A File System For Write-Once Media

Simson L. Garfinkel
MIT Media Lab
Cambridge, MA 02139
Garfinkel@MEDIA-LAB.MIT.EDU

September 25, 1986

Abstract

A file system for use with write-once media, such as digital compact disks is proposed. The file system is designed to be compatible with any operating system and a variety of physical media. Although the file system is simple to implement, it is robust, fast and full-featured. The file system is both operating system independent and site independent, providing a simple means of data exchange between computers.

This research was performed at the MIT Media Lab during the spring and summer of 1985, and was sponsored in part by a grant from IBM.

Acknowledgment

J. Spencer Love, MIT Information Systems, contributed to the design of this file system.

Contents

1 Introduction

The CDROM (Compact Disc Read Only Memory) is an application of compact disc technology to computer mass-storage. A single CDROM can store over 630 MBytes of information on its 5.25" surface. While commonly available CD drives are read-only, CD drives capable of recording on unrecorded blocks are becoming increasingly available. Such devices have been termed write-once CDROMs or DRAW-CDs (Direct Read After Write).

An ongoing research project at the MIT Media Lab has resulted in the development of a hierarchical file system for read-only and write-once CDs.

1.1 Using CDROMs with Existing Operating Systems

CDROMs, because of their high capacity, are ideally suited for storage of large static databases. The simplest of several CDROM storage strategies masters a CDROM with a database and stores pointers and references to the database on a seperate magnetic media. The CDROM and the magnetic media are distributed together as a set. The data stored on CDROMs mastered in this fasion is not accessible directly from the computer's operating system but only from specially written application programs.

Alternatively, a CDROM can be mastered as a block-by-block image of a magnetic disk; the CDROM could then be mounted into an operating system as a read-only file system. CDROMs which are block images of existing file systems can only be read by the operating system from which the image was created. A second drawback of this approach is that it does not easily extend to write-once media.

The Compact Disk File System overcomes these problems. CDFS stores directory information on the disk rather than on a separate multiple-write media. CDFS allows disks mastered under one operating system to be read and written by others, making it operating system independent. CDFS functions with both CDROMs and with write-once media.

2 CDFS Goals

CDFS realizes three goals: The appearance of an arbitrarily modifiable file system on a write once media, operating system independence and site independence.

The proposed use of CDFS is to allow personal files, residing on CD, to be transported and used at a variety of locations on many kinds of computers. Write once media also provide archiving functions, since all previous versions of a person's files would be stored on the same media. Backup procedures are also simplified.

CDFS assumes that sequential blocks of the write once media can be written. No other assumptions are made about the media or hardware. For example, the file system does not leave unrecorded blocks with the intention of later recording them. The file system will operate with an arbitrary media block size or block numbering scheme.

CDFS provides simple mechanisms for recovering previous versions of files or directories, even after files or directories had been "deleted." Automated recovery after media failure is a simple procedure thanks to redundent information stored within the file system.

Data transfer rates to and from optical disks are very fast. However, head repositioning is very slow. CDFS minimizes the number of head repositioning events necessary to read information from a disk, minimizing the impact of slow seek times.

CDFS identifies and incorporates all services provided by conventional file systems. CDFS provides a mechanism for incorporating in a compatible fashion those services which were omitted due to design oversight.

3 Data Structures and Transactions

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 regions: one in which blocks have been recorded, and one in which they have not. The maintenance of a list of free blocks is reduced to determining the first unwritten block on the disk.

3.1 End of Transaction Block

CDFS organizes information on the write-once media in sequences called *Transactions*. Each *Transaction* consists of a group of modified or newly created files, the directories containing those files, a *Directory List* and an *End-Of-Transaction* block (EOT). Batching write operations to the media minimizes the overhead of having to rewrite a directory each time a contained file is modified. Directories need only be written to the disk before the disk is dismounted.

When a disk is not in the process of a transactional update, the last block recorded on the disk is an EOT block. The EOT block contains a pointer to the last written Directory List and statistical information pertinent to the last transaction. (See Appendix B for a list of the fields contained in the EOT.)

Before a disk can be accessed by the CDFS the last EOT on the disk must be located. This process is called *Mounting* a disk. The last EOT can be located quickly with a binary search of the media. The EOT contains a pointer to the most recently written Directory List.

