIX	- An Operating System developed in the 1970s a type of which (BSD) is at the core of OS X
URL	- Uniform Resource Locator
VM	- Virtual Memory. Term used to describe the use of secondary memory (e.g. hard disk) as physical primary memory (e.g. RAM).
VFS	- Virtual FileSystem
VNODE	- Virtual node

1.2 References

1.2.1 MacOS X References:

IOKit documentation found at <coreos.apple.com/iokit/>
Mach 3.0 documentation found at <www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/documents_top.html>
BSD documentation found at <www.freebsd.org>, <www.netbsd.org>, <www.openbsd.org>
Core Foundation documentation found at <developer.apple.com>

1.2.2 Stackable FileSystem References

McKusick et al., The Design and Implementation of the 4.4 BSD Operating System, Addison-Wesley

Heidemann and Popek, “File-System Development with Stackable Layers,” ACM Transactions on Computer Systems, vol. 12, no. 1, pp.58-59, February 1994.

1.2.3 CD/DVD References

ATAPI SFF-8020i
ATAPI SFF-8090i
SCSI Multimedia Commands (MMC-2) www.t10.org

1.3 Code

1.3.1 Code Access

The CDDA Filesystem code is one of many VFS plugins. As such, the code for it is kept with all other VFS plugins (instead of in the Mass Storage SVN repository).

Access to the code is through SVN at the URL:
svn+ssh://src.apple.com/svn/fs/cddafs

1.3.2 Code Integration

The CDDA Filesystem project follows the branching and tagging mechanism used by CoreOS and Mass Storage Software. The project includes a branches, tags, and trunk directory, with the latest and greatest code available on trunk. The code on trunk shall always build properly. No changes shall be checked in on trunk unless they have been approved through the Core OS CCC process (https://coreos.apple.com/cgi-bin/wiki?keywords=CCC_Process). CDDA Filesystem, along with other VFS plugins, is traditionally integrated like other CoreOS deliverables (i.e. a separate integrator exists and handles merging, tagging, and submitting the code). Engineers are simply responsible for coding fixes, checking them in on branches, getting them peer-reviewed, tested, and provide information for the CCC approval process.

1.3.3 Code Building

The CDDA Filesystem code uses Xcode as its native build environment. You can use the Xcode GUI or xcodebuild from the CLI to build this project. It is recommended that you use 'buildit' to build the project and install it on your system:

sudo ~rc/bin/buildit . -arch i386 -arch x86_64 -merge /

After you run 'buildit', if the kext is not loaded, then you can insert a CD and run the new code. If the kext is loaded, unmount and eject all cddafs volumes and wait 1 minute for the kext to unload. Once it has unloaded, you may insert a new CD to cause the new code to load and be used.

If you wish to load the code manually, you may use 'kextload' to load the kext. Run 'man kextload' and man 'kextutil' to find out more information about loading code.

To validate the proper code is running, use 'kextstat'.

1.3.4 Target Footprint

The code shall be small (< 100K for multi-architecture builds), since it simply has to implement a few methods to be a filesystem and the runtime footprint shall be small as well (<50K), since only a handful of data structures are required for the filesystem to do its work.

2.0 Dependencies

The CDDA Filesystem relies on the kernel proper (particularly the BSD part of the kernel) to operate. Additionally, it relies on the system utility called 'mount' which is used to invoke mounting of the filesystem, the 'umount' utility used to unmount a filesystem, and the DiskArbitration subsystem, which allows seamless automounting of filesystems when a disc is inserted and a login session is active on the console.

No code directly relies on CDDA Filesystem, but some code does require that CDDA Filesystem implement certain features. iTunes and Music Player require the XML file called ".TOC.plist" to exist at the root of the filesystem. This metadata file is prefixed with a period (and marked invisible in FinderInfo flags) in order to hide it in the default Finder user interface. This file contains information about the Table Of Contents of the Audio CD and is used by iTunes to detect where tracks start on the raw disc. This file is sometimes referred to as the "XML File" elsewhere in the documentation.

3.0 Overview of AIFF file format

Part of the overall goal for CDDA Filesystem was the ability for any music player or movie player to be able to play back the audio content in the audio tracks. Apple did not want to invent a new file format specifically for Audio CD content, as it would take some time for developers to adopt it. Engineering found a solution in the AIFF file format. The AIFF file format uses QuickTime atom container chunks to describe data and meta-data about the audio content of the file. One of these 'atoms' describes the file as a 16-bit sample at 44.1kHz with data arranged in little endian (the content comes across an ATA bus and is defined to be little endian). Engineering chose this form of AIFF file, called AIFF-C, as the type of file format to use because it would reduce the number of byte swapping operations on the file data, it would allow for easy drag-and-drop support, and it would allow all current media players which used QuickTime or which understood how to decode and play AIFF-C files to play back the content of each file.

Since the tracks on the Audio CD only include the data, the CDDA Filesystem synthesizes these QuickTime 'atoms' at the beginning of each file, and then appends the audio data from the disc at the end of each track. A visual representation can be found in Figure 5-4.

