ORIG

DRAFT:
The Story of the Write Once File System

Simson L. Garfinkel
IRIS
Brown University

Abstract

This papers describes two file systems which were designed for use with write-once media: The Compact Disk File System (CDFS) and the Write Once File System (WOFS). CDFS was designed at the MIT Media Laboratory during the summer of 1985 and was successfully used to master five CDROM disks. Although CDFS was intended for use with write-once media, several design flaws prevented this possibility. The Write Once File System, designed during the summer of 1987, corrects the flaws of CDFS.

This paper also describes an application program written using CDFS which exports CDFS file systems to remote computers via a high speed network and Sun Microsystem's Network File System (NFS) protocol.

This paper is divided into chapters. Each chapter designs a particular project involving one of the above file systems. For the purpose of reprinting, irrelevant chapters may be omitted.

The research described in this paper performed at the MIT Media Lab was made possible by a grant from IBM.

J. Spencer Love, formally of MIT Information Systems, contributed significantly to the design of CDFS.

1 Introduction

In Spring 1985, I was employed as an undergraduate researcher at the MIT Media Laboratory's Electronic Publishing Group.[1] That spring, the laboratory received a Sony CDU-1 prototype CDROM reader. Walter Bender, my research advisor, asked me if I would like to experiment with the unit.

For many years, the Electronic Publishing Group (formally called the Architecture Machine Group) had been interested in optical storage devices, in particular optical video disks, both because of the large amount of available storage and the relatively low access times when compared with other media, such as videotape.[2]

The CDU-1's minimal documentation indicated that the drive could read 2K blocks from a "CDROM" disk, each which was identified by a minute, second and block number. We tried an audio disk in the player and discovered that audio

[1] An innovative program at MIT, "UROP" (undergraduate research opportunities program), allows undergraduates to participate in research projects in the Institute's departments and laboratories.

[2] One of the Group's earlier projects, "Aspen," allowed a person sitting in front of a touch sensitive monitor to "drive" a simulated car around the streets of Aspen, Colorado. To perform the demo, over 20,000 different views of the city had to be stored on a video disk. This was one of the first uses of interactive video disks. Other projects by the group included the use of write-once video disks in animation.

disks were not in the CDROM format. The test disk which Sony had been provided contained approximately 10MB of test patterns on it; the rest of the disk was blank. "I wish they had asked me what to put on the rest of the disk," a graduate student said to me. "I would have given them pictures."

We soon realized that in order to use our new CDROM player, we were going to have to make our own disk. The next question was how to arrange the information on the disk. In Spring 1985, neither Digital's UNIFILE CDROM format nor the High Sierra standard existed. Such standards were under development at the time, but we were largely ignorant of such efforts. Without pre-defined standards, the author decided to develop his own.

A simple way for us to have used our CDROM player would have been to master a CDROM that contained an exact, block-for-block disk image of an existing file system. With such a disk and a block level device driver, we could read the disk as a read-only file system on the same computer from which the data was originally created. Other computers could read the disk by using a suitable conversion library or through some sort of operating system independent network file system such as Sun Microsystem's NFS.

The problem with the disk block image approach was that it would not be extensible to write-once optical devices as they became available, and the group's experience with video disks led us to believe that write-once devices would soon be available. Although applications such as encyclopedias and maps would work fine with large read-only databases, other projects which we were interested in, such as personalized newspapers, would not. "It's much more interesting to think about what you can do with a 300MB database that is constantly growing," I told Dr. Bender, "than to think about all the ways to access 300MB of static data. Another application I was interested by was the possibility of using a write-once optical disk system as a personal, portable mass storage system for coherent use with the wide variety of computers available at MIT.

Over the next six months, under the supervision of Dr. Bender, I developed a file system for use with Write Once Media. At first, the effort consisted largely of late-night design meetings with J. Spencer Love, then a system programmer at MIT Information Systems, trying to devise a way to efficiently store and retrieve information from a write-once media. By the middle of the summer I had started on the file system's first implementation, which first became operational on August 15, 1985.

Since write-once drives and media were not available to use at the time, I developed the file system using a write-once simulator. The simulator presented the file system implementation with the appearance of a write-once device using a file resident on a magnetic disk. When we started mastering CDROMs later that year, we discovered that the simulator system doubled as an excellent mastering system: the simulator file, copied to a nine-track magnetic tape in *ANSI* format, was *all* that was required by 3M corporation to master our CDROMs.

1.1 Design goals

Our two primary goals in designing the file system were:

1. That the file system operate with the same level of performance with any size of write-once media, whether it be a 50MB optical card, a 650MB write-once CDROM, or a 10GB jukebox or optical platters, and that this performance be comparable to performance obtained with magnetic media.

   -and-

2. That performance of the file system would not degrade as a disk became filled with files and directories.

Since we didn't have a write-once device during the design process, we decided that we had

to make as few assumptions about the physical nature of write-once media as possible. The only assumptions that we made were that sequential blocks of data could be written to the write-once device, that the block size used by the device would be a constant from the first block to the last (although no specific block size was assumed), that blocks once written could be read in any order, and that the write-once hardware could determine whether a block had been written or was virginal.

We did not assume that blocks could be invalidated or destructively written after the initial write operation. We further did not assume that a media would be consistently mounted on the same operating system or computer: that is, we wanted a person using our file system to be able to freely move an optical platter from one computer to another, even if the two computers used different operating systems.

