MacFS: A Portable Macintosh File System Library

Peter A. Dinda George C. Necula Morgan Price

July 1998
CMU-CS-98-145

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We have created a Macintosh file system library which is portable to a variety of operating systems and platforms. It presents a programming interface sufficient for creating a user level API as well as file system drivers for operating systems that support them. We implemented and tested such a user level API and utility programs based on it as well as an experimental Unix Virtual File System. We describe the Macintosh Hierarchical File System and our implementation and note that the design is not well suited to reentrancy and that its complex data structures can lead to slow implementations in multiprogrammed environments. Performance measurements show that our implementation is faster than the native Macintosh implementation at creating, deleting, reading and writing files with small request sizes, but slower than the Berkeley Fast File System (FFS.) However, the native Macintosh implementation can perform large read and write operations faster that either our implementation or FFS.

Keywords: Apple Macintosh, HFS, Hierarchical File System

1 Introduction

With the large and growing number of Apple Macintosh computers in use, accessing data stored on Macintosh volumes has become important. Although Apple and other vendors provide software to read and write DOS FAT volumes and there are a number of products that implement the FTP protocol on Macintosh systems, both of these methods for accessing Macintosh data require physical access to a Macintosh and neither has the immediacy of directly using a Macintosh diskette or SCSI hard disk on one's machine of choice. Clearly, a Macintosh file system for non-Macintoshes is a useful thing.

Further, the broad range of "non-Macintoshes" (Unix workstations, DOS and OS/2 based PCs, etc) motivates a portable library approach. Instead of "reinventing the wheel" with each new platform, our portable library requires only that two low level routines (read/write sectors and some of the mount code) be written. The library provides a consistent programming interface over which such things as a user level API, a Unix VFS, or an OS/2 IFS can be built.

1.1 Goals

Our first goal was to define a programming interface for MacFS that could support a a variety of applications such as a user level library for accessing Macintosh volumes and a Unix Virtual File System (VFS). [VFS] We demonstrated that we fulfilled this goal by creating a robust user level library and utility programs, and an experimental VFS. Secondly, we wanted to make our implementation as portable as possible and to minimize and partition platform-specific code. Our implementation has only two small platform-specific modules and has been tested on a DEC Station/5000 and on an Intel 486 system.

Finally, we decided that our implementation had to properly handle the per-file Finder Information (see section 2), which, although not strictly a part of the file system, is nonetheless important in maintaining the "visual metaphor" that Macintosh users expect. We accomplished this by maintaining Finder Information on existing files, supplying reasonable information when creating files, and providing a mechanism for changing it.

1.2 Organization

In section 2 we begin by describing the Macintosh Hierarchical File System (MHFS) in some detail. Section 3 defines the programming interface of our library and our user level interface. Section 4 is a detailed description of our implementation. We discuss some of the issues that arise out of the complexity of MHFS in section 5. Section 6 presents performance data on our user level implementation and comments on our VFS implementation. Finally, we conclude in section 7.

2 The Macintosh Hierarchical File System

All Macintosh storage devices except floppy disks are partitioned into one or more volumes.[IMF] Volumes can contain four kinds of items: files, directories, directory threads, and file threads. Each item is described by a *catalog record*, which is analogous to a Unix inode.[FFS] Catalog records