3.2 Directory List

The Directory List is the key to an efficient implementation of a modifiable hierarchical file system on a write-once media. It is written at the end of each transaction but before the EOT. The Directory List consists of a header containing state information and an ordered array containing one entry for each directory in the file system (see Appendix C). Directories and subdirectories are located with reference to the Directory List, rather than with reference to the containing directory. Without a Directory List,

modifications deep within the file hierarchy require rewriting all containing directories, including the root directory, at the close of the transaction. Most CDFS implementations will store the complete Directory List in memory, rather than rereading it from the media.

Each entry in the Directory List array contains a `cdblock` pointer to the location of the most recent version on the disk of that particular directory. Each directory, in turn, contains `cdblock` pointers to the most recent versions of the files it contains. (A `cdblock` pointer is a 64 bit quantity which points to a specific block/byte pair on the disk.)

Rewriting the Directory List at the end of each transaction consumes substantially less space on the media than the alternative of rewriting every directory containing modified subdirectories. The storage space required by the Directory List is very small in proportion to the space available on optical media. For example, in June 1985 the entire user file system of MEDIA-LAB.MIT.EDU, a VAX 11/785 servicing several hundred users, contained 522 megabytes of user files in fewer than 2000 directories. If this entire file system was stored on one CDROM using CDFS, the space required by the Directory List would be 74,000 bytes, or less than 0.014 % of the total file space. The example is a bit contrivied since the expected use of optical media is as a personal storage media rather than a system storage facility.

3.3 Directories

A directory is a special structure in CDFS which contains a list of directories, files and links. Stored with the name of each file and link in a directory is the location on the media of the most recent version of that file or link. It is necessary to rewrite a directory at the end of a transaction in which one of its containined files or links have been modified. It is not necessary to rewrite a directory for which only subdirectories have been modified, since the location of these directories is determined from the Directory List.

3.4 Files

Like most file systems, CDFS stores user data in objects called files. CDFS files consist of two parts: the *file header* and the *file contents*. The *file header* is invisible to the user and stores out-of-band information about a file. The file header of a file contains a `cdblock` pointer to the file contents, which may be stored anywhere on the disk, but which typically follows the file header.

In CDFS, a file is modified by rewriting the file's file header and file contents to the media during the course of a transaction. CDFS associates a unique number called the `file_number` with each file on a disk. Successive versions of a file on a disk have an identical `file_number` but a sequential `version_number`. Directories are likewise numbered; a number, once used for a directory or a file number, may not be reused, even if the file or directory is deleted. The root directory of the disk is assigned the number one (1).

Under our current implementation, file contents must be stored contiguously and be completely rewritten when any modification is made within a file. This approach, while inefficient for large files which are modified in small ways, is easy to implement and relatively efficient for small files. For large files subject to relatively small modifications, such as databases, we have designed a second file storage

standard, described below in the Section 7.2, "fragmented files."

Some conventional file systems store most file attributes in the file's containing directory. CDFS attempts to keep directory size small. Small directories minimize the impact of having to rewrite the entire directory after a modification.

Some information stored in file headers is redundantly stored in directories in order to improve file system performance and protect against media failure. For example, the file list command can list the contents of a directory, along with the size, file type and modification time of every contained file, without requiring reference to the file header of each contained file. Storing the byte count of each file also aids in automatic recovery of data in the event of media failure.

Directories are implemented as a special kind of file in which the file contents are interpreted by the file system. The file contents of a directory are assumed to contain a `directory_info` data structure followed by an array of `dir_contents` elements, one for each entry in the directory. Each `dir_contents` element contains a pointer to the location on the disk of each file or link contained in the directory.

3.5 Links

A link is an entry in a directory which points to another entry in the file system, called the target of the link. A link target may be a directory, a file or another link. Links are stored as file headers without associated file contents.

3.6 Independence Through Detailed File Headers

CDFS file headers average over 240 bytes in length – substantially larger than the equivalent structures in other operation systems. CDFS can afford to be generous with the amount of storage space allocated to file headers because of the large amount of space available on optical media.

This additional space is used to make explicit information usually assumed by conventional file systems, which eliminates the dependence on operating system databases when decoding the contents of a disk. For example, CDFS stores the "owner" of every file in a 32 character array contained in the file header. The UNIX file system also has a notion of file owner, but instead stores a 16 bit user number for each file. In order to translate from user number to user name, a system database must be referenced.