This first version of the write-once file system was finished in September 1985 and was called the Compact Disk File System (CDFS), it's name indicating that it was designed primarily for use with compact disks (In 1985, the author had foolishly hoped that hardware vendors would be supplying consumers with devices which would be able to record in the audio CD and CDROM format.) The implementation was written in portable C code and operated without modification on Digital Equipment Corporation's VAX series of computers under UNIX, Sun Microsystem's workstations, and on IBM Personal Computers.

During the following year and a half, CDFS was used to master four CDROMs at the MIT Media Laboratory and one CDROM at Brown University's IRIS project.[3]

In December 1986 and January 1987, I developed a read-only CDFS implementation, the Micro-CDFS, to prove to Dr. Bender that it could be written in less than 10K of object code, hence the name. (The actual MCDFS implementation was less than 5K of object code and provided complete emulation for all of the UNIX system calls to access files in a read-only fashion.)

In the Spring of 1987, several research groups at MIT acquired write once devices and an initial attempt was made to use the CDFS. At this time, the author discovered that a basic assumption made by the CDFS implementation—that write operations to the optical media were completely reliable—was invalid. Design of a new version of CDFS (later named WOFS (Write Once File System)) with the assistance of James Anderson of MIT's Project Athena was commenced at that time, but final implementation was delayed until the author finished his undergraduate thesis[4] and graduated from MIT.

In June 1987, I was employed by the IRIS project at Brown University to write a program to allow the CDROM mastered in the CDFS format to be accessed via Sun Microsystem's Network File System. Because the project was designed to access CDROMs, the program (CDFSD – Compact Disk File System Daemon) only implemented the NFS procedures necessary for read operations. The read-only NFS server can be operated on any BSD 4.2 (or later) UNIX computer which supports sun's portmapper RPC protocol and has a device driver capable of reading blocks from a CDROM player. It is currently in use with an IBM RT/PC workstation.

In July and August 1987, the author completed the development of WOFS.

[3] The four MIT CDROMs consisted of an initial test disk, a disk containing a copy of the CIA's World Databank II and associated information, and two disks of encoded motion sequences for the group's "Movie's of the Future" project. The IRIS disk consisted of the Thesaurus Linguae Graecae Data Bank (all remaining greek documents from the ancient world) and associated index files.

[4] *Radio Research, McCarthyism, and Paul F. Lazarsfeld*, Simson L. Garfinkel, undergraduate thesis, Massachusetts Institute of Technology, June 1987.

2 The Compact Disk File System

2.1 How CDFS works

CDFS organizes write operations to the media as a stream of sequential block writes. No blank blocks are left in the anticipation of future block writes. CDFS' approach divides a disk into two discrete regions: one in which blocks have been recorded and one in which they have not. The file system implementation keeps record of where the dividing point between these two regions is by locating the last written block and maintaining that location during the course of all operations.

When the implementation mounts an optical disk, it first reads the first block on the disk which contains a special block called an End Of Transaction (EOT) block. The EOT contains, in a fixed byte-order representation, an 8 byte flag which identifies the disk as a CDFS-format disk and contains other important information. (The full CDFS EOT specification is presented in Appendix A.1)

The End Of Transaction block is so called because it is the last block written to the disk when the disk is dismounted. The disk blocks between successive EOTs record all changes made to the file system resident on the disk for a particular use session (called transactions).

After locating the first EOT on the disk, the implementation next locates the last written block on the disk[5] which should be another EOT, the last EOT written to the disk.

The EOT belongs to a class of objects called *file system structures*. File system structures are blocks of data, recorded by the file system implementation on the disk, which are decoded by the file system implementation and used by the implementation to locate and retrieve data which had been previously stored. Other file system structures include Directories, File Headers and Directory Lists.

File system structures contain pointers to other file systems structures on the same media. These pointers are called `cdblock` pointers, and reference a particular block number and offset within that block. By allowing `cdblock` pointers to reference exact bytes, the CDFS specification allows multiple file system structures to be packed within the same logical block, although this possibility was not exploited in our first implementation.

The most recent EOT on the disk contains a pointer to the most recent Directory List on the disk. The Directory List contains an array in which each element of the array identifies a directory on the disk and has a `cdblock` pointer to the most recent version of that directory. (The full CDFS Directory List specification is presented in appendix A.2.) Similarly, each directory on the disk contains a `cdblock` pointer to the most recent version of every file it contains.

The Directory List is the key to an efficient implementation of a modifiable hierarchical file system on a write-once disk. Since CDFS stores both files and directories on the same write-once media, it is necessary to rewrite a directory whenever a file contained within it is added, deleted, or modified. The Directory list eliminates the necessity of having to additionally rewrite all of the containing directories. Since the Directory List contains a `cdblock` pointer to the most recent version of each directory on the disk, directories are located *via* the directory list, rather than from pointers in their containing directories. When the location of a directory on the disk changes (as happens when the directory's contents are changed), it is necessary only to rewrite the Directory List to still retain a pointer directly to the most recent version of the directory. Without a Directory List, a modification to any sub-directory within the file hierarchy would require rewriting every directory above the modification, including the root directory, which would greatly increase the overhead of the file system. The Directory List is

[5] Locating the last written block on the disk is accomplished by a binary search across the media.

designed to be compact to further decrease storage requirements.