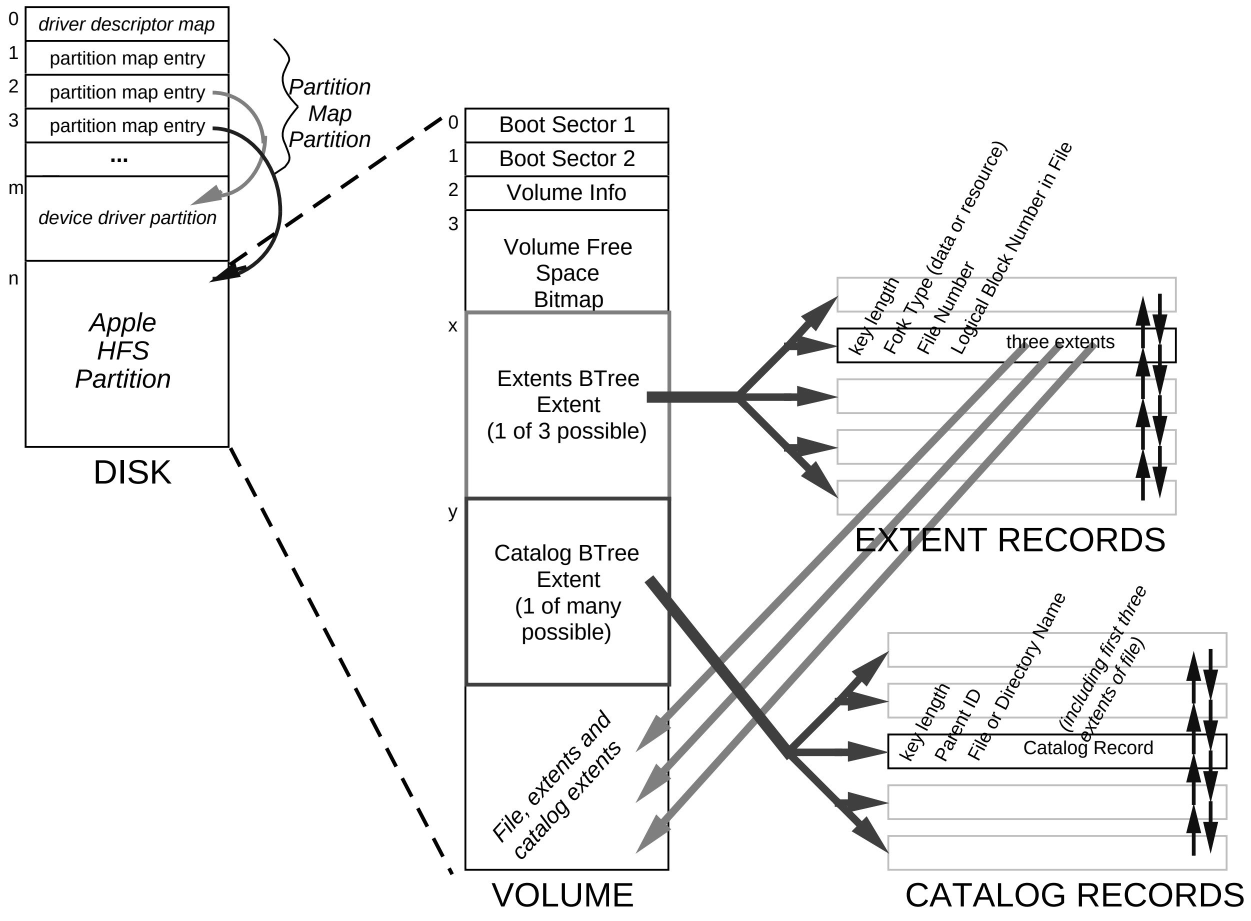

Figure 1: Macintosh Hierarchical File System Data Structures: The highlighted catalog record represents a single file. The highlighted extent record holds additional extent descriptors of the file's data fork beyond the three in the catalog record.

are organized in the on-disk *catalog B-Tree*. Directory contents are derived from searching the catalog B-Tree. Only a file can occupy space outside of its catalog record.

A Macintosh "file" contains two components, or *forks*. The *resource fork* is an indexed file containing code segments, menu items, dialog boxes, etc. The *data fork* has the "stream of bytes" semantics of a Unix file's contents. When we refer to a file, we mean the data fork portion of the "file".

Each fork is comprised of one or more *extents* or contiguous runs of blocks. An *extent descriptor* encodes an extent's starting block and length into a 32 bit quantity. The first *extent record* (three extent descriptors) of each fork is a part of the file's catalog record. Any further extent records are kept in the *extents overflow B-Tree*.

In addition to file and B-Tree extents, a volume also contains two boot blocks, a volume information block, and a free-space bitmap. There is a remarkable amount of redundancy in the on-disk data structures, which improves crash recovery.