The `site_name` field in the file header is another example of information implicitly assumed in most file systems is made explicit in CDFS. Most file systems assume that every file on the media is created at the same site, hence the name of the site is not stored on a per-file basis – indeed, it is usually not stored at all. In CDFS, where it is assumed that the media moved in the normal course of operations from site to site, the name of the site writing the file is stored in every file header. The `site_name` also facilitates the use of CDFS as a networked archiving systems in which it is desirable to tag each file with the name of the computer that wrote it.

The data in the CDFS file header is arranged in six structures called segments. Each segment is assigned a seperate name and version number in order to minimize the problems when reading disks

mastered with different versions of the file system. This modularization of the standard allows an intellegent CDFS implementation to partially recover when presented with a CD written with a non-standard CDFS implementation. For example, a site might have made a local change to the `site_info` specification, but if the `access_info` is unmodified the CDFS implementation might still be able to locate the file from the file header.

The file header segments are:

`fileheader` Contains general information about the file, including file number and file type. The `fileheader` segment also contains the offsets of the other segments in the file, relative to the start of the file header. The `fileheader` segment must be the first segment in the file header, but the other segments may follow in any order.

`access_info` Contains file ownership and access permissions.

`backup_info` Contains historical information about the file, including pointers to previous versions and full pathname.

`file_info` Contains file attributes specific to the current version of the file, including version number, write time, creation time and file length.

`site_info` Contains information pertinent to the site which wrote the file.

`property_list_info` Provides a mechanism for attaching additional out-of-band information to a file.

In a link, the `file_info` segment is substituted with a `soft_link_info` segment:

`soft_link_info` This structure replaces `file_info` in links and contains information about the link, including target directory and path.

(See appendix E for the structure of each of these segments.)

4 An Example Transaction

This section follows the development of a file hierarchy on a compact disk over the course of two contrived Transactions. In this example, transaction update occurs as it does in the Media Lab's CDFS implementation. Other implementations might perform transactional update differently.

In actual practice, virgin disks are "preformatted" with a super block, an empty root directory, a Directory List, and a second EOT. This process is not required by the specification and has been omitted here for clarity.

As previously mentioned, CDFS only writes consecutive blocks. In this example, the CD starts blank. In the first Transaction, the compact disk is created with two files, *life.c* and *wheel.c*, in the root directory. In the second Transaction, the file *life.c* is modified while the file *wheel.c* remains untouched.

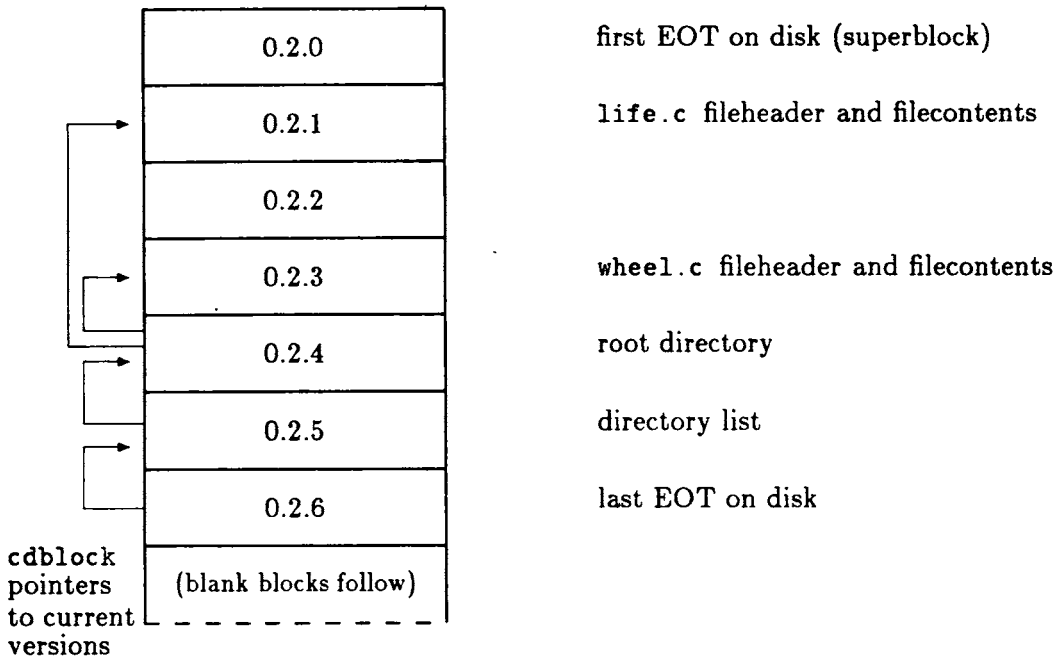

Figure 1: Direct Read After Write disk after a sample transaction

In this example, the first transaction is finished after CDFS writes the two files, the Directory List and the EOT to the disk. A pictorial view of first few CD blocks is presented in Figure 4. A description of the function of each block follows.

