An MS-DOS File System for UNIX

Alessandro Forin
Gerald R. Malan

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

We have written DosFs, a new file system for UNIX that uses MS-DOS data structures for permanent storage. DosFs can be used anywhere a traditional UNIX file system can be used, and it can mount disks written by MS-DOS as regular UNIX partitions. DosFs can be used as the root partition, and exported by an NFS server. Our motivation for this work was efficient disk space utilization; DosFs provides from 30% to 80% better disk utilization than the 4.3 BSD Fast File System (FFS). In addition, we found that the disk block allocation algorithm used by DosFs lets us exploit large contiguous disk operations, providing a five-fold improvement over FFS for uncached operations. To the best of our knowledge, this is the first file system implementation that allows the user to specify the logical block size at mount time. A user can mount the same fileystem with a larger block size when file accesses tend to be sequential and a smaller one when they tend to be scattered. The MS-DOS structures were designed for a single-user system and do not support access control. We solved this problem with simple extensions that are backward-compatible with MS-DOS.

1. Introduction

Today's micro-kernel architectures make it possible to run multiple operating system environments simultaneously. For example, a notebook computer running Mach 3.0 [2] can have both UNIX and MS-DOS disk partitions. In fact, an MS-DOS emulator [12] can be started from within the UNIX environment so that Windows applications can run in parallel with UNIX applications. Unfortunately, UNIX cannot access files on the MS-DOS partition while the MS-DOS emulator can access all UNIX files. The perception from UNIX is that part of the disk is missing, causing inconvenience both in resource utilization and in the sharing of data between applications that belong to the two different worlds. Other commercial MS-DOS emulators have similar limitations. A number of toolsets do exist to let UNIX users access MS-DOS floppies and disks, but they only transfer data between the UNIX and MS-DOS file systems. The best solution is an integrated file system that empowers all existing tools and applications.

We had both technical and practical motivations for this work. On the technical side, we wanted to examine and compare the permanent data structures used by MS-DOS and FFS, check whether it was possible to extend a single-user file system like the MS-DOS one into a multi-user safe, networked file system, and understand some initial performance figures for uncached disk accesses. On the practical side, we wanted to extend our UNIX system to support more disk formats, including MS-DOS and the ISO 9660 [5] disk formats. Some of our users expressed a desire to have the UNIX backup and restore utilities work on their MS-DOS data. Finally, a UNIX local filesystem should be usable as the root file system for a multi-user UNIX system.

There are very few file systems for UNIX that handle permanent storage, the latest BSD 4.4 release has added

LFS to the only one that was previously provided. There are a few other file systems outside of the BSD sources, but they all use permanent data structures similar to the original UFS ones. The permanent data structures used by MS-DOS are different enough from the standard UNIX ones to make it interesting to explore the effects on functionality and performance. For example, there are no inodes in MS-DOS.

The MS-DOS file system was designed for a single-user environment, on a personal computer. UNIX is used in multi-user, possibly security-sensitive environments. On security grounds, the single-user file system model is not acceptable on UNIX. For instance, MS-DOS has no provisions for storing user ids or access control information, but it is not acceptable to make files and directories readable and writeable by all users. The models of the expected file system usage might instead be closer. On UNIX, it is common to assume a time-sharing model, such as was traced by John Ousterhout [10] in his investigations of the 4.2 BSD file system. But even the most powerful workstations today are typically used by a single user at a time, and the file systems themselves use sophisticated caching strategies. It might well be that the MS-DOS model is quite appropriate for most UNIX users after all.

In our initial tests we found that native MS-DOS 5.0 was up to five times faster than our emulated UNIX (BSD 4.3 on top of Mach 3.0) for large uncached reads, and four times faster than the monolithic Mach 2.5. We traced the disk operations with a SCSI bus analyzer to find an explanation. We found that the MS-DOS file was contiguously allocated and was being accessed with four times larger block sizes (e.g. 16 KB at a time) just like our user program was requesting. The FFS file was allocated in an interleaved fashion to cope with rotational delays [8], and was being accessed at the file system block size, 4 KB. The extra seek times accounted for the remaining discrepancies.

Assuming we were able to reach the same optimal disk allocation for files as in MS-DOS, we realized that we could provide new functionality to our users, namely the ability to specify the disk access size (logical block size) dynamically, at mount time. Depending on prevailing use, with DosFs we can either use a small or a large block size, whichever gets the best performance, without any need for rebuilding the file system. A smaller block size is preferable in case buffer-cache misses are non-sequential and dispersed across the file system. A larger block size is best in the more common case that accesses tend to be sequential and clustered around a few files at a time.