While not strictly a part of the file system, it should be noted that several catalog record fields are reserved for the exclusive use of *Finder*, a program which handles user access to the file system and automatically maintains associations between applications and data files. Thus, MacFS must also maintain this Finder info.

Figure 1 graphically summarizes MHFS, the Macintosh Hierarchical File System.

2.1 Naming and the Catalog B-Tree

Every file and directory on an MHFS volume has an identification number, similar to an inode number in the Unix file system. However, a file or directory is named by its *parent's* identification number and the file or directory's filename, which is a 32 character string that can contain nulls. This combination is the search key to the volume's catalog B-Tree file.

The catalog B-Tree differs from a traditional B-Tree structure in that all the nodes at each level of the B-Tree are linked together to form a doubly linked list, and all of the records are in the leaf nodes. These variations permit accessing many items in the same directory by traversing the leaves using the linked list. Strictly speaking, the MHFS B-Trees are a variant of B+-Trees [Comer], although Apple's technical documentation calls them B*-Trees. For the purposes of this paper, it is only important that they are B-Trees and we refer to them as such.

Each directory, including the root directory, contains its directory thread, which has the empty filename. The directory thread record contains the name of the directory, and the id of the *parent* of the directory. Similarly, file threads contain the name of a file and the id of the directory they are in. While every directory must contain a directory thread, file threads are very uncommon. In fact, both are examples of MHFS' redundancy – for undamaged trees, threads are not strictly necessary.

Both file and directory records contain 32 bytes of information used by Finder. Although the contents of the Finder information fields are outside of the purvue of the file system, placing valid and useful information in these fields in necessary for transparent migration of data to a Macintosh. Most importantly, the creator name and file type information *must* be correct because some Macintosh applications will only see "correctly" typed/created files.

The first three extent descriptors for the catalog B-Tree file are kept in the volume information block. If the catalog B-Tree file grows beyond three extents, the remaining extent descriptors are kept in the extents overflow file.

2.2 The Extents Overflow B-Tree and Finding a File's Extents

A file's catalog record contains the first three extent descriptors of both its resource and data forks. Many Macintosh files, even large ones, have three or fewer extents. Any extents beyond the first three are kept in the extents overflow B-Tree file. This B-Tree has the same structure as the catalog B-Tree, except that it holds extent records (three extent descriptors) and is searched by file ID, fork ID, and logical block within fork. Finally, it cannot grow beyond three extents. Those three extents are described by an extent record in the volume information block.

3 Programming Interfaces

The programming interface of the MacFS library is referred to as the *m* interface. This interface uses only MHFS concepts. For example, a file or folder is named by the mount record of the

m Function	*mac* Function
mmount(pn,mr)(user), mmount(vnode,mr)(VFS)	macmount(pn,mr)(user)
munmount(mr)	macunmount(mr)
mcreatefile(mr, pcr, fn, cr)	maccreatefile(mr, pn, mo, mf)
mcreatefolder(mr, pcr, fn, cr)	maccreatefolder(mr, pn)
mmove(mr, old pcr, old fn, cr, new pcr, new fn)	macmove(mr, old pn, new pn)
mmovefile(mr, old pcr, old fn, cr, new pcr, new fn)	no equivalent
mmovefolder(mr, old pcr, old fn, cr, new pcr, new fn)	no equivalent
mdeletefile(mr, pcr, fn, cr)	macdeletefile(mr, pn)
mdeletefolder(mr, pcr, fn, cr)	macdeletefolder(mr, pn)
various functions	macgetfinderinfo(mf, field1, field2)
various functions	macsetfinderinfo(mf, field1, field2)
mopen(mr, pid, fn, cr)	macopen(mr, pn, mo, mf)
mclose(mr, pid, fn, cr)	macclose(mf)
no equivalent	macseek(mf, offset, whence)
mread(mr, cr, offset, numbytes, buf, actual)	macread(mf, numbytes, buf, actual)
mwrite(mr, cr, offset, numbytes, buf, actual)	macwrite(mf, numbytes, buf, actual)
mgrowfile(mr,cr,newsize)	no equivalent
mtruncatefile(mr, cr, newsize)	no equivalent
mdirentries(mr, pid, cr list)	macdirentries(mr, pid, cr list)

Figure 2: MacFS Programming Interface (*m* functions) and MacFS User Level API (*mac* functions). The following abbreviations are used: [p]cr=[parent] catalog record, fn=file or folder name, mf=user level file handle, mo=mode options, mr=mount record, pid=parent ID, pn=pathname.

volume it is on, the catalog record of its parent folder, and its name. In some cases only the ID of the name is used. This is because the parent of the root folder has no catalog record, only a canonical ID.