4.0 High level overview

4.1 Background history

When Mac OS X was in development, there were multiple ways to play back Audio CD content. The first required the optical drive in the Mac unit to have an analog (wired) connection directly from the back of the drive to an audio input source on the MLB. This was unattractive because it caused cable routing issues and required specialized sound hardware to pull the data in and play it back. Additionally it required extra commands to be sent to the drive to play audio data this way. In the last versions of Mac OS 8.x, systems started shipping with optical drives that could play back Audio CD content as digital data. This data was called CDDA, or Compact Disc Digital Audio. The idea

behind this type of playback is that all audio data is “read” from the disc, just like regular file data. Instead of sending special commands to the drive to enable playback, an application can simply read the data from the disc, format it appropriately for the sound device, send the data to the sound device, and the sound device plays back the audio data. This new method of playback allowed for several cost reductions on the device side and on the MLB, so new hardware was only capable of CDDA playback.

Additionally, the Mac OS 8.x and 9.x experience for Audio CDs was not very good. When the user inserted an Audio CD, the CD mounted and presented Audio Tracks for each track on the disc. However, double-clicking a file always opened Apple CD Audio Player, which sent commands to the drive to perform analog playback or, if CDDA was supported, it told the CD/DVD driver to send Audio Data directly to the Audio device driver! Additionally, drag-and-drop simply copied alias entries to another partition, which again would simply open Apple CD Audio Player in order to play the song. No copying of the data was possible. Sadly, direct manipulation principles which were the heart of the Mac experience were not used.

At the time the CDDA Filesystem was first thought about, the Mac OS X landscape was much different than it is today. CDDA Filesystem was targeted as a feature for Developer Preview Release 4, the last release before Mac OS X, 10.0 would be released to the public. There was no iTunes application for OS X, but its precursor, SoundJam, existed as a third party product. MP3 files were the new craze. So, the Mac OS X team decided to build an Music Player application that could use QuickTime to play MP3s and hopefully Audio CD content.

So, given two clear issues at the time, the first being all new devices removing support for analog playback and moving to digital data, and the desire of the Music Player application team to have a simple, file-based interface for playing back audio files like MP3s and AIFF files, and given the tight schedule for this application to make it into DP4, Apple engineers brainstormed and came up with the idea of exposing Audio CDs with the files as true audio files. The audio files could be played back by Music Player, QuickTime Player, SoundJam, and other third party apps that used QuickTime for playback. Additionally, direct manipulation principles were espoused so that copying the file from the disc to a local drive actually copied the sound data and created a file which could be played back by the aforementioned applications as well. The Music Player team and the CDDA Filesystem engineers performed a “meet-in-the-middle” approach, which allowed each to develop against a known standard (AIFF-C files), and that allowed an easy, decoupled development model.

By presenting the audio data on the CD as just file data, the CDDA Filesystem achieved its number one goal relatively easily. File data was file data and could therefore be copied from disc to another location with all data preserved. Additionally, since an established file format was used for file data, it allowed QuickTime, Music Player, SoundJam,

iTunes, QuickLook and others to easily provide playback and previews. Finally, since the Audio CD had a mountable filesystem, it allowed DiskArbitration and the Carbon File Manager and NSWorkspace manager to be used to eject the disc once the user was done.

Because the CDDA Filesystem simply appends a header to each file and never performs any transforms on the data supplied by the drive, the data rate at which file data can be delivered is 100% based on the rate at which the hardware performs. This allowed the filesystem to meet all targeted performance goals (copy a 4 minute song in ~15 seconds) and keep a low footprint.

Since CDDA Filesystem is implemented as a Virtual File System plugin, it slots into the Mac OS X system design similar to other Virtual File System plugins. Mac OS X has a VFS layer that abstracts out the idea of files in a filesystem (See Figure 4-1). It also integrates with the disk buffer cache / vm page cache called the Unified Buffer Cache (UBC).

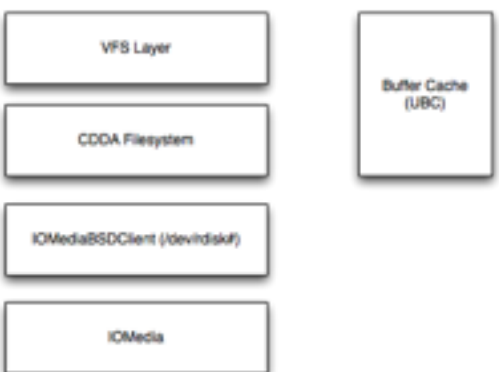

Figure 4-1. CDDA stack-up.

CDDA Filesystem implements required routines to lookup files, create virtual nodes (vnodes) to represent files, and read data and attributes (metadata) for each node (See Figure 4-2). As far as filesystem implementations go, CDDA Filesystem is an extremely easy filesystem to implement. At most there may be 99 tracks on an Audio CD, and none of the tracks exist inside other containers (i.e. there is no hierarchy with folders and files).

This is referred to as a “flat” filesystem. It makes the lookup of nodes extremely easy (and fast), and obviates the need for a complex locking strategy when traversing a directory hierarchy because there isn’t a hierarchy!