Block Location Explanation:

0.2.0 The first block of a CDFS generated disk contains an EOT. EOTs contain media-specific information about the disk, such as the recording format, `cdblock`, and the version of the CDFS by which the disk was created. This block also contains the name of the disk's owner, the name of the site which created the CD and other human-readable information.

The Philips/Sony CDROM standard specifies that the first two seconds of a CDROM must be blank. We have placed the superblock at 0.2.0 in this example. (The Media Lab implementation of the CDFS actually sequentially scans the first five seconds of a CD searching for a superblock. When it finds the block, it examines the block's self-referential pointer to determine if there was any skew introduced in the CDROM mastering process.)

0.2.1 The file `life.c` is stored contiguously, preceded by its file header. The contents of the file are positioned independently of the file header but follow contiguously in this example.

0.2.3 `wheel.c` is the second file on the CD, stored as file header followed by file contents.

0.2.4 At the end of the transaction, directories modified during the course of the transaction are written to the disk. In this example, the root directory contains two entries.

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

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

The second transaction takes place some time after the first. Between the two transactions, the disk has been removed from the write-once drive, perhaps moved to a different site. Before the second transaction begins, the disk must first be mounted. When the disk is mounted, CDFS reads the disk's format from the superblock, then locates the last written block, which is an EOT. Since the disk was properly dismounted at the end of the first transaction (that is, an EOT was the last written block on the disk), the disk mounts without any errors.

From the EOT at **0.2.6**, CDFS obtains the location of the Directory List. The Directory List contains pointers to the current version of every directory on the disk. Each directory contains pointers to the file headers of all non-directory contents (subdirectories are located via the directory list). File headers point to file contents. In this manner, locating the EOT allows any byte resident in the file system to be easily located.

After the second transaction, in which a new version of *life.c* is written, the pictorial view of the CD would resemble Figure 2. Note that none of the blocks written in the first transaction have been modified.

New block Explanation:

0.2.7 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.

0.2.10 A new root directory is written out after the modified files. The directory points to the most recent version of the file headers of the files it contains.

0.2.11 The Directory List is written after the last directory. It contains a pointer to the most recent version of each directory.

0.2.12 The End-Of-Transaction block is written last. It contains a pointer to the previous End-Of-Transaction and to the current Directory List.

Note: Directories, Directory Lists and End-Of-Transaction blocks may be written at any time. The Media Lab implementation of CDFS maintains all modified directories in memory until the transaction is closed; in a limited-memory implementation, directories could be written to disk and freed from memory at any time.

5 Driver Layer

The CDFS Driver Layer is responsible for reading and writing data to the DRAW device. Other tasks, such as spin-up, spin-down and seeking are preformed automatically by the Driver Layer as required to accomplish reading and writing functions.

Locations on the write-once media are denoted by `cdblock` pointers. A `cdblock` pointer consists of 64 bits partitioned into a group of bits which denote a block address and a group of bits which denote a byte offset in that block. Thus, a `cdblock` pointer points to a particular byte on the compact disk.