The API of the user level library we built using MacFS is referred to as the *mac* API. This interface mimics the Unix open/read/write/close interface. A file or folder is named by the mount record of the volume it is on, and an absolute pathname. Both interfaces are detailed in Figure 2.

4 MacFS Software Structure

We decided early on that MacFS should be easily portable to different operating systems and hardware. This resulted in a design which has only two platform-dependent module, *read/write device support*, which exports procedures for reading and writing partition relative sectors and reading absolute sectors from a device, and *mount support*, which has the concept of a platform-dependent device name. We have implemented two sets of these modules, one using Unix I/O calls (open/read/write/close) and device naming (/dev/...), the other using Unix kernel buffer operations and internal kernel device names.

Above the read/write device support module, MacFS consists of a large amount of shared code. Small modules include *partition support* and *volume bitmap support*. The major shared code

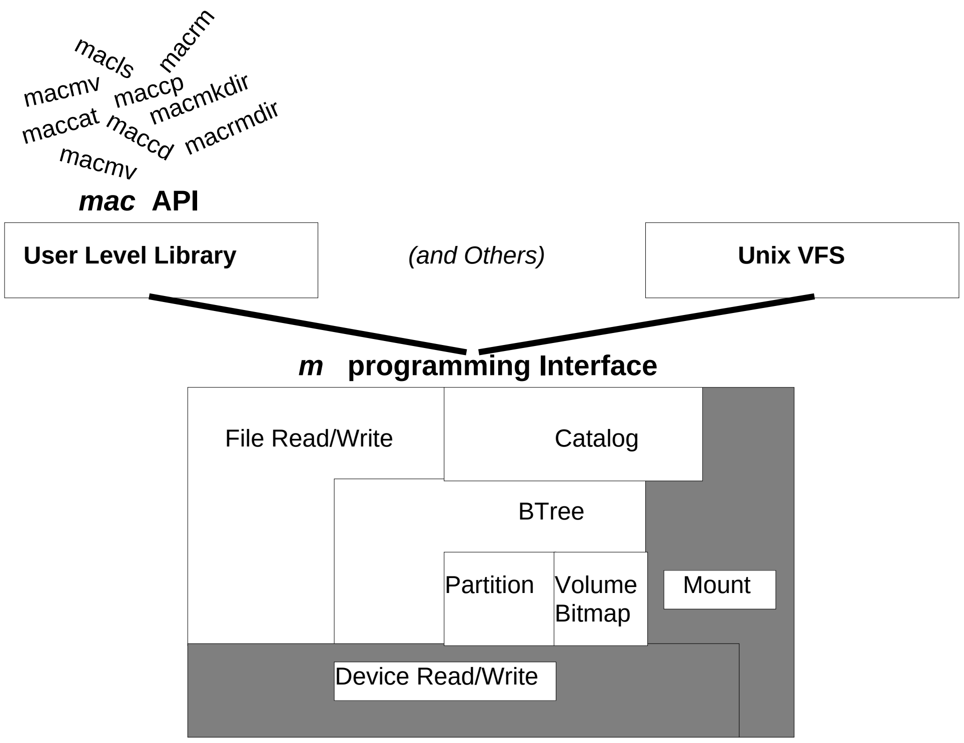

Figure 3: MacFS Software Structure - The shaded modules contain platform-specific code

modules are *B-Tree support*, *catalog support*, and *file read/write support*. The mount, catalog, and file read/write support modules collectively export a base interface, the *m* interface, which contains functions to do all typical file system operations in a non platform-specific way.