In practice, write operations to the optical disk are batched in groups called Transactions. During a Transaction, new files are written to the disk as they are created, but changes to their containing directories and the Directory List are buffered in memory. At the close of the Transaction, all modified directories, the Directory List, and a new EOT are written and the disk can be removed from the drive. In the event of an interrupted Transaction (such as a power failure or removal of the disk from the drive before dismounting), the File Header which is written with each file contains enough information to reconstruct the directories, Directory List, and EOT which had not yet been written.

2.2 Two Example Transactions

Although the CDFS standard does not specify the order in which Transactions should occurs, most implementations will follow a process similar to example outlined bellow.

In this example, the write-once media starts blank. In the first Transaction, the two files, *life.c* and *wheel.c* are written and placed in the root directory. In the second Transaction, a new version of the file *life.c* is written. This example Transaction is intentionally simple.

As defined in the standard, the first block on the disk must be an EOT. During the course of the first transaction, the two files are written to the disk. When the transaction is completed, the CDFS implementation writes a new copy of the root directory to the disk, a new Directory List, and a new EOT. The EOT contains a pointer to the Directory List, which contains a pointer to the root directory, which contains pointers to each of the files. The state of the disk at the completion of the first transaction is depicted bellow:

The following blocks were written on the first transaction:

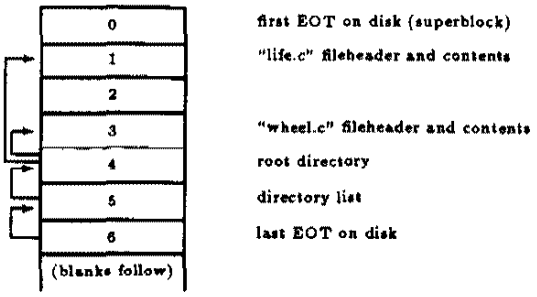

Figure 1: Direct Read After Write disk after a sample transaction

0 The first block of a CDFS disk contains an EOT, which identifies the disk as a CDFS disk. This block also contains the name of the disk's owner, the time of the disk's creation, the name of the site which created the CD and other interesting information.

1 The file *life.c* is stored contiguously, preceded by its File Header. Since the File Header contains a `cdblock` pointer to the contents of the file, the File Header could equally well be written after the file. The file begins in the same block as the header and extends for two blocks.

3 The file *wheel.c* is the second file on the CD, stored as file header followed by file contents. The file and its contents fit within the single block.

4 The root directory follows. The directory contains a `cdblock` pointer to each file within it. `cdblock` pointers are drawn as arrows in this example.

5 The Directory List follows the directories. It contains a `cdblock` pointer to each directory on the disk.

6 An EOT is the last block written to the CD. It contains a `cdblock` pointer to the Directory List.

The section transaction takes place independently of the first. At the start of the

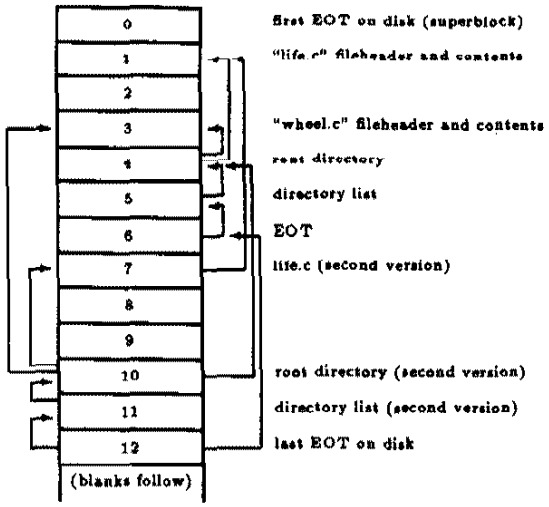

Figure 2: Direct Read After Write disk after a second sample transaction. Arrows on the left hand side trace pointers from the EOT to the most current version of each file. Arrows on the right hand side trace pointers to previous versions.

second transaction, the implementation reads the first and the last blocks on the disk. A new version of the file *life.c* is written to the disk and the Transaction is ended, which causes a new root directory, Directory List and EOT to be written. After the second transaction a schematic view of the optical media would looks like figure 2.

While no blocks on the disk have been changed, the last EOT commences a chain of pointers which points now to the newest version of each file, because the definition of "the last EOT" has changed.

The blocks that were written on the second transaction include:

7 The first file written on the second transaction is the updated version of *life.c*. The new file header of *life.c* contains a pointer to the previous version (shown on the right). The new version of *life.c* is three blocks long.

10 The new version of the root directory contains `cdblock` pointers to the most current version of each file within it. Note that directories can (and often do) reference files which were written on previous Transactions.

11 The Directory List is written after the root directory. It contains a pointer to the most recent version of that directory.

12 The EOT block is written last. It contains a pointer to the most recent Directory List. It also contains a pointer to the previous EOT.

2.3 Design choices made in CDFS

The design choices made in developing the CDFS were based both upon the desire for the file system to efficient and for the file system to be usable in an environment of heterogeneous operating systems, hardware vendors and administrative policies. A secondary goal was to provide substantial amounts of redundant information in the filesystem so that information could be easily — even automatically — recovered from damaged disks.