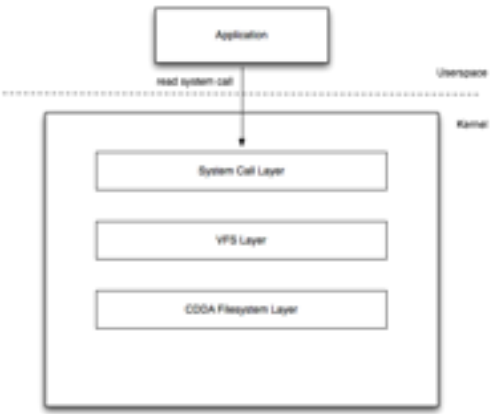

Figure 4-2. Example of software layers for read system call.

4.2 CDDA Components

In order to mount a CDDA Filesystem, there are several software components that interact with each other to perform the mount.

4.2.1 Filesystem Bundle (cddafs.fs)

During boot, diskarbitrationd searches the /System/Library/Filesystems directory for any bundles that end in “.fs”. It validates the bundle (mostly by looking for the Info.plist inside it) then catalogues all of these bundles and keeps an internal database of filesystems and media types.

4.2.2 Utility Program (cddafs.util)

Whenever an IOMedia object is registered in the IORegistry, diskarbitrationd is notified of its existence and tries to match a filesystem to it. diskarbitrationd looks at the Content Hint key in the IOMedia object and decides which filesystems’ utility program to call.

For an Audio CD media object, cddafs.util gets called to probe the device and tries to match against it. If it recognizes the disc as one which contains audio tracks, it returns that it recognizes the disc and proposes a name for the mountpoint (e.g. “Audio CD”). This name can be localized or it can be a name from the user’s iTunes library. Once diskarbitrationd creates the mount point in /Volumes, cddafs.util is called again and told to mount the filesystem on the disc. It then checks to make sure the filesystem is resident in the kernel and if it is not there, loads it into the kernel (via a fork() and exec() of the ‘kextload’ utility). It then calls the mount() system call. The mount() system call will send the mount request to the filesystem specific mount program which mounts the filesystem.

The CDDA FileSystem utility resides in /System/Library/Filesystems/cddafs.fs/Contents/Resources/. When called, it gets the disk name from diskarbitrationd (via command line arguments) and parses the options sent via command line arguments. It then executes the command, typically probe or mount.

If it is asked to probe the medium, cddafs.util gets the TOC data out of the IORegistry via some CoreFoundation and IOKit Framework calls and parses the data to find out if there is at least one audio track on the disc. If the utility finds that there is at least one audio track, it returns an identifier letting diskarbitrationd know it wants to mount on this medium.

If asked to mount the medium, the utility calls the private mount tool (see § 4.2.3) via the mount utility with the -t flag set to identify the filesystem type as “cddafs”.

4.2.3 Mount Tool (mount_cddafs)

The mount tool is a user space application which shall only be invoked via the mount() system call (and generally only by diskarbitrationd). The mount tool is responsible for checking if the kernel extension has been loaded into the kernel yet and loading it if it is not presently loaded. After it does that, it attempts to mount the filesystem via the mount() system call. It passes the arguments to the filesystem and waits to see if there is an error.

An enterprising user can call mount_cddafs from the command line as well in order to mount the media. However, diskarbitrationd will not be aware of this mount and it may present issues across unmount.

4.2.4 Kernel Extension (cddafs.kext)

Kernel extensions are bundles (see CFBundle documentation for more details) which commonly end in .kext. The CDDA FileSystem kernel extension, cddafs.kext, follows the CFBundle conventions of having a Contents directory in which there is an XML file in Apple PList (short for property list) format, a PkgInfo file which describes the bundle format, a version.plist which is generated by the Build and Integration group to keep

track of which version of sources was built with, and a MacOS directory in which the executable kernel module is stored. The kernel extension has the most code of any piece in the project. The kernel module sources are comprised of four C-source files, one C++ source file, five C/C++ header files, and some associated information in the build settings and bundle settings. Most of the filesystem code is written in C, but in a few instances it was necessary to move some functionality over to a separate file which would work with the C++ compiler (mostly related to the Table of Contents parsing and getting data from the IORegistry).

The kernel extension is loaded on demand by the utility programs and can be dynamically unloaded by the kernel extension daemon (kextd) when no CDDA filesystems are currently mounted. Currently, kextd waits one minute after the last unmount of a CDDA filesystem before it unloads the code.

5.0 Data Structures, Classes, Functional Units

5.1 VFS Interfaces

CDDA Filesystem is written in object-oriented C. That may sound funny, but think of it this way: it's implemented like a C++ class, except instead of being a class with a virtual function table that implements each required method, it instead builds tables of routines (*struct vfsops* and *struct vnodeopv_desc*) and registers these tables with the VFS layer when the kext is loaded into the kernel. The vfs operations structure describes filesystem-level, as opposed to file/vnode level, operations to be performed, such as mounting a new filesystem, remounting an existing filesystem, getting the root node of the filesystem, get attributes about the filesystem, and unmounting an existing filesystem.