On top of this, we have built two interfaces. First, the *mac* API is an interface with Unix-style file naming and manipulation functions. When MacFS is compiled into a user level library, it is the mac API which is exported. We have written mac-based versions of the standard Unix utilities ls, cp, cat, cd, mkdir, rmdir, rm, and mv to demonstrate how the mac functions are used. The mac library and the example utilities should be directly portable to any platform which has a Unix-like I/O library.

Second, we implemented an experimental VFS interface on top of the m interface so that Macintosh volumes can be directly mounted on Unix systems as if they were Unix file systems. We feel that the m interface is sufficiently general that mountable file systems for other operating systems' foreign file system interfaces, such as OS/2 IFS, can be constructed on top of it.

The MacFS software structure is graphically summarized in figure 3.

4.1 Common Data Types

Throughout MacFS, a *mount record* represents a volume. In our initial design, it only contained basic information such as partition offsets, but we soon discovered that converting between unaligned, big endian data on a Macintosh disk and aligned, possibly little endian data in memory in order to change such frequently accessed structures such as the volume information block consumed far too much time. Therefore we now keep the volume information block, the free space bitmap, the headers for the extents and catalog B-Trees, and an extent record cache for the catalog B-Tree in a volume's mount record.

A *catalog record* represents a file, directory, or thread. It has the same fields as the on-disk catalog record, but adds lock, reference count, and validity fields. Opening a file instantiates its catalog record in memory. The in-memory catalog record is then used for all read/write operations on the file and is finally flushed back to the disk on a close call.

Reading, writing, growing, and truncating a file builds an in-memory *extent descriptor list* that holds the extent descriptors necessary for the request.

4.2 Read/Write Device, Partition and Mount Support

An early design choice for the read/write device support module was to support multiple sector reads and writes. With the late addition of partitioning support, these functions were modified so all sector requests were made partition relative. Further, a read absolute sector function was added for the use of the partition support module. Finally, an unbuffered write function, used at sync time, was added.

The partition support module simply sets the offset necessary to make all reads and writes partition relative. Surprisingly, a Macintosh device can have a wide variety and number of partitions, including device driver partitions.

The mount support module provides mount, unmount, and sync functionality. Mounting a device consists of opening the device, setting the partition offset, and reading in the volume information block, free space bitmap, and the headers for the extents and catalog B-Trees. Syncing writes back those data structures using unbuffered writes. Unmounting syncs, closes the device, and marks the mount record as invalid.

4.3 B-Tree Support

The purpose of this module is to manage the two major data structures that exist on a Macintosh volume: the catalog B-Tree and the extents overflow B-Tree described in section 2. The module relies on functions to read/write raw sectors from disk and exports functions to lookup, insert and delete catalog and extents records from the B-Trees.

The module is structured in two functional layers: a generic B-Tree layer and a layer specialized in the catalog B-Tree and the extents overflow B-Tree. The only purpose of the specialized layer is to translate the information between on-disk catalog and extents records and in-memory structures.

On a Macintosh volume all data is laid out unaligned in big endian byte order. On an MC68000 based machine turning the volume data structures into usable C structures could be done by casting the raw data from the disk to carefully chosen C structures or in the worst case by a memory to memory copy operation.

The machines we used to implement MacFS use little endian byte order (MIPS R3000 and I486) and have precise alignment requirements (MIPS R3000), so we have to copy and byte-swap all the information when translating between disk structures and memory structures. In addition, the specialized layer must translate between Pascal strings (on the disk) and C strings. These must be done on all operations, including key comparison during tree traversals, which is a significant overhead cost.

The generic B-Tree module is used to perform lookup, insertion and deletion in a generic B-Tree. The module is parameterized with a key compare routine that is dependent on the type of tree acted upon.

4.4 Catalog Support

The catalog support module exports functions for: creating and deleting files and directories, moving and renaming files and directories, getting a list of the contents of a directory, opening and closing files and directories, and changing Finder information on open files. Rollback is used to recover from errors in all functions.

Naming and Catalog Records

In the catalog support module, files and directories are identified by the volume's mount record, the catalog record of their parent directory, and their own name. For example, creating a file requires a mount record, the catalog record of the directory the file will be placed in, and the name for the file. The catalog record of the newly created file is returned.