Finally, we wanted to support the ISO file system organization (and the RockRidge extensions for POSIX/UNIX [13]) that is standard in CD-ROM disks. We started the project with some code written by Pace Willisson at Berkeley, which provided a read-only ISO file system implementation. The code was missing the RockRidge (RR) extensions and the file handle operations for use with NFS. Both were easy to add. Support for the MS-DOS file system looked, at first, like another simple addition because the ISO file system has various similarities with the MS-DOS file system. We just had to add the write-part--- which turned out to be more than twice the initial code.

The remainder of the paper is structured as follows. Section 2 describes the organization of the MS-DOS file system and permanent disk data. We assume the reader has some knowledge of the BSD FFS data structures, descriptions can be found in [8, 7]. Section 3 details the main design and implementation problems. Section 4 illustrates the programs we wrote for file system maintainance. In Section 5 we look at the performance optimizations from the following viewpoint. We use native, uncached MS-DOS as the base for each test. Each optimization is described and its individual effect quantified relative to either the base or the final performance figure. In many cases we can use run-time flags to toggle a specific optimization on or off to help us understand the measurements. Section 6 reports on the first evaluation criteria we used, namely the disk utilization profiles for FFS and MS-DOS. Section 7 describes the testing setup and the results of various performance tests. The tests themselves were taken from those commonly used in the file system literature and were selected to illustrate the effects of one or more performance optimizations. In Section 8 we point out similar and related work and finally draw a few conclusions from our experience and illustrate the status and availability of the code.

2. The MS-DOS File System Structure

Starting from the first sector of a disk, the MS-DOS file system layout is as follows; further details can be found in [1]. The initial section of the disk contains the primary (1 sector) and the secondary bootstraps (a variable number of sectors). This is followed by one or more copies of the File Allocation Table (FAT), which is followed by the root directory entries. The rest of the disk is divided into dynamically allocated *clusters*. Figure 2-1 graphically depicts this layout.

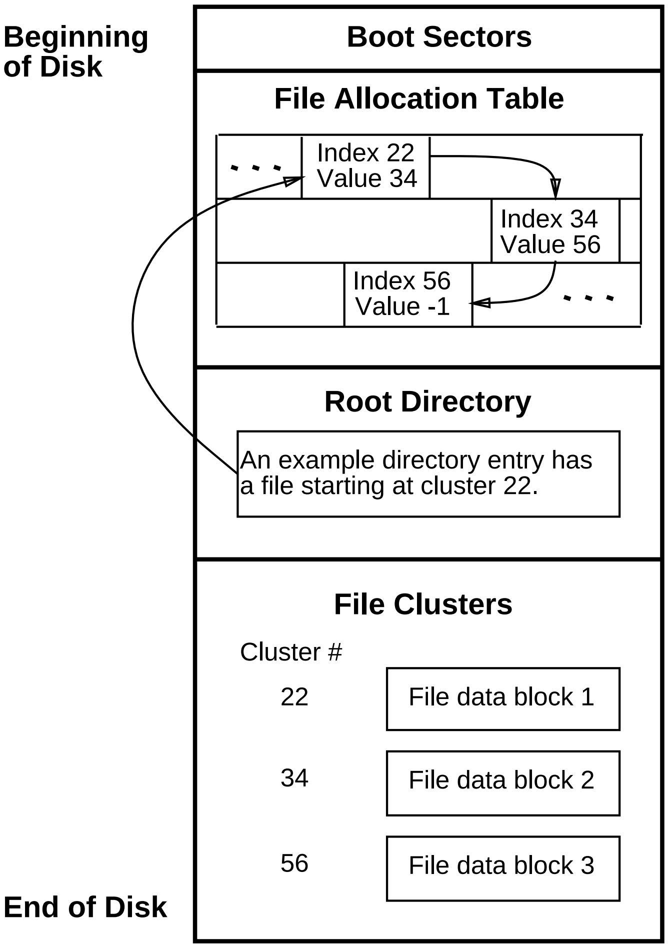

Figure 2-1: Organization Of The MS-DOS File System

The first sector of the disk contains the primary bootstrap code, intermixed with geometry information about the disk and possibly partitioning information. It also records the location, size and number of FATs, the size of the root directory, and the size of physical sectors and clusters. The primary bootstrap code uses this information to load the secondary bootstrap code into memory and transfer control to it.