Table 5-1 lists the VFS operations supported and the mapping to CDDA methods. All other VFS operations are unsupported (e.g. no support for NFS exports of CDDA files).

VFS Operation	CDDA Operation
vfs_mount	CDDA_Mount
vfs_unmount	CDDA_Unmount
vfs_root	CDDA_Root
vfs_getattr	CDDA_VFSGetAttributes
vfs_vget	CDDA_VGet

Table 5-1. VFS to CDDA mapping table.

The vnode operations structure describes operations to be performed at a file/vnode level, such as lookup, open, close, read, pagein, get extended attributes, etc (See Table 5-2).

VNode Operation	CDDA Operation
vnop_lookup	CDDA_Lookup
vnop_open	CDDA_Open
vnop_close	CDDA_Close
vnop_getattr	CDDA_GetAttributes
vnop_read	CDDA_Read
vnop_remove	CDDA_Remove
vnop_rmdir	CDDA_RmDir
vnop_readdir	CDDA_ReadDir
vnop_inactive	CDDA_Inactive
vnop_reclaim	CDDA_Reclaim
vnop_pathconf	CDDA_Pathconf
vnop_pagein	CDDA_Pagein
vnop_blktooff	CDDA_BlockToOffset
vnop_offtoblk	CDDA_OffsetToBlock
vnop_getxattr	CDDA_GetXAttr

The CDDA Filesystem registers these routines during its Apple_CDDA_FS_Module_Start() routine, which is the first thing called when the kext is loaded into the kernel. This code creates a *struct vfs_fsentry*, which describes the VFS operations, the vnode operations, flags for the filesystem, and the filesystem name.

Finally, CDDA Filesystem requires its own mount arguments structure to be passed in along with the device node used for mounting the filesystem. This arguments structure includes things like the number of audio tracks on the disc, a blob of data which includes the name of each track, another blob of data in XML format for describing the tracks on the disc (used by iTunes and Music Player). Finally, it passes a file type and creator, used on OS X to associate the files presented with the default playback app (iTunes). See § 5.3.1 for more information about the mount arguments data structure.

5.2 SCSI Multimedia Commands data structures

The CDDA Filesystem only mounts on Audio CDs. CDDA Filesystem uses the SCSI Multimedia Commands standard and the table of contents (TOC) of the disc to figure out which tracks are audio and which have data.

Once a disc has been inserted into an Optical Disc Drive, Mac OS X's storage stack reads the TOC from the disc and determines how many tracks and sessions are on the disc, based on the TOC Data format 2h. TOC Data format 2h describes the size of the TOC data, first and last session numbers, and is followed by several data structures which define each track and session (See Figure 5-3). Inside each descriptor, there is a "control" field which describes the track as containing data or audio data. CDDA Filesystem utilizes this information to determine if there are any audio tracks on the disc and whether or not it shall try to mount the filesystem.

It shall also be noted that Compact Discs were originally created solely for the use of audio data, and therefore a block size of 2352 bytes was used.

5.2 AIFF-C structures

AIFF-C utilizes the QuickTime atom container format. As noted in Figure 5-4, it contains several atoms inside it to describe the file and file data. All atoms contain a header which has a 32-bit unique ID (legacy Mac OS four character code), and size for the atom data. Each chunk then has its own internal data. The most important of these chunks is the ExtCommonChunk which describes the number of audio channels, the number of sample frames, the sample size, sample rate, and compression type. This structure is utilized to describe the audio data as it is streamed from the audio disc (16-bit stereo, 44.1kHz, little endian). Additionally the SoundDataChunk is important because it tells the playback application where the actual audio data starts. CDDA Filesystem utilizes this chunk to align playback on a natural block size for audio CDs (2352 bytes).

Figure 5-4. AIFF-C file format.

5.3 CDDA data structures

5.3.1 CDDA Args (*struct AppleCDDAArguments)*

When mounting a CDDA Filesystem instance, several arguments are passed to the filesystem from user space. To accomplish this, the mount() command takes an argument structure (the first entry of which shall be the path to the device on which the mount will occur), followed by filesystem specific data. *struct AppleCDDAArguments* passes a wealth of information from user space to the kernel, including the number of audio tracks found on a disc, a data blob for the album/track names, a data blob representing the TOC in XML format, and the file type and creator to be used for the audio tracks on the disc. As part of the mount process, these arguments are copied into the kernel and stashed in the struct *AppleCDDAMount* and other data structures.

5.3.2 CDDA Mount (*struct AppleCDDAMount*)

The CDDA Mount structure is allocated at mount time and is associated with the *struct mount* used to refer to the newly mounted filesystem. Code interested in the *struct AppleCDDAMount* data can call vfs_fsprivate() and pass the *struct mount* in order to get to this data.

The *struct AppleCDDAMount* contains a pointer to the root vnode of the filesystem (the filesystem is said to 'cover' this vnode). For instance, if you insert a disc for the first time, diskarbitrationd will create a folder in the HFS+ filesystem you rooted from in /Volumes/ called "Audio CD". So, the root vnode is the one that points at /Volumes/Audio CD. The instance of cddafs is mounted on / Volumes/Audio CD.