Extending the naming conventions of MHFS (section 2) by using catalog records instead of IDs was necessary to simplify the VFS level implementation of MacFS and should make writing installable file systems for other operating systems simpler. Catalog records are the closest MHFS analogy to a Unix inode. For some m functions, a catalog ID must be used because the parent of the root folder has no catalog record, yet it must be possible to open, close, and list the contents of the root folder.

Directory Contents

A function which returns a list of the names and catalog records of all the items in a directory is provided. This is a necessary support function because, unlike the Unix file system, MHFS directories are not files that contain lists of subitems. Altough we support "opening" directories, the semantics of opening a directory is to return its catalog record, which contains no information about the contents. To obtain a directory list, repeated catalog B-Tree searches are done with descending filenames until the directory's thread record is found.

4.5 File Read/Write Support

Files are identified in the file read/write support module by their catalog record and the mount record of the volume they are on. This is all that is necessary because the functions in this module only modify the in-memory catalog record of an open file and the extents B-Tree. The module provides functions to read and write files and to truncate the length of files.

Read/Write Requests and Extent Descriptor Lists

Any read or write request first generates an extent descriptor list enumerating the extents necessary to complete the request. For example, suppose a file has two extents of 512 byte blocks, the first, 10(2) starting at block 10 and extending for two blocks, and the second, 50(10) starting at block 50 and extending for 10 blocks. A read request for 1000 bytes starting at byte 1000 will result in the extent descriptor list $\{11(1), 50(2)\}$ being "knitted." The "block extents" represented by the elements of an extent descriptor list are translated to "sector extents" for the ultimate calls to *readsects()*.

Growing a File

When a file is created, a single extent of the volume's default "clump size" is allocated for it. When a given write will extend beyond current physical length of a file, *mgrowfile()* is called to expand the file. First, a search and update of the volume free-space bitmap is performed, building an extent descriptor list detailing the newly allocated space. The search starts at a per-volume "start of next allocation search" kept in the volume's information block. Then, the new extent descriptors are incorporated into the file's extent records. Any open slots in the catalog record's extent record are first filled. This is followed by a foray into the extents overflow B-tree, updating and/or adding new extent records for the file.

In all cases, new extent descriptors are coalesced into existing extent descriptors, if possible. For example, suppose a file has a single extent record $\{10(2), 50(10), 0(0)\}$, and a bitmap search resulted in the extent descriptor list $\{60(20), 100(1), 110(10)\}$. mgrowfile() will update the extent record to $\{10(2), 50(30), 100(1)\}$ and add a second extent record, $\{110(10), 0(0), 0(0)\}$ to the extents B-Tree. Extent coalescing is instrumental to fast file access in MacFS.

We feel that extent coalescing could be done more often if each file catalog record had a "start of next allocation search" field. The per-volume field seems to be a holdover from when the Macintosh was a one-application-at-a-time, one-document-at-a-time machine. Even in simple situations where two files are being grown in a interleaved fashion, extent coalescing becomes very unlikely on the Macintosh and in the current implementation of MacFS.

5 MHFS Design Issues

The B-Tree structure allows for relatively fast lookups (the tree has a maximum of 8 levels), which leads to fast `stat` operations because, as described in section 2, all the file information is stored in the catalog B-Tree. This is particularly useful on the native Macintosh because path traversal involves `stat`ing every file at each directory level [1]. However, inserting and deleting file system metadata can be very expensive. There are several other issues that arose in our implementation.

[1] For example, traversing the path **a:b:c** in either a dialog box or Finder involves opening **a**, `stat`ing all its items, having the user select **b**, then `stat`ing all the items in **b** ...

5.1 B-Tree Complexity

A large part of the B-Tree code is needed to handle extremely uncommon cases which could have been eliminated if the MHFS design were less flexible. Complexity is added by the fact that each B-Tree has an internal, non-fixed size and noncontiguous bitmap spread among the regular nodes. This bitmap records the allocation status of the B-Tree nodes. This extra code is involved only if the number of files is extremely large (30,000) because the first portion of the bitmap is stored at a known location. If the number of B-Tree records were bounded, the complete bitmap could be stored at a known location.