The FAT is a fixed size table that describes the allocation status of the disk's clusters. Entries are either 12 or 16 bits wide, depending on the total number of clusters in the disk partition. A floppy has a 12 bit FAT, a fixed disk a 16 bit FAT. The first two FAT entries are reserved. MS-DOS uses a single FAT copy on floppy disks, and two copies on fixed disks to protect against disk corruption. DosFs accepts and keeps coherent any number of copies of the FAT.

To determine which disk blocks are allocated to a file (or directory) we must start from the file's directory entry. The entry contains the file's starting cluster number, e.g. the first block of the file. To find the next block, we use the cluster number as an index in the FAT. The value at that entry is the cluster number of the second block, and so on. The last block of the file has a FAT entry value of -1. If a cluster is not allocated, its FAT entry is zero. Other distinguished values are used to indicate reserved clusters (bad blocks, for instance).

Directory entries are fixed in size, 32 bytes each. Every entry contains the name and extension, an attribute byte that indicates, among other things, if this is a directory or a file, a timestamp, the size of the entry (zero for directories), and the starting cluster number. There is room for 10 more bytes that are defined as reserved by the file system specifications. The upper two bits of the attribute byte are also reserved/undefined. In the initial versions of MS-DOS, the root directory was the only directory and it could not grow. Later versions of MS-DOS are backward compatible and also have a fixed, contiguously preallocated root directory. The root directory is therefore special, fixed both in size and disk location at file system creation time. Other directories can be grown/shrunk as necessary. Some directory entries are special, and identified as such by the attribute byte. The MS-DOS image itself, for instance, is marked invisible and read-only. **Label** entries contain information about the creation time of the file system, and the volume name. There is only supposed to be one such label entry, in the root directory.

Clusters are logical blocks, e.g. multiples of the physical sectors. Their size is typically 512 bytes on small floppies and 2048 bytes on fixed disks. Just as in FFS, any cluster can contain either file data or directory information. Unlike FFS, no other type of information (metadata) is recorded in the clusters. The cluster size can actually be set to any power of two multiple of the physical sector size, similarly to the fragment size in the UNIX BSD file system. Unlike BSD, the cluster size can be up to a 16 bit multiple of the sector size. In practice, the upper limit supported by MS-DOS is 64 KB.

3. Design Issues

The MS-DOS file system evolved from its CP/M predecessor, it is aimed exclusively at a single-user machine, with no protection between users of the file storage. It strives to minimize resource utilization, both in memory and on disk. But it lacks features present in UNIX file systems, such as file links.

DosFs is implemented as a layer of code below the virtual file system (Vnode) layer [6]. The Vnode layer has a well defined interface, which must be fully implemented to cover all aspects of UNIX file system functionality. This presented us with various design issues, which we will describe in the rest of the section. Table 3-1 summarizes these issues, and indicates how they are solved in the various file systems.

Problem	*FFS*	*IsoFs*	*MS-DOS*	*DosFs*
Access Control	**inodes**	**extended attributes**	**none**	**directory ext. or mount option**
Filename size	**255**	**30 or RR unlimited**	**11**	**11**
Filename chars	**ASCII**	**Upcase+**	**Upcase++**	**mount option**
Inode numbering	**Inode Table**	**Extent num**	**N/A**	**Cluster num**
Links	**hard+soft**	**hard+soft with RR extensions**	**none**	**soft**
NFS access	**inode num**	**inode + dir**	**N/A**	**inode + dir**
Special devices	**standard**	**RR ext.**	**no**	**directory ext.**
Setuid files	**standard**	**RR ext.**	**no**	**directory ext.**

Table 3-1: Permanent Data Structures And UNIX Functionality Issues

3.1. Protection

In FFS, a directory entry contains only the name of the file and the inode number. A separate table, the inode table, contains a 128 byte record for each inode with various information, including the user-id, group-id and access permissions of the inode, its type and (a portion of) the list of blocks owned by the inode. There are no inodes in the MS-DOS file system, and a directory entry does not have the user-id, group-id, or access mode information (except for read-only files). There are two approaches for adding access protection to DosFs. We could use the 10 reserved bytes to store the extra information or we could dynamically calculate the access permission for each file.