The design of the CDFS file header clearly illustrates how our concerns translated themselves into the file system structures. CDFS file headers average over 240 bytes in length – substantially larger than the equivalent structures in other operation systems (for example, UNIX inodes), but still very small when compared to the average size of files on an average computer or when compared to the amount of space available on optical media. CDFS uses the space to store on a per file basis information which traditional operating systems store once per magnetic disk. (The specification for the CDFS file header is given in appendix A.6).

For example, the CDFS file header includes the full user name of the person who created the file (rather than merely storing the user's "number," as the UNIX file system does, or storing nothing at all, as the MSDOS and Macintosh file systems do, assuming that all files on the same computer are owned by the same user), the site name of the computer which created the file, and

backwards pointers to the previous version of the file and the last EOT on the disk when the file had been written. By storing the site name on a file-by-file basis, CDFS allows the possibility that files from multiple sites may be stored during the course of a single transaction (as would be the case for a networked CDFS file server or archiving service). By storing the full user name, rather than just a number, the standard allow a disk to be moved to a site where the user name to user number mapping is not known without sacrificing legibility of the file list command.

By explicitly assigning a version number and a known length to each file system structure, the CDFS specification allows the possibility for extension to be made within the context of the existing standard. For example, files which are actually links to other files are implemented as different version of the `file_info` structure in the file header.

While data transfer rates to and from optical disks are very high, head repositioning time is very slow. CDFS is designed to minimize the number of head repositioning events necessary to read information from a disk.

2.4 Advantages of CDFS Over Traditional File Systems

The principle advantage of using write-once optical media over magnetic media (besides the increased storage space), is the ability to recover any file or document that was ever stored. Even if a file is updated by a later version or if it is "deleted" from its containing directory, it is in principle always possible to find the original document. CDFS realizes this possibility by providing specific interfaces for locating previous versions of files or for "undeleting" files after they have been deleted.

Since blocks, once written, are never changed, the process of performing an incremental backup of a CDFS disk is simplified considerably over magnetic file systems: the disk's newly written blocks are merely copied, block-for-block, to the backup disk, which after the backup becomes an identical copy of the "working" disk.

Use of CDFS allows the same media standard to be used for archives, backups, working files and transportation of data. A single, unified set of utilities can then be employed for all file operations.

3 CDFS CDROMs and the CDFS/NFS Server

Early in the CDFS development effort, we recognized clear advantages to using the same file system standard for read-only and write-once optical storage devices. Beyond the ability to use the write-once system as a mastering platform for the read-only disks, using the same file system standard allows same software which was used to create the read-only disk to retrieve the information. Another exciting possibility allowed by using the same file system for read-only media as for write-once is that of using write-once media as a publication format, to which a user can add new information (such as personal comments or updates) in a similar manner to the way a user can write in the margins of a book with a felt tip pen. Possibilities such as these led the Electronic Publishing Group to adopt CDFS as our standard for CDROMs in Fall 1985.

After the Electronic Publishing Group made its first CDROM, the author was contacted by Paul D. Kahn at Brown University's IRIS group, requesting help in making the TLG CDROM mentioned above. The disk was pressed in spring 1986 and a variety of application programs were written to use it. The programs accessed the information on the CDROM via the CDFS subroutine library. During the summer of 1987, the author moved to the IRIS project as a temporary systems programmer to write a program to allow the TLG CDROM to be read via the Sun Microsystems Network File System (NFS) protocol.

The program, the Compact Disk File System Daemon (CDFSD), allows a CDFS mastered disk

to be read (from any computer which uses the Sun NFS protocol) as if it was a standard unix file system. Logically, CDFSD incorporates the CDFS subroutine library into the operating system's kernel, allowing the subroutines to be removed from user level programs.

3.1 NFS

NFS is a layered system which allows one computer to access files on a remote computer as if they were mounted locally. NFS is based upon Sun Microsystem's Remote Procedure Call (RPC) library, a system which allows one computer to execute functions on another. RPC is, in turn, based upon Sun's External Data Representation standard (XDR), which allows computers of from different manufactures using different byte ordering systems to exchange all types of data in a byte-order, word-size independent fashion.[6]

In the working NFS system computers are classified as *servers* and *clients*. A server is a computer which maintains a file system locally and makes the files in it available to other computers—the clients—via the network. Although the operating system (the kernel) of the client must be modified to recognize which file operations should be performed locally and which translated into RPC calls, the server can be implemented as a user level program with no modifications to the server's operating system.[7]

[6] Sun was required to write XDR since the computer which their early products were based upon, the Motorola 68010, was a low-byte first byte ordered machines, while other machines owned by Sun's targeted customers were high-byte first ordered machines.

CDFS (and WOFS) implements its own byte-order independent external data representation standard which, while not as comprehensive as Sun Microsystems, is marginally faster in execution. CDFS adopted VAX byte-ordering. WOFS changed the byte-ordering to 68000 byte-ordering. When CDFS mounts a disk, it examines the byte-ordering used on the disk and can reverse its byte-ordering on the fly.

[7] In practice, the NFS server is written into the server's operating system to provide direct access to file system structures (in particular, unix inodes) and to improve performance.

The NFS system actually consists of two protocols: MOUNT and RFS (Remote File System). The MOUNT protocol provides a means for servers to identify to clients what filesystems are candidates for file sharing. MOUNT also allows servers to monitor clients which are using which shared file systems, principally to allow notifications to be sent to clients when servers are about to be withdrawn from service. The MOUNT protocol's only interface with the actual file system is to return upon request to the client a `fhandle` to the root directory of any given file system.