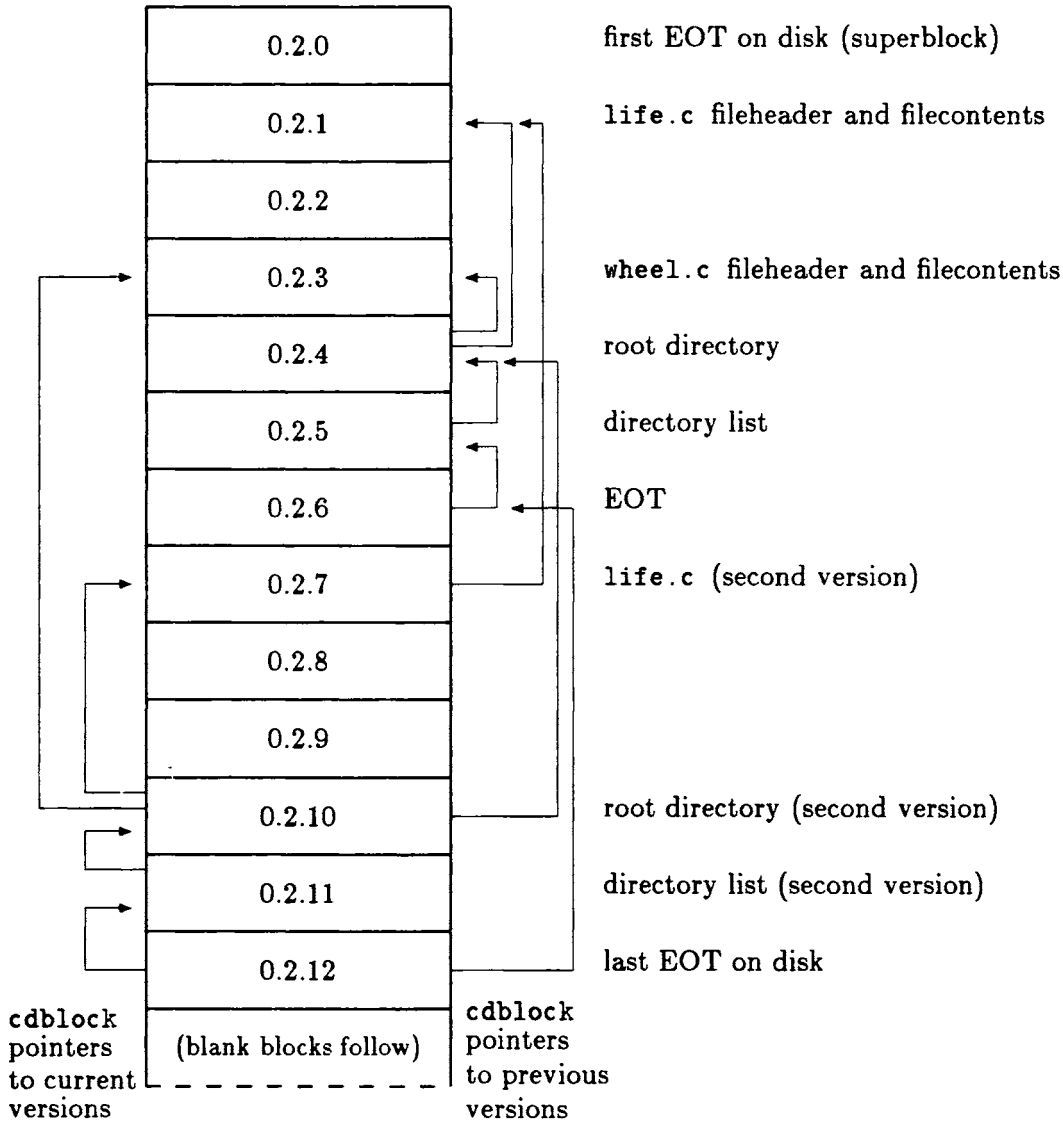

Figure 2: Direct Read After Write disk after a second sample transaction

The partitioning and interpretation of the 64 bit `cdblock` quantity is the responsibility of the Driver Layer. Presumably, the block address field will be further partitioned in a way reflecting the addressing scheme used by the DRAW hardware. Details of the partitioning scheme are recorded in the `pointerdes` array located in every End-Of-Transaction block on the disk (the actual format of the `pointerdes` array is presented in Appendex B).

In the case of audio-format CDROMs, the `cdblock` is partitioned into 48 bits of block address and 16 bits of byte-offset in block. The block size is 2048 bytes. The 48 bit block address is further partitioned by the driver into three 16 bit fields: minute, second and block number. Other partitionings are possible. When the audio partitioning scheme is used, the contents of the `pointerdes` array in every End-Of-Transaction naturally follows as:

Name	`modulo_of_value`	`bits_in_value`
Minute	70	16
Second	60	16
Block	75	16
Offset	2048	16

The remaining entries in the `pointerdes` array are blank and ignored.

The use of the `pointerdes` array allows multiple partitioning schemes using the same physical media (although not on the same disk). It also allows multiple media formats to be used easily by the same CDFS implementation.

The driver layer must provide the following functions to the CDFS:

- Return physical information pertaining to the disk currently mounted, including block size and the last addressable block on the disk.
- Read a block, returning the contents, if possible, or an error indication. Error indications must distinguish between a virgin block and an unreadable block.
- Write a sequential block on the CD.
- Calculate successive `cdblock` addresses.
- Find the first unwritten block within a given range. Normally, the driver layer accomplishes this function with a binary search of the media.