The first approach is acceptable if MS-DOS applications ignore those 10 bytes (they do) or if backward compatibility and interoperability are not concerns. It is our goal to interact as smoothly as possible with MS-DOS, including potential applications that might themselves store information in those reserved bytes. Therefore we provide the extended information option only if DosFs is mounted as the root partition, or if explicitly requested via mount options. In practice, we have not yet found an MS-DOS application that would look at the content of reserved bits and bytes, nor does MS-DOS itself complain or reset these bytes. A disk utility (such as our *dosfsck*) can simply reset the extra protection information, to guarantee 100% compatibility with MS-DOS. We have designed our extensions so that no data is lost if the access information is destroyed.

For the second approach there are several possibilities:

1. [root,wheel] owns all files; write permissions depend on mount mode (read-only prevents all writes) and on the read-only bit in the attribute byte; all users have equal access rights;
2. same as above, but the user who issued the mount command is substituted for [root,wheel]; access modes are taken from the umask and propagated to all files; other users have access rights as per the umask;
3. allow the user to specify an arbitrary user-id, group-id, and access mode mask; otherwise, same as #2 above;

All three options essentially give the same ownership to all files, they just specify this ownership differently. We selected option #2 as default because it allows a user to mount his/her volumes and access them safely via NFS without any special options. Privileged users or operators can use option #3.

Regardless of the mount options, if a file on disk has the special extension marker we use the protection information stored on disk; only the mode mask is used to mask off specific bits. An interesting use of the user-specified mode mask is to disable setuid programs or execute permissions, regardless of their value on disk. This could be used by a cautious operator when mounting suspicious floppy disks on a general-purpose machine, to get rid of Viruses.

A possible alternative was to make use of the special *label* directory entries. MS-DOS specifies that they are optional and they would only appear in the root directory. We could put a label entry on each directory, to store the protection information. This idea is similar to the AFS Access Control Lists, which are set on a per-directory, not per-file basis [14]. We were told by Microsoft that it is not a good idea to misuse label entries.

The ISO file system defines *extended attributes* that provide protection information, including owner and group identifiers, and permissions. These are stored along with the file data in the first block of the file, separated from any directory entry information. ISO leaves open the mapping of this information onto the receiving system’s administrative database.

3.2. File Naming

MS-DOS file names are limited to 12 characters, 8 bytes of name proper, an implied dot character to separate the file extension, and three characters of extension (suffix). The ASCII space character must be used to pad both the name and the extension fields. The MS-DOS specifications only dictate 8-bit ASCII characters but some internal MS-DOS functions converts all characters to uppercase. Therefore in practice all file names in MS-DOS have uppercase only ASCII characters, and digits. Special punctuation symbols are discouraged; some might produce peculiar effects.

The ISO file system sets a total limit of 30 characters on a file name, not counting the dot separator between name and extension and the trailing version information (a semicolon followed by a digit). The only permissible characters are digits, uppercase letters, and underscore. The RockRidge (POSIX) extensions use only the ASCII codes for letters and digits, period, underscore and hyphen. Extra information is added to a directory entry to express a file name in this POSIX format and bypass the ISO restrictions. This file name encoding scheme has no maximum length.

UNIX BSD file names have a set limit of 255 characters, and no restrictions on their content except for the first seven bits of the ASCII code. In principle, raising the 255 limit would be just a matter of modifying one define and recompile. There is only one function that imposes the seven bit restriction in our 4.3 BSD file system code.

To cope with this jungle of rules and exceptions we wrote separate file name comparison and translation functions for the FFS, DosFs and the ISO file systems. In the case of the ISO file system, these handle the RR extension and recover the original UNIX names, bypassing the ISO limitations. The file name length limitation for MS-DOS instead remains, we are still looking for an acceptable workaround, such as using label entries or a separate string table.

We did remove the uppercase-only limitation: at mount time it is possible to specify one of three translation options. File names can be untranslated (except for the implied dot between name and extension) and look exactly as they would under MS-DOS. Or we can map letters to lowercase and clean up the extra spaces, to make file names look more UNIX-like. This is the default option. When used as a root file system we use a third scheme that treats name and extension as a single string, without any implied "dot" and without case conversions. This third option provides the most "proper" feeling for UNIX users, but it might make the file name unacceptable to native MS-DOS.