RFS is the protocol in which performs the actual mapping from operating system service calls on the client to RPC calls which are executed on the server. The basis of the RFS protocol is the `fhandle` concept. A `fhandle` is a 32 byte opaque data "cookie" which is passed between a server and a client to identify files and directories used in RPC operations.

When an NFS file system is mounted, the MOUNT program returns to the client a `fhandle` for the server's root directory (or the root directory of a particular file system.) This `fhandle` is then returned to the server, by the client, when other operations, such as read directory, or lookup file in directory, are requested. RFS is *stateless*, in that a `fhandle` uniquely identifies a file or directory even if the NFS server is restarted (for example, after a system crash). If the client waits for the server to restart, it can continue operations as if nothing had ever happened.

3.2 Compact Disk File System Daemon

The Compact Disk File System Daemon (CDFSD) is a user level program which implements the server side of both the MOUNT and RFS protocols. CDFSD accepts and responds to RPC requests using Sun Microsystem's `svc_register()`, `svc_run()`, and `svcsendreply()`+ RPC library functions. Since no provisions are made in their the RPC or the NFS protocols for multiple server processes on a single host (for example, by

assigning multiple program numbers to NFS servers or allowing NFS to use arbitrary UDP port numbers), CDFSD cannot coexist on a machine which is also a unix-filesystem NFS server.

Unlike a unix **fhandle**, which consists of a file system number and an inode number, the CDFSD **fhandle** consists of the following information:

- The file number.
- The containing directory number.
- The drive number.
- The file header location.

Although the CDFSD file number and containing directory number can be derived from the file header location, incorporating this information into the **fhandle** eliminates the necessity of having to read the referenced file's file header before the RPC call can be serviced. The overhead of including this information is negligible to the server and zero to the client.

CDFSD implements the following RFS procedure calls:

RFS_NULL This procedure does nothing and is used for timing.

RFS_GETATTR This procedure returns attributes for a file or directory. It is the RPC implementation of the unix stat() and fstat() calls. The CDFSD implementation necessarily performs translation between CDFS file headers and unix stat buffers (which are used by NFS).

RFS_LOOKUP This procedure performs the translation between file names and **fhandles**. It is the only RFS procedure that creates **fhandles**.

RFS_READ This procedure reads a requested number of bytes in a file from a given starting location. As CDFS files are stored contiguously, this procedure merely counts from the start of the file the requested offset and then copies data directly from the CDROM to the RPC reply buffer.

RFS_READDIR This procedure is used by NFS to read the contents of directories in a operating system independent manner. This procedure necessarily performs the translation between the CDFS and unix directory structure directories, which NFS uses.

While UNIX directories contain only file names and inode numbers, CDFS directories contain additional information (such as file size and last modification date). This additional information was intentionally placed in the directory structure to speed extended directory list commands (e.g. the MSDOS "DIR" command or the UNIX "ls -l" command) by eliminating the need for the optical disk to seek to each file header when a directory list command is executed. Unfortunately, since NFS is basically a translation of the UNIX file system to RPC subroutines, this additional information is not useful to CDFSD.

RFS_STATFS This procedure returns information about the CDFS file system.

3.3 Caching and Performance

CDROM performance (and performance of optical storage systems in general) depends on two factors: how long it takes for the drive's read head to traverse the surface of the media to the correct block on the disk (seek time) and how long it takes to transfer data from the disk to the host computer (transfer time). CDFSD employs two forms of caching to improve subjective performance: block caching for blocks containing file system structures and read-ahead caching for data transfers.

All CDFSD procedures call the function **cd_read_** to transfer data from the CDROM to the computer. The third parameter of the **cd_read_** function is an integer flag which is used to differentiate reads of file system blocks from reads

of user data. `cd_read_` maintains a 300 block associative cache for all read operations of file system structures. This cache dramatically improves performance of file system operations such as "open file," and "read directory," by changing them from disk-to-memory transactions to memory-to-memory transactions. This cache is invisible to the procedure calling `cd_read_` .

When `cd_read_` is forced to read a block of data from the disk, it reads several blocks at a time into a read ahead cache. If successive read requests are for a block in this read ahead cache, `cd_read_` does not have to perform another physical read. This cache dramatically improves performance of reading user data from files, since the time required to read two blocks from the CDROM is negligibly more than the time required to read one block. This cache is also invisible to the procedure calling `cd_read_` .

In comparison between the CDFSD and the standard NFS daemon operating from a magnetic file system, we find that many operations involving small files are marginally faster with the CDROM (definitely a result of the block caching) and sustained read operations involving large files are marginally slower with the CDROM (owing to the slower transfer rate from CDROM to computer, when compared with magnetic.)

4 The Write Once File System

When MIT received its first shipment of optical drives and media in Fall 1986, the author discovered a serious flaw in the design of the CDFS implementation: no provision was made for failed writes on the optical media. Although the CDFS standard was insensitive to this oversight, the MIT Media Laboratory's implementation depended on disk write operations being perfectly reliable.

During the summer of 1987 the CDFS implementation was completely redesigned to allow for failed media writes. At the same time, the CDFS standard was substantially revised based on the accumulated knowledge over the previous two years. These changes substantially reduced the complexity of the file system implementation. In recognition of the substantial changes, the file system was renamed the Write Once File System.