Note that the CDDA Filesystem keeps a refcount on the root vnode until the filesystem is unmounted by calling vnode_ref(). It does not keep an iocount on the vnode. For more discussion about the difference between refcounts and iocounts, see Developer documentation at **TBD.**

The *struct AppleCDDAMount* also contains the root vnode ID, a 32-bit number which should not change because the filesystem keeps a refcount on the vnode.

struct AppleCDDAMount contains a pointer to the XML file vnode. This vnode is similar to the vnodes which are created for the audio tracks in that it can be generated on the fly via lookup(), it can be recycled, etc. However, since it isn't a track, it doesn't have certain attributes like the starting LBA, so it is kept associated with the other mount data. Associated with this vnode pointer is the xmlFileFlags which, in combination with the mount lock, is used to prevent multiple threads from calling lookup() and getting a different vnode created and returned. The XML file data itself is allocated in the kernel and a pointer to it is kept in *xmlData* and the size kept in *xmlDataSize*.

The CDDA Filesystem includes a feature which allows it to extract data from iTunes' database and use that information to get album titles and track names and display this information to the user for each of the tracks on an Audio CD. To accomplish this, the userspace utilities such as mount_cddafs or cddafs.util, which are used to mount the disc, extract this information from the CD Info.cidb file in the user's ~/Library/Preferences folder. The utility then passes this information to the kernel via the mount() system call. In the CDDA_Mount() routine, the arguments structure is passed in and it copies the additional data for the nameData. The nameDataSize is set to be the size of the data copied into the kernel so that it can be freed when the filesystem is unmounted.

Perhaps one of the most important items kept in the *struct AppleCDDAMount* is a pointer to the *struct AppleCDDANodeInfo* array (*nodeInfoArrayPtr*). This pointer is what is used during the CDDA_VGetInternal() operation which looks up vnodes asked for by name via CDDA_Lookup() (and its caller lookup()) or via a persistent vnode identifier via CDDA_VGet() (and its caller vget()). The number of elements in this array is stored in *numTracks*.

struct AppleCDDAMount contains mutex locks and their associated lock groups and attributes. This is the central lock, for each instance of the filesystem, that provides the synchronization for operations such as lookup and reclaim.

Invariably, applications like Finder, tar, backup utilities, etc., want to know the mount time for the volume, so the mount time is stored inside this struct as well (it can't be written to the disc after all because the disc is read-only).

Lastly, since OS X has a long history of applications being associated with files, the CDDA Filesystem synthesizes FinderInfo data for each audio track. The mount structure holds the optional fileType and fileCreator attributes which can be passed in via the mount options.

5.3.3 CDDA Node (*struct AppleCDDANode*)

The CDDA node is the basic data structure attached to every virtual node represented by the filesystem. It contains the nodeType (e.g. track, directory), a pointer to the vnode associated with it, a pointer to the vnode representing the block device from which the data will be read, and a nodeID. The nodeID is used for persistence. By convention, the root directory of the filesystem is node number 2. In order to make mapping of nodeID to track ID easy during CDDA_Lookup() and CDDA_VGet() operations, we use a convention of offseting the track number by 100 (i.e. track number 1 = node ID 101). This convention was arbitrary but chosen for the following reason: Audio CDs have at most 99 tracks. The first contiguous namespace of 99 tracks that did not include the root filesystem node

ID of 2 is 101-199. We could have started at ID 3 and subtracted 2 from every node ID to get the track ID, but doing thing this way makes it easy when looking at data structures in debug output or in the debugger. Since we allocated 100-199 for audio tracks and ID 2 for the root filesystem ID, there was only one file ID that needed accounting for: XML File ID. File ID 3 was chosen as it is the next available number.

After the nodeID, there is a union of the different types of nodes found in the filesystem. The union covers directory, file, and xml file. The structures for each are covered in later sections.

5.3.4 CDDA Directory Node (struct AppleCDDADirectoryNode)
The *AppleCDDADirectoryNode* structure is used to describe the top-level directory in which all the file nodes are found (including regular files and the XML file). The node includes information on the number of directory entries and the directorySize in bytes. This information is returned in CDDA_VFSGetAttributes() and in CDDA_GetAttributes() when called on the root directory node.

5.3.5 CDDA File Node (*struct AppleCDDAFileNode*)
struct AppleCDDAFileNode includes the AIFF header data used to make the audio track data look like an AIFF-C encoded file. Additionally, the file node structure has a pointer to the meta-data for each node (nodeInfoPtr), which is referenced from the mount point. This allows the node to get recycled and created again later (the data for the node including its name sticks with the mount point and not the node itself since the data is not on the disc).

5.3.6 CDDA XML File Node (*struct AppleCDDAXMLFileNode*)
The last of the file nodes is the XML file node structure. It contains information about the fileSize and a pointer to the fileData. Both the fileSize and fileData exist within the mount point as well, so that this node can be recycled and created only when necessary. There is an arbitrary limit on the XML file data size of 40K. It should be vetted for a worst case scenario, including where every track is in its own session.