The feature that is responsible for most of the extra complexity of the B-Tree code is that both trees are files, which can dynamically grow and are not necessarily contiguous. The first three extent descriptors of both B-Trees are stored in the volume information block. Any further extent descriptors are stored in the extents overflow B-Tree. This feature is not documented so we tried to experiment with a native Macintosh file system. We noticed that the catalog B-Tree grows relatively quickly because of the high size of a catalog record (5 records in a node on average) while the extents overflow B-Tree grows very slowly because of a general low fragmentation and because the extent record is small (40 records in a node on average). Our experiments have shown that the additional extents for catalog B-Tree are stored in the extents overflow B-Tree. We were unable to sufficiently fragment a native Macintosh volume to cause the extents overflow B-Treeto grow beyond three extents. Therefore, in our implementation we refuse to grow the extents tree beyond the three extents that can be stored in the volume information block. Growing the extents overflow B-Treebeyond three extents and storing the additional extent descriptors in the extents overflow B-Tree itself would lead to circularity and possible deadlock.

Storing the extents of the catalog B-Tree in extents overflow B-Tree can lead to very expensive metadata accesses even in the case of a simple lookup. This can happen when the path from the root to the target leaf contains nodes located in different extents that must be searched in the extents overflow B-Tree. To alleviate the problem we currently cache the catalog B-Tree's extent descriptors.

5.2 Reentrancy

Making an MHFS-compatible file system reentrant and highly concurrent amounts to permitting concurrent operations on the B-Trees. This turns out to be a difficult task. We note that Apple's implementation is not reentrant.

Fine grain synchronization – at the B-Tree node level – would maximize concurrency. Algorithms exist for such synchronization, but they determine whether a node needs to be locked or not (unsafe versus safe) by exploiting bounds on the number of keys in a node. For example, [Bayer] notes that for a degree d B-Tree, a node is safe on insert if it has fewer than $2d$ keys because if one of its children needs to be split, there is room for the middle key on the node. However, the MHFS B-Tree nodes hold variable length keys, making it much more expensive to determine the whether there is enough room for another key. Further, growing an MHFS B-Tree into a new extent requires a second, coarser level of synchronization.

Currently, we simply lock the entire catalog B-Tree or extents overflow B-Tree when used. This does not limit concurrency much because catalog records are instantiated in memory when a file is opened, and few files have more than the three extent descriptors stored in the catalog record. Even on a heavily fragmented volume, concurrency between the catalog B-Tree and extents overflow B-Tree remains.

Our conclusion is that the designers of the Macintosh file system considered the flexibility of the file system data structures the most important issue. We feel that with simpler and possibly fixed size and position data structures the file system would perform better and would also allow for better reentrant implementations.

6 Evaluation

To evaluate the MacFS library, We addressed the issues of the portability of the code, the versatility of the m programming interface, and the functionality and performance of the implementation.

6.1 Portability and Versatility

We developed the MacFS library on two very different machines, a DECstation 5000/120 and an Intel 486 system. The significant differences between these machines demonstrates the portability of the code. Further, we implemented both a user level library and a Unix VFS on top of the m programming interface of the MacFS library, demonstrating the versatility of this interface. Other implementors have used MacFS to build an OS/2 IFS driver and for accessing Macintosh media from a professional digital audio recorder.

6.2 Functionality

We used the Macintosh Finder and the Macintosh Norton Utilities 2.0 to test that disks modified by our user-level and VFS-level MacFS implementations remain Macintosh compatible. Although we are able to copy any type of file, all new files created have the Finder Info of text files, unless modified using macsetfinderinfo().

6.3 Performance

We evaluated the performance of our user level library, the native Macintosh file system, and the Unix FFS using five simple benchmarks:

- Create: Create a large number of files.
- Delete: Delete a large number of files.
- Write: Create, write, and close one 5,000,000-byte file with request sizes of 500, 5,000, and 50,000 bytes.
- Overwrite: Open, overwrite and close one 5,000,000-byte file using the same range of block sizes.
- Read: Read one 5,000,000-byte file using the same range of block sizes.