4.1 WOFS changes to the CDFS standard

The principle difference between the WOFS and the CDFS is the adoption of a new kind of file system structure called a "file system block," (fsblock for short). An fsblock is a 16-byte header, containing a flag, a self-referential pointer, a type identifier and a chain count (see appendix B.2) which identifies a block on the optical media as a block in which file system structures are stored. By moving all CDFS self-referential pointers to a new layer procedure layer between the hardware device driver and the CDFS subroutine library, CDFS was greatly simplified. Other changes made to the standard include:

- Giving explicit lengths to all variable length file system structures.
- Inclusion of the directory name into each directory list element of the directory list to allow resolution of path names without requiring each directory's file header and directory contents to be loaded.
- Lengthening of the maximum file name size from 48 bytes to 80.
- The adoption of a new file type—Log files—which allows low-overhead append-only logs to be maintained.
- The implementation of fragmented files.

5 Related Work

Other proposal for the use of write-once media (e.g. Easton 1985) assumed that write-once hardware

would have the ability to invalidate information previously recorded on the surface or to fill in fields previously left empty. Time has shown these assumptions to be largely incorrect, due to the nature of the error correction codes employed in the low level driver hardware.

Nearly all other proposals for the use of write-once media involve storing directory information on some form of rewritable magnetic storage system. The Amoeba file server [Mullender and Tanenbaum, 1985] is an example systems of this type.

A significant excpetion is the recent work on log files at Stanford University [Cheriton and Finlayson, 1986]. Like the write once file system, the Cheriton and Finlayson's log files exploit the characteristics of write once devices rather than attempting to hid them from the user. The service they describes "provides efficient storage and retrieval of data that is written sequentially (append-only) and not subsequently modified." This paper was the inspiration behind the WOFS log file facility.

6 References

"Working Paper for Information Processing: Volume and File Structure of CD-ROM for Information Interchange," prepared as a working paper of the CD-ROM Ad Hoc Advisory Committee, popularly known as the High Sierra Group, *Optical Information Systems*, January–February 1987.

Log Files: An Extended File Service Exploiting Write-Once Optical Disk, David R. Rheriton and Ross S. Finlayson, Stanford University, 1986.

"Thermo-Magneto-Optical Disk Promises High-Capacity, Low-Cost Removable Storage," *Digital Design*, August 1985.

key-Sequence Data Sets on Indelible Storage, M. C. Easton, Technical Report RJ 4776 (50637), IBM Research Laboratory, San Jose, California, July 1985.

Dynamic Linking in a Small Address Space, M. L. Kazar, S. B. Thesis, Department of Electrical Engineering and Computer Science, MIT, May 1978.

"A distributed file service based on optimistic concurrency control," S. Mullender and A. Tanenbaum, *Proceedings of the ACM Symposium on Operating System Principles*, 15–62, December 1985. The paper that describes the Amoeba file server.

"A Fast File System for Unix," M. K. McKusick, W. N. Joy, S. J. Leffler and R. S. Fabry, Berkeley Unix documentation, version 4.2, 1983.

The Multics System: An Examination of its Structure, Elliott I. Organick, MIT Press, 1972, pages 217–234.

"Compact Disc Digital Audio Systems," David Ranada, *Computers and Electronics*, August 1983.

"Networking on the Sun Workstation," Sun Microsystems, Mountain View, CA 94043. 1985

A CDFS Specification

A.1 End Of Transaction (EOT) format

```
typedef struct {
        int32   modulo_of_value;
        int16   bits_in_value;
        int16   pad;
} pointerdef;

#define EOT_ID_STRING_LEN 8
#define EOT_ID_STRING "\237\002\CDFS\255\000"
#define EOT_VERSION 1
#define CDFS_IMPLEMENTATION_ID 1

typedef struct {
        char    id_string[ EOT_ID_STRING_LEN ];

        int16   eot_version;
        int16   eot_length;

        cdblock eot_location;

        int16   eot_checksum;
        int16   CDFS_implementation_id;

        cdblock current_dir_list;
        cdblock previous_eot_location;
        cdblock next_eot_location;

        int64   filesystem_creation_time;
        int32   trans_number;
        int64   trans_start_time;
        int64   trans_end_time;
        int32   files_written_on_trans;
        int32   dirs_written_on_trans;
        int32   next_free_file_number;

        pointerdef      pointerdes[16];

        int16   number_of_used_pointerdefs;
        char    encryption_standard[32];
        char    owners_name[ variable ];

} eot_format;
```

A.2 Directory List structures

A.3 Directory List Header format

```
#define DL_ID_STRING_LEN 8
#define DL_ID_STRING "\237\001CDFS\250\000"
#define DIR_LIST_VERSION 1

typedef struct {
        char    id_string[ DL_ID_STRING_LEN ];

        int16   dir_list_version;
        int16   dir_list_header_length;
        cdblock dir_list_loc;

        int16   dir_list_checksum;
        int16   pad;

        cdblock prev_dir_list;

        int32   dir_list_entry_count;
} dir_list;
```

A.4 Directory List Entry format

```
typedef struct {
        int32   dir_number;
        cdblock header_location;
        int32   containing_dir;
        int64   modify_time;
        int64   contained_bytes;
        int16   header_size;
        int16   pad;
} list_element;
```