6 Current Implementation

CDFS is currently implemented as a C subroutine library. The implementation is byte-order word-size insensitive. The implementation runs on CDFS on a Digital VAX 11/785 running Berkeley 4.3 UNIX, a network of Sun Microsystems 68000-based workstations, under MSDOS 3.1 on several IBM PC microcomputers, and on the IBM RT/PC. Programs which wish to access files stored on a CD must be

specially modified and linked with the CDFS library. Work is underway on a networked file system server[3] for CDFS which will allow a CD to be mounted transparently into our Sun and VAX file systems.

The CDFS was developed with a DRAW simulator. The simulator, running under UNIX, makes several large disk files take on the appearance of a write-once CD. The simulator-based system doubles as a premastering system for the making read-only CDROMs.

When used as a mastering system, data desired on the final CDROM is copied into the simulated CD's CDFS file system. When all of the desired files are in place, the UNIX files are copied to eight 2400 foot 6250 bpi nine-track tapes, which are used to master the CDROM.

CDFS will be used for user file storage in a public workstation environment. During a work-session, the user copies the files he is working on from the CD to a local hard disk on the workstation. At the end of the work-session, modified files are copied back to the CD. Eventually, we hope to be able to support diskless workstations and automatic copying of modified files back to CD.

The Media Lab CDFS implementation lacks fragmented files, addnames, resolution of links and handling of reserved properties. These features will be added as needed during the coming year.

7 Technical Issues Arising From Design

7.1 Active File Collection

As a CDFS updates media, an increasing amount of "garbage" – old versions of files, directories, directory lists and EOTs occupies the disk. Eventually, the disk fills up. At or before this time, the most current version of each file in the hierarchy can be copied to a fresh disk to give the user more working space. This process is analogous to stop-and-copy garbage collection in many LISP environments.

7.2 Fragmented Files

"Fragmented files" is a CDFS file representation format which does not require file contents to be written contiguously on the disk. The fragmented file specification (see Appendix E.1) allows large databases on a write-once CD to be updated without requiring the entire database to be rewritten to the disk.

Fragmented files consist of a file header, a file map and strips of data, or fragments, containing the file's contents. The `file_location` field of the file header points to the file map, which in turn directs CDFS to the location of valid data in the file. Data stored in the fragments may reside in arbitrary locations on the disk. The format of the file map is presented in the appendix.

The intent of the fragmented file specification is to require only modified data of a file to be written to the write-once media when large files are updated. Data can be logically deleted from a fragmented file by rewriting the file map. Likewise, data can be inserted or modified merely by rewriting the file map and the associated new data. Fragments of data referenced by the file map can logically or

physically overlap, although the fragments must be identical in the region of overlap. The file map may reference fragments of data which are resident in other files, or in previous versions of the same file.

Fragmented files additionally allow multiple files to be open for update at once. Without fragmentation, implementations must either open only a single file for output, or buffer file write operations on a file-by-file basis, extensible for user applications.

7.3 Addnames

Each file or directory occupies a single location in the hierarchical directory tree. The 48-character name field in a directory permits long, descriptive names for files and ensures that file names of most other file systems will fit. There is no restriction on the length of pathnames (the file name appended to the list of all of its containing directories) or the number of subdirectories.

CDFS allows files and directories to have multiple names, similar to the "addname" facility present in the MULTICS[2] operating system. After a file or directory is created with one name, additional names can be added. Addnames can be deleted from files with more than one name. CDFS system calls behave in the same manner regardless of which addname of a file is specified. Deleting a file with several addnames deletes the file – a separate facility is provided for removing addnames. All of a file's addnames are confined to the same directory.

Addnames are stored in a CDFS directory as entries which have a filetype of ADDNAME. Addnames are resolved by searching the containing directory for a non-addname entry with the same filenumber as the addname. Information such as the file version number and file location is found from the primary name's entry. Included in the addname's header is a count of the number of the file's addnames. In the addname file headers, only the `file_name`, `modify_time`, `file_number` and `file_type` directory entry fields of an addname are relevant.

7.4 Soft Links

A link is an entry in a hierarchy which points to another entry (called the target) elsewhere in the hierarchy. Links have been implemented under several operating systems, including MULTICS[2] and UNIX[1]. CDFS provides a linking facility generally refered to as "soft links."