5.3.7 CDDA Node Info (*struct AppleCDDANodeInfo*)
The CDDA Node Info structure is used for Audio CD tracks only. It contains a back pointer to the vnode associated with this track. It contains the name and nameSize as a C-String, the TOC information for that particular track, the logical block address where the data for that track starts, the number of bytes in that file (including the header which fakes the AIFF-C data), and flags which help to synchronize creation of the vnode associated with the CDDA Node Info.

5.3.8 FinderInfo and ExtendedFinderInfo (*struct FinderInfo*)

For each CDDA Node, the Finder requests properties via the getattrlist() system call. getattrlist() will call CDDA_GetXAttr() to get the extended attributes for the node in question. For each node, CDDA Filesystem ensure that it sets the fileType and fileCreator to the values passed in during the mount process (typcially those of iTunes). Additionally, the FinderInfo is filled in with information to tell the Finder where to place the files (for now we fill in the location to be -1 vertical and -1 horizontal which tells the Finder to perform its own layout each time the disc is inserted and the Finder window is opened and set to that volume). Additionally, the FinderInfo is filled out make the XML file node ".TOC.plist" invisible to the user by setting the invisible bit in the FinderInfo flags and all tracks are set to have their file extension of ".aiff" hidden from the user. These decisions were made by Apple's Human Interface group and shall not change without prior consent of the HIG.

5.4 Diagrams of data structures

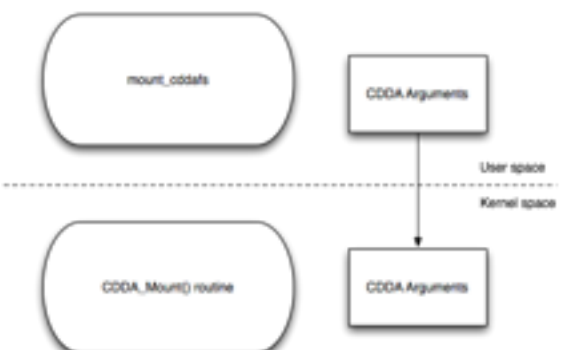

Figure 5-5. Copyin of CDDAArguments structure.

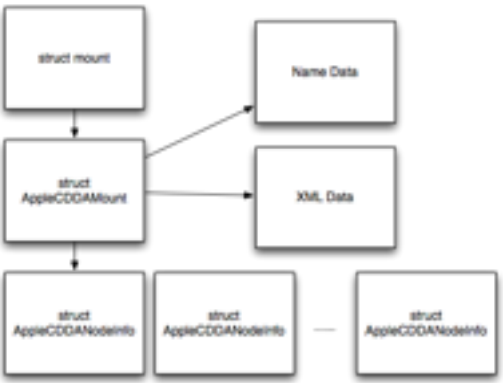

Figure 5-6. Mount point related relationships. *struct AppleCDDAMount* owns the Name Data memory, XML data memory, and array memory for all instances of *struct AppleCDDANodeInfo*. The memory is only freed upon unmount and is therefore shown as owned by the mount-point related struct.

Figure 5-7. Vnode and CDDA Node mappings. Solid lines imply strong ownership between the memory associated with the node and vnode. Dotted lines indicate weak relationships (pointer) to data owned by another entity.

6.0 State machines

The only state machine in CDDA Filesystem is a simple state machine used for reading data in from the Audio CD during the CDDA_Read() or CDDA_PageIn() methods. Figure 6-1 shows the state machine flow for reading sectors from the disc.

Figure 6-1. I/O state machine for CDDA_Read() and CDDA_PageIn().

7.0 Algorithms

7.1 Vnode Lookup Algorithm

Lookup of vnodes is the most common operation in the filesystem. This routine must be very fast.

7.1.1 Validate name

Lookup first checks for the easy cases, '.', '..', and '.TOC.plist'. If the file being looked up is not one of these two, then lookup sanity checks the file name to ensure it ends in .aiff.

7.1.2 Call CDDA_VGetInternal()

Once it validates the file name, it looks at the first two characters of the name. These two characters are the track number (in ASCII) for the Audio Track. Since CDDA Filesystem controls the track names, it specifies that these are always the first two characters in the file name. CDDA Filesystem then adds the offset to get

the track number into the file ID representation (i.e. it adds 100) and calls the CDDA_VGetInternal() routine.
If the file is one of '.' or ".TOC.plist", CDDA_Lookup() calls CDDA_VGetInternal() with the appropriate fileID (2 for the '.' directory and 3 for ".TOC.plist").

7.1.3 Algorithmic Analysis

The complexity of this algorithm is dependent on the complexity of the CDDA_VGetInternal() routine.

7.2 CDDA_VGet() algorithm

Lookup of vnodes using a persistent file ID is the second most common operation in the filesystem. This routine must also be very fast.

7.2.1 Validate File ID

VGet first checks the file ID to validate that it is within range. Only fileIDs 2, 3, and 101-199 are valid file IDs.