A.5 Directory Format

```
#define DIRECTORY_INFO_VERSION_ 1
typedef struct {
        int32   directory_info_version;
        int32   directory_info_length;

        int32   directory_entries;
        int32   directory_entry_size;
} directory_info;

#define MAX_COMP_LEN 48
typedef struct {
        char    file_name[ MAX_COMP_LEN ];
```

```
        cdblock header_location;
        int64   modify_time;
        int32   file_number;
        int32   file_size;
        int32   file_version;
        int16   file_type;
        int16   header_size;
        int16   addname_count;
        int16   pad;
} dir_contents;
```

A.6 File Header Format

```
#define FILE_TYPE       1
#define DIRECTORY_TYPE  2
#define SOFT_LINK_TYPE  3
#define FRAGMENTED_TYPE 4
#define ADDNAME_TYPE    6

#define FH_ID_STRING_LEN 8
#define FH_ID_STRING "\237\001\CDFS\255\000"

#define HEADER_VERSION 1
typedef struct  {
        char    id_string[ FH_ID_STRING_LEN ];
        int16   header_version;
        int16   header_length;
        int16   header_checksum;
        int16   fileheader_length;

        cdblock fileheader_location;

        int32   file_number;
        int16   file_type;

        int16   access_info_offset;
        int16   backup_info_offset;
        int16   file_info_offset;
        int16   site_info_offset;
        int16   property_list_offset;

} fileheader;

#define ACCESS_INFO_VERSION 1
#define GROUPLEN 32
#define OWNERLEN 32

typedef struct {
        int16   access_info_version;
        int16   access_info_length;

        char    file_owner[OWNERLEN];
        char    file_group[GROUPLEN];
        int16   file_access;
} access_info;

#define DOWN_DIR_CHAR 0376
#define UP_DIR_CHAR   0375

#define BACKUP_INFO_VERSION 1
typedef struct  {
        int16   backup_info_version;
        int16   backup_info_length;

        int32   containing_directory_number;
        cdblock previous_version_location;
        cdblock previous_eot_location;
        int16   filename_offset;
        int16   previous_version_header_size;
        char    backup_pathname[ variable ];
} backup_info;


#define FILE_INFO_VERSION 1
typedef struct {
        int16   file_info_version;
        int16   file_info_length;

        cdblock file_location;
        int32   file_length;
        int64   write_time;
        int64   creation_time;
        int32   file_version_number;
} file_info;


/* If block is a soft link, use soft_link_info
 * to decode file_info */

#define SOFT_LINK_VERSION 1
typedef struct {
        int16   soft_link_info_version;
        int16   soft_link_info_length;

        int64   creation_time;
        int32   target_dir;
        int32   target_version;
        char    target_name[ variable ];
} soft_link_info;

#define SITE_INFO_VERSION 1
typedef struct {
        int16   site_info_version;
        int16   site_info_length;

        char    opsys[16];
```

```
        char    opsys_version[16];
        char    site_name[ variable ];
} site_info;

#define PROPERTY_LIST_VERSION 1
typedef struct {
        int32   property_list_version;
        int16   property_list_length;

        int16   property_list_entries;

} property_list_info;

typedef struct {
        int16   property_name_len;
        int16   property_value_len;
        char    property_name[ variable ];
        char    property_value[ variable ];
} property_list_record;
```

A.7 Fragmented files filemap

```
#define STRIP_INFO_VERSION 1
typedef struct {
        int32   strip_info_version;
        int32   strip_info_length;

        int32   strip_count;
} strip_info;

typedef struct {
        cdblock loc;
        int32   valid_chars;
        int32   ordinal;
} fragmented_des;
```

A.8 Directory Format

```
#define DIRECTORY_INFO_VERSION_ 1
typedef struct {
        int32   directory_info_version;
        int32   directory_info_length;

        int32   directory_entries;
        int32   directory_entry_size;
} directory_info;

#define MAX_COMP_LEN 48
typedef struct {
        char    file_name[ MAX_COMP_LEN ];
        cdblock header_location;
        int64   modify_time;
        int32   file_number;
        int32   file_size;
        int32   file_version;
        int16   file_type;
        int16   header_size;
        int16   addname_count;
        int16   pad;
} dir_contents;
```

B WOFS Specification

B.1 Time and wblock structures

```
typedef struct {
        u_long  high;
        u_long  low;
} int64;

typedef int64 wtime;

typedef struct {
        u_long  block;
        u_short offset;
        u_short pad;
} wblock;
```

B.2 File System Block

```
#define FS_BLOCK_FLAG "\237\200\005\000"

#define FS_EOT_TYPE 1
#define FS_DL_TYPE  2
#define FS_FH_TYPE  3
#define FS_DIR_TYPE 4

#define FS_HEADER_SIZE (4 + sizeof(wblock)
        + 2 * sizeof(u_short))

#define FS_DATA_SIZE \
BLOCK_SIZE - FS_HEADER_SIZE

typedef struct {
```

```
        char    flag[4];
        wblock  loc;            /* self  */
        u_short type;
        u_short chain_count;
        char    data[ FS_DATA_SIZE ];
} fs_block;
```

B.3 Flags and other definitions

```
#define EOT_FLAG        1
#define DL_FLAG         2
#define DI_FLAG         3
#define FH_FLAG         4
```