			500 bytes/req			5,000 bytes/req			50,000 bytes/req		
Platform	Create	Delete	Write	Over.	Read	Write	Over.	Read	Write	Over.	Read
Macintosh	7.5	7.5	26.5	27.0	28.7	118.2	119.2	124.2	527.0	532.9	548.9
MacFS	24.6	23.2	83.0	71.1	173.6	147.8	182.4	303.8	211.8	169.7	303.2
Unix FFS	111.5	571.7	265.7	150.0	254.7	373.7	139.7	406.2	410.1	388.8	398.7
	files/s	files/s	KB/s	KB/s	KB/s	KB/s	KB/s	KB/s	KB/s	KB/s	KB/s

Figure 4: Performance Comparison between the native Macintosh file system, the MacFS user library, and the Unix FFS

The the difference between the Write and Overwrite benchmarks is that Overwrite does not include the cost of growing the file. All reads and writes were sequential. The benchmarks were run on a 240 MB Hitachi DK312C disk on a DECstation 5000/120 (24 MHz MIPS R3000) and on a Macintosh 145B (25 MHz MC68030.) On the Macintosh, the benchmarks were run using the native OS and file system, APS SCSI driver 2.7, and the Think C 5.0 Unix I/O emulation libraries. In all cases, the disk was formatted with a 4K logical block size.

Figure 4 compares the performance of the three file systems. Note that the MacFS library can create and delete files three times as fast as the native Macintosh file system. There are three explanations for this. The first is that the Macintosh may have to send refresh messages to open Finder windows and dialog boxes on file creation or deletion. The second is that the Macintosh may immediately update the on-disk mount record and free-space bitmap, something that MacFS defers. Finally, initializing Finder Information may be more complex on the Macintosh. The Unix FFS is much faster than either MHFS or MacFS due to the greater simplicity of FFS metadata.

In the Write, Overwrite, and Read tests, the MacFS library is consistently slower than the Unix FFS, but faster than the native Macintosh file system for all but the largest request size. At 50,000 bytes per request the native Macintosh is even faster than the Unix FFS. We suspect that with larger request sizes the Macintosh is better able to amortize the cost of extent lookup, or that a different extent lookup algorithm comes into play with very large request sizes. Noting that MacFS read-write speeds do not increase monotonically with request size and that read performance seems to peak around 5,000 bytes per request, the latter suspicion is probably the case.

The performance of our VFS implementation is not sufficient to warrant inclusion in the table. Because it was intended mainly as a proof-of-concept of the versatility of the m interface, we did not tune its performance.

7 Conclusion

We have presented MacFS, a portable Macintosh file system library, and described its design, implementation, and performance. We demonstrated the versatility of the library interface by implementing a user level library and a Unix VFS on top of it. In addition, we commented on the design of the Macintosh Hierarchical File system data structures. Although we feel that the complexities of MHFS metadata makes a maximally concurrent implementation difficult, our performance measurements show that MHFS can achieve quite high data read/write performance.

Bibliography

[Bayer] R. Bayer and M. Schkolnick, “Concurrency of Operations on B-Trees,” *Acta Informatica*, 9,1 (1977), 1-21.

[Comer] D. Comer, “The Ubiquitous B-Tree,” *ACM Computing Surveys*, Volume 11, No. 2, June 1979, 121-137.

[FFS] M. McKusick, et al., “A Fast File System for UNIX,” *ACM Transactions on Computer Systems*, Volume 2, No. 3, August 1984.

[IMF] *Inside Macintosh: Files*, Apple Computer, 1992, p23–76.

[OS2IFS] L. Solomon “OS/2 Installable File Systems,” *EDM/2*, Volume 1, Issues 3 and 5. *An electronic newsletter FTPable from ftp-os2.cdrom.com.*

[VFS] S.R. Kleiman., “Vnodes: An Architecture for Multiple File System Types in Sun Unix,” *USENIX Association: Summer Conference Proceedings*, Atlanta, 1986.