7.2.2 Call CDDA_VGetInternal()

Once the file ID has been validated, it can be passed as an argument to CDDA_VGetInternal(). There is nothing else this routine does.

7.2.3 Algorithmic Analysis

The complexity of this algorithm is dependent on the complexity of the CDDA_VGetInternal() routine.

7.3. CDDA_VGetInternal() algorithm

CDDA_VGetInternal() is the clearing house for all lookups, whether they are performed via persistent fileID or by name.

7.3.1 Check for Root

CDDA_VGetInternal() checks for which fileID (ino) is being requested. If the root file ID (2) is asked for, the filesystem already contains a pointer to the root vnode in the mount structure, and it contains a refcount on the node, so it knows the vnode is always present. It simply grabs an iocount on the vnode using vnode_getwithvid() and returns that vnode to the caller with the iocount established, or it returns an error.

7.3.2 Check for XML file

If the CDDA_VGetInternal() routine gets fileID 3 requested (XML File node), it first grabs the mount lock which protects the AppleCDDAMount data structure. It

then checks to see if another thread is busy looking up the vnode. If it is, it blocks, waiting for a wakeup() from the other thread performing work. Once it is guaranteed to be the only thread looking at this fileID, it checks if the mount data structure has an XML file vnode already created. If it does, it calls vnode_vid() to get the current vnode ID, drops the lock on the mount structure, and calls vnode_getwithvid() to grab an iocount on the vnode. If there is no error, that vnode is returned to the caller. Otherwise, the error from vnode_getwithvid() is returned. vnode_getwithvid() may return an error if the vnode is being recycled or reclaimed by the VFS layer.

If the XML file vnode does not exist in the mount structure, the lock is released, and the new XML file node is created. Since vnodes are created with an iocount, if there was no error during creation, the vnode can be returned to the caller. Otherwise, the error from vnode creation is returned to the caller.

Whether or not the vnode was created or simply an iocount was taken, and whether or not an error occurred, this code must issue a wakeup() to any threads waiting for the XML nodeID.

7.3.3 Check for CD Audio Track
If the CDDA_VGetInternal() routine gets a fileID between 101-199 inclusive, it must translate the fileID to the track number (by subtracting 100), and then it must grab the mount structure lock. It then must check the NodeInfo Array created at mount time and scan it for the entry with the track number which matches the entry. Since CDDA Filesystem **cannot** assume audio tracks will be in sequence without gaps (i.e. non-contiguous track numbers), it must search the entire space for the correctly matching track entry in the AppleCDDANodeInfo array.

If no entry is found, ENOENT is returned, otherwise the routine checks to see if a vnode has been created for the matching entry. If a vnode is there, it releases the lock on the NodeInfo Array and calls vnode_getwithvid() to take a refcount on the vnode before passing the vnode back to the caller. If the vnode is not there, it creates a new vnode to represent the file and returns the vnode to the caller. The same rules apply as with the XML file node, meaning that if another thread is looking for the same fileID, it must sleep and wait for a wakeup. Also, when the thread has finished acquiring an iocount or creating the new vnode, it must wakeup any waiting threads.

7.3.4 Algorithmic Analysis
Analysis of this algorithm shows that in the average and worst case, the entire AppleCDDANodeInfo array must be traversed in order to find the vnode

associated with the fileID/file name. This means the algorithm is linear in nature and would be represented in big-O notation as O(n).

7.3.5 Algorithmic Enhancements

7.3.5.1 Hash Table

Note that it is possible to make lookup O(1) by implementing a hashing algorithm which takes the nodeID and/or track name into account, but it was decided that since N is bounded to at most 99 tracks and a hash table would occupy much more memory, the tradeoff for lookup performance did not justify an improvement to the lookup algorithm in this way.

7.3.5.2 Binary Tree

Note it is possible to make lookup O(log(n)) by using a binary tree for searching for the nodeID. Again, because N is bounded to at most 99 tracks and a hash table would be better performing, this algorithm would probably not be used.

7.3.6 Additional Notes

An additional point to be made about the CDDA_VGetInternal() routine is that it is extremely easy to follow and very rudimentary in nature. The CDDA Filesystem is a flat filesystem (i.e. no directories other than the root one). This makes the code in cddafs much more simple than other filesystems which have to worry about the way in which locks are taken and released in their lookup code while traversing complex directory hierarchies and it means the filesystem doesn't need to worry about hard links or soft links.

7.4 CD Table Of Contents Parsing Algorithm

Since the CDDA FileSystem has to represent every possible audio track on a compact disc, an efficient parsing algorithm became necessary. CDs can have very strange layouts depending on how they were mastered, so some have multiple sessions, some have data in pre-gap areas, and some just have data in between audio tracks. This makes it very important to parse the information in the CD's Table of Contents correctly and quickly. The CDDA FileSystem uses the Table of Contents format defined in SFF-8020i ATA Packet Interface for CD-ROMs revision 2.6, January 22, 1996.

7.4.1 Get TOC Data

Grab the TOC Data from the IORegistry. Calculate the number of track descriptors by subtracting the size of the first and last session info (two bytes) and dividing that by the size of a track descriptor (11 bytes).