3.3. Inode Numbering

FFS maintains permanent inode tables on disk, and there is a direct mapping from inode number to disk address. Since ISO file systems do not have inode tables we used the first block number of directories and files (the *extent*, in ISO parlance) as the inode number. We maintained this scheme for the DosFs file system, the only modification is that our *dosfs_ialloc()* function must procure a cluster each time we create a new "inode". For directories this is not a problem since a directory by definition must contain at least the dot and dot-dot entries (this is true of both UNIX and MS-DOS) and therefore can never be totally empty. For files this means that zero-length files still own at least one cluster. Special devices are the only inodes for which we might safely and quickly optimize away this cluster, by dynamically generating impossible cluster numbers and modifying the file system utilities to recognize these directory entries.

This simple scheme no longer worked when we added the RockRidge extensions to the ISO file system. The problem is with symbolic links. In ISO a symbolic link is entirely defined in the directory entry, it does not have any associated disk space. We could have used extension-specific knowledge, such as looking into the extra information provided by the RR records that include an inode number. Instead, we took an approach that should still work with some other future extensions to the ISO standard. The high bits of the inode number are the extent, and for regular files and directories, the low bits are zero. For symlinks, the high bits are the extent of the directory where the symlink lives, the low bits are a function of the directory entry (e.g. the offset in the extent). Any given ISO extension need only tell if this is a symlink or not, the existing code will build the inode number appropriately.

3.4. Hard/Symbolic Links

MS-DOS has neither symbolic links nor hard links. Implementing symbolic links was simple. We just took one unused bit in the attribute byte of the directory entries to mark the entry as having our special extensions, and another bit in the extension bytes to indicate a symbolic link. We store in the link's cluster the file name the link is pointing at, just like FFS. For MS-DOS the file is a regular file of length equal to the link name, and the content of the file is the link name itself.

Short FFS symlinks can be optimized by using the direct block fields of the inode to store the file name this is a link to, avoiding one disk access while doing expansion of symbolic links. DosFs has no inodes and cannot use this optimization. Notice, however, that in terms of disk accesses DosFs is always at par with the optimized case. FFS must access the directory entry, the inode, and (possibly) the link name. DosFs must access the directory entry and (always) the link name.

Hard links are more difficult. One approach is to just build another entry that points to the same starting cluster. This creates two problems: uniqueness of inode numbers is lost, and there is no obvious way to keep (on permanent storage) a reference count of the number of entries that point to the same inode. The first problem means, for instance, that size and other attributes that are stored in a directory entry are not kept coherent, unless the entry that is a hard link actually points back to the original entry it is a link to. Since a cluster can hold a number of directory entries this variation can still perform well, for the higher probability of finding the pointed-to entry in memory. But more importantly, the second problem means that there is no way to know when an inode can actually be released. One solution to this problem is to use more of those 10 reserved bytes to hold a hard link count.

Note that these issues do not arise on ISO CD-ROM file systems for the simple reason that the file system is immutable. The RockRidge extensions do support symbolic links, and they also provide reference counting. Implementing hard links and making a mutable ISO file system is therefore feasible.

For expediency, we chose to implement symlinks but not hard links. We needed symbolic links to handle the file system structure with more flexibility, especially when DosFs is the root. We did not find any mandatory reason to have hard links, so we left the issue open for future extensions.

3.5. NFS Access

Exporting a (local) file system via NFS is not difficult. The NFS layer gives to client machines a *handle* on Vnodes, the local file system provides functions to build such a handle and map it back to a Vnode. FFS handles contain just the inode number and a generation number for coherency purposes. Neither ISO nor MS-DOS have inode tables separate from directory information, we could not use just the inode number as is done in FFS. If the inode is not in the cache we need to refer back to some directory entry to recover size, modes and timestamps of a DosFs or ISO inode. This is also true with the RockRidge extensions to ISO.

The handle that is exported to remote nodes in NFS has a predefined size, but fortunately it is opaque. No application appears to make special use of the content of an FFS handle. We added two more fields to the FFS file handle scheme, the cluster number of the directory that contains an entry for the given inode, and the offset of this entry. If the inode is not in core it can be reconstructed by recovering the original directory, checking its coherency, and looking up the directory entry. Note that protection issues are no different when access is via NFS than when access is local.

3.6. Other extensions

When we first tried to use DosFs as a root file system on a DEC Alpha workstation we discovered a small number of other problems. To begin with, we had to modify both the boot programs and the Mach default pager [3] to understand the MS-DOS file system structures. The first must be able to find, read in, and transfer control to the boot image. The second has similar needs with respect to the UNIX server image. In addition, it might have to use a DosFs file system for paging purposes.