B.4 End of Transaction

```
#define EOT_VERSION 2
typedef struct {
        u_long  flag;
        u_short eot_version;
        u_short eot_length;

        wblock  current_dir_list;
        u_long  prev_eot_loc;
        u_long  next_eot_loc;

        wtime   filesystem_creation_time;

        u_long  trans_number;
        wtime   trans_start_time;
        wtime   trans_end_time;

        u_long  files_written_on_trans;
        u_long  dirs_written_on_trans;
        u_long  next_free_file_number;

        char    volume_name[32];
        char    encryption_standard[32];
        char    owners_name
   [FS_DATA_SIZE-128];

} eot_format ;
```

B.5 Directory List

```
#define DIR_LIST_VERSION 3
#define MAX_COMP_SIZE 80

typedef struct {
        u_long  flag;
        u_short dir_list_version;
        u_short dir_list_header_length;

        wblock  prev_dir_list_header;

        u_long  dir_list_entry_count;
} dir_list_info;        /* array follows */


typedef struct {
        u_long  dir_number;
        wblock  header_loc;
        u_long  containing_dir;
        wtime   modify_time;
        int64   contained_bytes;
        u_short header_size;
        u_short pad;
char    name[MAX_COMP_SIZE];
} dir_list_element;
```

B.6 Directory format

```
/*
 * A dir is a normal file in which
 * the contents consist of a
 * dir_info header followed by a an
 * array of dir_contents. The
 * length stored in the file_info
 * structure of the file_header is
 * equal to sizeof(dir_info) +
 * dir_entries * dir_entry_size;
 */


#define DIR_INFO_VERSION 2
typedef struct {
        u_long  flag;
        u_long  dir_info_version;
```

```
        u_long  dir_info_length;

        u_long  dir_entries;
        u_long  dir_entry_size;
} dir_info;


#define ROOT_DIR 1

typedef struct {
      char      file_name[MAX_COMP_SIZE];
      wblock    header_loc;
      wtime     modify_time;
      u_long    file_number;
      u_long    file_length;
      u_short   file_type;
      u_short   header_size;
      u_short   addname_count;
      u_short   pad;
} dir_contents;
```

B.7 File Headers

```
/* Define file types */

#define NOTHING_TYPE    0
#define FILE_TYPE       1
#define DIR_TYPE        2
#define LINK_TYPE       3

#define HEADER_VERSION 2
typedef struct  {
        u_long  flag;
        u_short header_version;
        u_short header_length;

        u_long  file_number;
        u_short file_type;

        u_short access_info_offset;
        u_short access_info_length;
        u_short backup_info_offset;
        u_short backup_info_length;
        u_short file_info_offset;
        u_short file_info_length;
        u_short site_info_offset;
        u_short site_info_length;
        u_short property_info_offset;
        u_short property_info_length;

} fileheader;

#define UNIX_ACCESS_INFO_VERSION 1
#define NAME_LEN 32
typedef struct {
        u_long  access_info_version;

        char    file_owner[ NAME_LEN ];
        char    file_group[ NAME_LEN ];
        u_short unix_access;
} access_info ;


/* backup info defines */
#define BACKUP_INFO_VERSION 2
typedef struct  {
        u_long backup_info_version;
        u_long  containing_dir_number;
        wblock  prev_version_loc;
        u_long  prev_eot_loc;
        u_short prev_version_header_size;
        char    filename[MAX_COMP_SIZE];
} backup_info;

#define CONTIG_FILE_INFO_VERSION 1
#define FRAG_FILE_INFO_VERSION 2

typedef struct {
        u_long file_info_version;

        wblock  contents;
        u_long  byte_count;
        wtime   write_time;
        wtime   creation_time;
        u_long  file_version_number;
} file_info ;
typdef file_info frag_file_info;

/* In a fragmented file:
 *  contents    - loc of frag_des array.
 *  byte_count  - size of the array.
 */
```

```
typedef struct {
        wblock  loc;
        u_long  valid_chars;
        u_long  ordinal;
} frag_des;

#define LOG_FILE_INFO_VERSION 3
typedef struct {
u_long  file_info_version;

wblock entry;
u_long entry_bytes;
wtime entry_time;

wblock prev_header;
u_long total_bytes;
wtime creation_time;
} log_file_info:


/* If file_header is a soft link,
 * use soft_link_info_
 * to decode the file_info */

/* As denoted in symbolic links */
#define DOWN_DIR_CHAR 0376
#define UP_DIR_CHAR   0375

#define SOFT_LINK_VERSION 3
typedef struct {
        u_long  soft_link_info_version;

        int64   creation_time;
        u_long  target_dir;
        u_long  target_version;
        char    target_name[ 256 ];
} soft_link_info ;

#define FIRM_LINK_VERSION 4
typedef struct {
        u_long  firm_link_info_version;

        int64   creation_time;
        u_long  target_dir;
        u_long  target_version;
        u_long  target_filenumber;
} firm_link_info ;
```

```
#define SITE_INFO_VERSION 1
typedef struct {
        u_long site_info_version;

        char    opsys[ 16 ];
        char    opsys_v[ 16 ];
        char    site_name[ 64 ];
} site_info ;


#define PROPERTY_LIST_VERSION 1

typedef struct {
        u_long  property_list_version;
        u_long  property_list_length;
        u_long  property_list_entries;
} property_list_info;

#define variable 1
typedef struct {
        u_short property_name_len;
        u_short property_value_len;
        char    property_name[ variable ];
        char    property_value[ variable ];
} property_list_record;
```