7.4.2 Loop over track descriptors

Loop over the number of track descriptors calculated in §7.4.1.

7.4.2.1 Check for A2h
Check to see if the point field in the descriptor is 0xA2 (the session leadout). If so, save the currentOffset (which track descriptor 0,1,2,...) in the currentA2Offset.

7.4.2.2 Check for Audio Track
Check to see if the track is an audio track by verifying that the point field is between 1 and 99 (inclusive) and that the control field doesn't have the digital data bit set (use a mask on the hex value 0x04). If it is an audio track, build a track name based on the number in the point field, build a new vnode and associated cddaNode and fill in the cddaNode structure. The fields to fill in are the numBytes field which is calculated from the current track descriptor and the proceeding one (if the proceeding one is not beyond the end of the disc - if it is, use the currentA2Offset which was stored above). Build an AIFF Header for the file, set the vnode type and the cddaNode type and calculate the frame offset at which the track starts (use the track descriptor's PMSF values).

7.4.2.3 Advance to next descriptor
Increment the currentOffset and decrement the number of descriptors left to parse and start at §7.4.2.1

7.4.3 Algorithmic Analysis
Analysis of this algorithm shows that both its average and worst cases increase with the number of tracks and sessions. This means the algorithm is linear in nature and would be represented in big-O notation as O(n). It is bounded by the largest amount of TOC data available (99 sessions each with 1 track). This algorithm cannot be made any faster than it is.

8.0 Synchronizing and Threading

8.1 Synchronizing
Since the CDDA Filesystem is a flat filesystem, little synchronization is required. Data either exists in memory or directly on the disc, so no data requires any synchronization. Synchronization is required to protect the mount point data structures and to ensure that vnode creation and reclamation occurs in a safe fashion. There is only one lock per filesystem mount point which covers all synchronization needs. If lock contention were a major concern, the locking could be broken down into smaller locks, but it would add risk and was deemed unnecessary for the type of operations being performed. Operations that require the lock include:

- CDDA_VGetInternal() which is called from CDDA_VGet(), CDDA_Root(), and CDDA_Lookup()

- CDDA_Reclaim()

Lookup and VGet can race against Reclaim, so the lock is required to ensure that the correct vnode is returned to Lookup and VGet. This may mean that Reclaim is called and frees a vnode, only to turn around and recreate the same vnode. That is OK, and likely happens to other filesystems as well. VFS manages an LRU cache of inactive vnodes and makes the policy decisions. CDDA Filesystem only needs to code the implementation to follow these policy decisions. If engineers working on the filesystem notice pathological cases where this occurs frequently, attention should be brought to this at the VFS policy level.

Synchronization using the lock is used in another case other than Lookup/VGet racing against Reclaim. Multiple threads in the system can race to Lookup or VGet the same node. In this case, the synchronization prevents multiple vnodes from being created and allows only one thread to create the vnode that is required and allows the other to simply grab an iocount on it.

8.2 Threading

There is no formal threading policy for CDDA Filesystem. It doesn't require any kernel threads to do background work, so all requests are made on client threads. Client threads include applications, utilities, or daemons making system calls to perform I/O, look up a vnode, or get attributes of a node. The system does have kernel threads which can act like client threads, such as for VM pageout operations or for cleaning buffers in the background. However, since CDDA Filesystem is a read-only filesystem, neither of these cases should occur. The multiple threads entering the filesystem concurrently means synchronization is required to ensure the proper data and meta-data is returned to the caller. See §8.1 for more information about synchronization.

9.0 Debugging and Profiling

9.1 Debugging

Debugging a filesystem can prove to be a very tricky task. A lot of early debugging (circa 1999) used printf because there was no other easy way to debug a filesystem. Since that time, the kernel trace buffer has evolved, FireWire kprintf has become available, and gdb support for kernel extensions has evolved to the point where breakpoints can be set and used to step through code.

Most of CDDA Filesystem has been well debugged by now and is in maintenance mode, so not a whole lot of effort has been placed into updating the debugging infrastructure. There are ways to enable debug printf for most entry points into the filesystem. It would

be great to have additional things like kernel tracepoints, but that is left to a future release.

There are some handy kernel GDB macros available in the kgmacros file that can help someone developing a filesystem. For instance, there are:

showallmounts
showallvnodes
showallvols
showvnode
showvnodepath

9.2 Profiling

Some rudimentary profiling can be achieved with 'fs_usage'. No other profiling information is provided by the filesystem. Performance has been deemed acceptable since iTunes playback, QT playback, QL playback, and copying performance all seem to work as expected. Further work is required to provide a full profiling infrastructure. Kernel tracing would be appropriate for such work.

10.0 Constraints

CDDA Filesystem is limited to reading CDDA tracks from CDs. It does not work on DVDs or BDs as DVD and BD media are authored with the UDF filesystem. Additionally, Super Audio CDs (SACDs) content is not copied in full by the filesystem. A special player is required to extract the subchannel information from SACDs in order to get 20-bit fidelity (instead of 16-bit).