We then found a number of instances in filesystem-independent UNIX code where FFS data structures and semantics were assumed. For instance, mounting of the root file system itself was special cased for FFS and NFS and so was the checking of special device close operations against mounted file systems. Even the shutdown of the system assumed the root was an FFS file system. We also saw no reason why the root file system should not be un-mounted, e.g. on clean shutdowns of the system.

Two more extensions were necessary to get the system up in multiuser mode. The first and most obvious, in hindsight, was the need for special devices. We used the *mode* extension field to mark a file as a UNIX special device and stored the major/minor information in the reserved bytes. Special devices are the directory entries that make the most use of the reserved bytes in MS-DOS directory entries; all 10 bytes are used up.

The second, slightly less obvious, was the need for setuid programs. CMU UNIX runs the single user shell not as *root* but as the less privileged user *opr*, therefore we just could not do much without encountering a protection block. Once again, we used the *mode* field for this information, making it almost identical to the FFS *i_mode* field. The only difference is that DosFs does not support the special mode value used by *badsect(1)* for creating bad block files. There is no need for this, the FAT defines special entry values to mark unusable clusters.

4. Tools

While developing DosFs we found the need for tools to accomplish a variety of tasks, such as file system construction, check and repair, bad block handling, analysis, and tuning. We patterned a small set of tools after the well known UNIX ones: *dosmkfs*, *dosfsck*, *dosbadsect*, *dosdumpfs*. All of the tools are considerably simpler and smaller than their FFS counterparts. The *dostunefs* program is actually just a compaction tool that reallocates clusters on disk to keep file/directory data contiguous for best performance. We hardly ever use it. A similar effect can be obtained simply with a dump/dosmkfs/restore, but with tunefs there is no need for a separate tape or temporary disk partition.

Once we started to use DosFs as root partition we decided to actually integrate dosfsck back into fsck, which can now handle either type of file system, including support for the so-called "preen mode". This is the way fsck operates non-interactively, e.g. during automatic reboots. There are only two very embarrassing cases where fsck will not automatically repair the disk: when an allocation chain in the FAT has a loop and when two allocation chains merge. FFS and fsck have many more mutual understandings of accidents that "should not happen". Another strange property of the DosFs fsck is that it is CPU-intensive, not I/O intensive. Most of the time is spent verifying the FAT allocation chains in memory, I/O operations to read directory entries take little time.

5. Optimizations

A number of optimizations came to us for free, by using the UNIX buffer cache for I/O. These include the caching of I/O buffers, read-ahead for sequential file accesses, and delayed-writes. Other optimizations were trivial to add, such as caching of name lookups. Handling of the FAT table was simplified by its relatively modest size. Based on our SCSI traces, and on McVoy's work on extent-based performance [9], we implemented large I/O operations.

The effects of I/O buffer caching are illustrated in Section 7, by comparing native MS-DOS and DosFs small file accesses. On a well-tuned UNIX system, cached read and write operations should perform almost at memory copy speed. For illustration purposes, we have intentionally avoided the use of any of the many native MS-DOS extensions that would perform file caching. For instance, Microsoft provides the SmartDrive optional driver which Windows 3.1 installs by default. With this proviso, read and write operations that hit in the cache are from 3 to 15 times faster in DosFs than native MS-DOS.

The benefit of delayed writes is illustrated by the **crtdel** test, where a relatively small file is created, written and deleted a number of times. The performance gain over the uncached case is up to a factor of 40. If we disable read-aheads, the elapsed time for the sequential, uncached read of a 5 MB DosFs file in 2 KB blocks increases 30%. The effective bandwidth drops from 480 KB/sec to 340 KB/sec.

The **crtdel** test reveals the effects of one interesting difference between FFS and DosFs. In FFS, directory entries and the inode table must be kept coherent in spite of potential power failures. When creating a new directory entry FFS incurs a cost of three writes. The first one is to the inode table, the second to the directory entry, and the last again to the inode table to set the reference count properly. Hard links require a proper handling of the reference count field. In DosFs we only need two writes, one to the directory entry and one to the FAT. This is still true even if we implemented hard links, because the reference count would have to be part of the directory entry itself, not of a separate inode table.

Another optimization is caching of name lookups, which also came to us for free since the name cache operates on Vnodes and is therefore filesystem-independent. The effects of this optimization can be partially quantified in the **open** tests, where a file is repeatedly opened and immediately closed. FFS and DosFs get a relatively better score on the longer pathnames than MS-DOS, but the times show that MS-DOS is also doing name caching. If we disable the namecache DosFs incurs a 15% higher cost per component. Note that in this test the read operation necessary to reconstruct the inode will always hit in the buffer cache. If the system is loaded the read might miss, and the price will be much higher.