A Soft Link consists of an entry in the containing directory and a file header, in which the `file_info` segment of the file header is substituted with a `soft_link_info` structure. The `soft_link_info` segment stores the target of the link. The target is specified by a directory number, a character string and a version number. The directory number is interperted as a starting point for the filename resolution of the character string.

If the target directory number of a soft link is the same as the directory number in which the link resides, then the link is a pointer to another name in that directory. Conversely, if the target directory number is 1 (the root directory), the link points to an absolute location in the hierarchy.

Since the target directory is pointed to by a directory number, rather than a directory name, a

link will continue to point at its target directory even if the target directory is renamed or moved. This feature facilitates the management of hiearchies containing large numbers of links.

Links which contain non-zero target version numbers are interperted as pointers to a particular version of the file or directory, rather than as a pointer to the most recent version.

7.5 Undeleting Files

In CDFS, a file is "deleted" by rewriting a directory without an associated entry for that file: the actual file on the disk is never modified. Files may be "undeleted," or reinserted into the hierarchy, by reinserting a file's entry into the containing directory. This entry can be found by searching through previous versions of the containing directory. This pointer in the containing directory's file header to the previous version of the directory makes this search a simple task.

8 Summary

The Compact Disk File System (CDFS) has been developed by the MIT Media Lab to allow immediate exploitation of DRAW devices as they become available. CDFS was designed to provide the useful capabilities of multiple-write file systems on write-once media. The CDFS also provides new functionality as a consequence of the write-once nature of the media. For example, it provides for a complete modification history of every file and the ability to recover files after they have been "deleted."

CDs are portable and, using CDFS, the information stored on them is operating system independent. CDFS stores all information pertaining to a given volume on the volume itself. No directory information need be stored on associated multiple-write media. This permits a file system resident on a CD to be moved from one site to another by merely moving the disk from one drive to another.

CDFS is a useful superset of all commonly available file systems. CDFS stores most information about file contents and attributes that are maintained by most operating systems including UNIX, MULTICS, VMS, VM/CMS, MSDOS, TOPS-20 and many others. This permits, for example, UNIX hierarchies and VM/CMS minidisks to be copied to the same CD and accessed using either operating system.

CDFS maintains the appearance of a traditional read/write, hierarchical file system, with directories, files, file property lists and links. The standard also includes provisions for coherent extension.

CDFS uses a "layered" approach in implementation. Specifics such as block size, block addressing, and device capacity are relegated to a "driver layer" which need provide only a small number of standard primitives to the CDFS. Driver layers can be written to allow use of the CDFS with any write-once media, including DRAW digital video disks and punched cards.

CDFS never rewrites or overwrites the contents of a block. Instead, CDFS continually writes new versions of files, directories and other volume information. Normally, the "current" or most recent version is automatically selected and updated. Earlier versions of files and directories can be quickly retrieved.

CDFS does not require the preallocation of separate data and directory partitions on the CD. Instead, the CD is treated as a stream of blocks which are written sequentially. A file system which preallocated space would either unreasonably restrict the user (who might have run out of directory space while ample file space remained) or require complex overflow area management.

While CD drives have high transfer rates, they have long seek times. With this in mind, the CDFS has been designed to minimize the number of seeks required to find any file on a volume. Mounting a CD may require up to twenty seeks to locate the end of the written block stream, but once a CD is mounted, any file's current version can be located in at most one seek per level of depth in the directory tree. An additional seek may be required to read the contents of the file.

CDFS stores a substantial amount of redundant information on the disk, allowing accurate automatic reconstruction of a volume after media failure or interrupted transactions.

CDFS does not require that files, directories, or other volume information be written in any particular order on the CD. Very little information need be cached to implement CDFS, although performance can be substantially improved by caching of all file system structures. In our prototype implementation, performance improvements of over 1000% have been realized through caching.

A Technical Notes on CDFS Data Structures

Note on data structures presented in this document:
All data structures and program examples in this document are presented in the form of C language fragments taken from the MIT Media Lab's CDFS implementation. In the interest of clarity and portibility, the programming example only reference derived types such as `int16` *(a sixteen bit unsigned integer) and* `int32` *(a thirty two bit unsigned integer). These derived types are defined separately for each compiler.*
`Intnn` *defines an unsigned integer* **nn** *bits wide, stored low-byte first.*

The End-Of-Transaction, Directory List and Fileheader structures all contain three distinct means of self-validation. These are an identifying constant string, a checksum and a self-referential pointer. Using concurrently all three of these validating methods minimizes the chance of a mis-identification, such as confusing a file header with a file that contains a file header image. Finally, validation is only required to detect and recover from media failure. Normally, the EOT is located by searching the media for the last-written block and the Directory List and fileheaders are located from `cdblock` pointers in the EOT, Directory List and directories, respectively.