We decided to keep the FAT entirely in memory, to speed up many file access operations. The maximum size of a FAT is 128 KB, which would describe a 512 MB partition at a cluster size of 8 KB. In practice, most MS-DOS disk partitions are much smaller than this, for instance on a floppy the FAT is approximately 4 KB and on a 60 MB disk partition of 2 KB clusters the FAT is 60 KB. In terms of main memory consumption the FFS inode cache alone is much more expensive (in our system) than the DosFs FAT table. In the Mach 3.0 system this memory is pageable. If we find that keeping the FAT in memory creates problems we can take advantage of the I/O buffer cache and reduce this memory cost to a user-specified number of I/O buffers per mounted file system. If frequently accessed, the FAT blocks will be found in the buffer cache. It is difficult to isolate the performance effects of this optimization, because the occasional I/O operations to recover the FAT would be unpredictably intertwined with the other I/O operations.

We applied other minor optimizations in the FAT handling code. For instance, every DosFs Vnode has a lookup hint that is used when mapping file offsets to disk clusters. We do not have to scan the entire allocation chain when doing sequential reads. We also optimized the rewriting of the FATs since they tended to be a noticeable cost, especially on the slow floppies. On each sync(2) call we rewrite (if modified) only the primary FAT, the alternate FATs are rewritten only when the disk is unmounted. If we encounter a power failure the status of the disk will be as of the last sync(2) call, and dosfsck takes care of rewriting the alternate FATs.

Our disk block allocation algorithm is extremely simple and similar to the one used by MS-DOS [1]. When extending a file we scan forward in the FAT starting from the last block of the file. If we do not find a free block, then we scan backward from the last block of the file. When creating a new file or directory we start from (an estimate of) the lowest numbered free block in the FAT. The first two rules tends to enforce sequentiality and clustering, the third tends to accumulate in-use blocks at the beginning of the disk, reducing seeking. The algorithm is extremely simple, and extremely effective if executed sequentially. It is not executed concurrently in MS-DOS, or for the most part, in our UNIX workstations.

The optimization that had the single most visible payoff was to perform I/O in sizes larger than the file system's cluster size. This is possible because the block allocation algorithm is very effective in keeping a file's data contiguous. We can essentially chose any size for I/O and use it as the equivalent of the FFS logical block size. The DosFs logical block size is specified by the user at mount time. The default value of 16 KB was chosen because it is the knee in the performance curve across several disks, and across three types of workstations, but we make no claims as to its generality and applicability. The implementation is simple, we use a special version of *bmap()* that returns the start and size of the contiguous chunk of disk that contains a given cluster.

One last optimization we adopted from MS-DOS: we do not force the users to discover optimizations on their own, unless they are really obvious. This is in sharp contrast with the FFS philosophy. In [8] it was advocated that users would know the characteristics of all disks and even the speed of their processors relatively to the disks they used. Users would otherwise perform experiments to define the correct values for the file system parameters, and use tunefs(1) to optimize them. There is only one tuning parameter in DosFs, the logical block size for large I/Os. This is optionally decided at mount time, has an appropriate default, and the meaning is more intuitive than most FFS parameters (and/or combinations thereof).

6. Disk Utilization

In this section we analyze the overheads and the costs for metadata in DosFs, and we compare them with FFS. We have performed a simple experiment involving the storing of a set of files on a floppy disk, and we have derived the formulas that characterize the space requirements of the two file systems.

Allocation Size	*Waste*	*FFS free*	*DosFs free*	*Difference*
512	**50 KB**	**326 KB**	**422 KB**	**+29%**
1024	**105 KB**	**273 KB**	**368 KB**	**+34%**
2048	**220 KB**	**138 KB**	**260 KB**	**+88%**
4096	**475 KB**	**-86 KB**	**4 KB**	**n/a**

Table 6-1: A Simple Utilization Test: Moving Data To A Floppy Disk

The data set included two software packages in sources, objects, documentation files and man pages for a total of 978,634 bytes in 194 files and 7 directories. The floppy media has a formatted capacity of 2880 sectors, or 1,474,560 bytes. Table 6-1 illustrates the results of unpacking the data from a tar file onto the floppy file system. Four cases each for FFS and DosFs are presented, with fragment/cluster sizes ranging from 512 to 4096 bytes. The FFS block size was kept constant, 4096 bytes. We report the amount of free space after the unpacking (a negative value indicates the amount of data that did not fit), as reported by df(1). The FFS entry at 4096 bytes is misleading since not all of the data fit on disk. DosFs leaves from 30% to 80% more free blocks than FFS, depending on the allocation size.