All CDFS timestamps are 64 bit numbers representing the number of seconds (not counting leap seconds), since 12:00 Midnight, January 1, 1901 GMT.

CDFS checksums are calculated as the sum `int16` integers, modulo 65536. The `checksum` field of a checksummed data structure is set so that the checksum of the structure will evaluate to 0.

Elements in the directory list are sorted by directory number, root directory (directory number one) first. Directory numbers are the file numbers of directories. File numbers are unique on a disk and assigned sequentially.

A.1 Notes on End-Of-Transaction

`Encryption_standard` is a site-dependent reserved space to allow for the denotion of encrypted disks. The field is ignored in the Media Lab's CDFS implementation.

`Owners_name` is a NULL terminated string denoting the ASCII representation of the name of the owner of the CD. The length of the string may be calculated by subtracting the offset of `owners_name` in the structure from `eot_length`, or by searching for the NULL.

A.2 Notes on Directory List

The directory list array is stored sorted by directory number.

The `modify_time` of a directory is the most recent modification time of any of its contents. This quantity is recursively calculated.

The `contained_bytes` of a directory is the total of bytes contained by a directory, including bytes contained in subdirectories. In the case of fragmented files, file size refers to the number of valid data bytes allocated by the file's file map.

`Modify_time` and `contained_bytes` of a directory are calculated when the transaction is closed.

A.3 Notes on Directory Structures

A directory is a file with `filetype` of `DIRECTORY TYPE`. The file's content's consist of a `directory_info` structure followed by a packed array of `directory_contents` elements.

Directory entries are stored with 48 characters reserved for the entry name. These entries are padded with the NULL character but need not be NULL terminated. The octal characters 000 (NULL), 0376 (down) and 0375 (up) are reserved and may not be part of an entry name.

Subdirectories are located by reference to the directory list, not to the `header_location` field in the containing directory. Therefore, the `header_location` for a `directory_entry` that is a directory is meaningless; an all zero location is used.

`Directory_contents` elements are stored sorted alphabetically by `file_name` for fast lookup.

A.4 Notes on File Headers

`Access_info_offset`, `backup_info_offset`, `file_info_offset`, `site_info_offset` and `property_list_offset` are figured as byte offsets from the start of the `fileheader`.

`ACCESS_INFO_VERSION` #1 defines file access permissions as in the UNIX file system. Access permissions are defined by types of access allowed to the file's owner, users in the file's group, and

everybody else.

`Backup_info_.backup_pathname` denotes the location of the file referenced by the file header when the file header is written. If any containing directory is renamed or moved, `backup_pathname` will be wrong. `filename_offset` is the byte-offset of the filename in `backup_pathname`, counting from the beginning of the `backup_pathname`.

The property list begins with the `property_list_info` structure. If the offset to this structure is 0, then there is no property list. The property list is used to denote a set of user-definable properties associated with the file. Properties may be flags, such as "BITSTREAM" or "CLASSIFIED," in which their `property_value_len` is 0. Values, if specified, must be in ASCII.

The following properties are at the present time defined:

BITSTREAM Denotes that the information stored in the file is to be taken as a stream of bits and not as a stream of 8 bit characters (as would be the case with an ASCII text file or an object code file on a computer with a word-size that is a multiple of 8 bits). characters.

The BITSTREAM property must be accompanied by the UNIT property.

RECLEN n Record size of the file, expressed in characters.

UNIT Size of the word of the machine that created the BITSTREAM. On a PDP-10, for example, this would have the value of 36.

VOLUME-NAME For use with soft links, denotes a link to a file resident on another CD.

B 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;
```

C Directory List structures

C.1 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;
```

C.2 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;
```

D 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;
```

E 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;
```

E.1 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;            /* location of first byte in strip. */

        int32   valid_chars;    /* number of valid bytes in strip */

        int32   ordinal;        /* byte offset in logical file of strip */
} fragmented_des;
```

References

[1] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for unix. *Berkeley UNIX documentation*, 4.2, 1983.

[2] Elliott I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, 1972.

[3] Sun Microsystems, Inc. *Networking on the Sun Workstation.* Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA, 94043, 1985.