The first column reports the number of bytes wasted due to fragmentation. Suppose we put a one-byte file in a single directory on the floppy. We will need one disk block for the directory, and one for the file. Most of the bytes in the two blocks are wasted, in DosFs we really needed 33 bytes and we used instead 1024 bytes, or more. We computed the fragmentation by looking at each file and directory and evaluating the extra bytes on disk that were allocated and unused because data did not fill a block. The space wasted is approximately the same in the two filesystems and originates for the most part from files. This measure is important in two respects. It shows that an improper allocation block size can esily cost 40% of the available space on disk. It demonstrate that Extent Based allocation, which is equivalent to even larger allocation sizes, is an expensive proposition for typycal UNIX files.

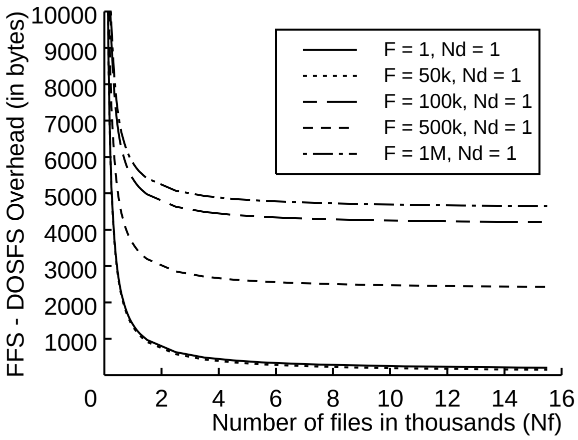

Figure 6-1: Metadata Costs: FFS Over DosFs

Let us now look more closely at the overheads incurred for meta-data in the two file systems, by comparing the total per-file disk costs for a file of length **F** in a file system of fragment/cluster size **A**. In the formulas, **ceil**(X,Y) is the function that rounds up X to the nearest Y multiple. DosFs will use

```
DosFsCost = D + (B * Ceil( F/A, 1 )) + Ceil( F, A )

D = Ceil( Nd * 32 , A ) / Nd
```

bytes on disk, where **Nd** is the number of files in the same directory and **B** is the number of bytes per cluster in the FAT, e.g. 2 for a disk and 1.5 for a floppy. The three terms of the sum represent the cost **D** in the directory entry, the FAT, and the file itself. The FAT cost is paid once at file system creation time, and with the exception of the root directory all other costs are dynamic allocations. The fixed, upfront cost for DosFs in the (best) floppy case is 2 KB, in a 60 MB disk partition this is 64 KB.

For FFS, we arrive at the following expressions

```
FFSCost = D + 128 + Ceil( F/(8 * A), 1 ) + C / Nf + Ceil( F, A )

D = Ceil(Sum(Max( 12, 8 + namelen(i))), A) / Nd
```

Where **Nf** is the total number of files on disk, and **C** is the fixed cost in cylinder groups and replicated superblocks. For a floppy this cost is at minimum 104 KB in three cylinder groups, for a 60 MB disk partition it is 1.4 MB. The five terms of the sum are the amortized directory entry cost **D**, the inode, the allocation bitmap, the amortized fixed cost, and the file itself. For FFS all but the directory entry and file data costs are paid at disk creation time. The formula above is only valid for (relatively) small files, for very large files we must also add the costs in double and triple indirect blocks.

Figure 6-1 shows what happens as we add more files to the file system. The curves assume a given file length **F**, and a fixed number of files per directory. Note that up to a file length of 64 KB FFS does not use indirect blocks to indicate the blocks that belong to files, it uses the direct block list in the inode which is a pre-paid cost. This fact is illustrated in more detail in Figure 6-2, where we vary the size of the average file in a file system with 6,000 files and an average of 20 files per directory. In both Figures the fragment size is 512 bytes and the logical block size is 4 KB, in a 60 MB partition. The ramps are the points at which FFS must allocate a new block for the (indirect) block list. Once that cost is paid DosFs has a lesser advantage, but the DosFs costs still grow at half the speed of the FFS costs. This is because FAT entries are 16 bits while FFS disk block numbers are 32 bits.

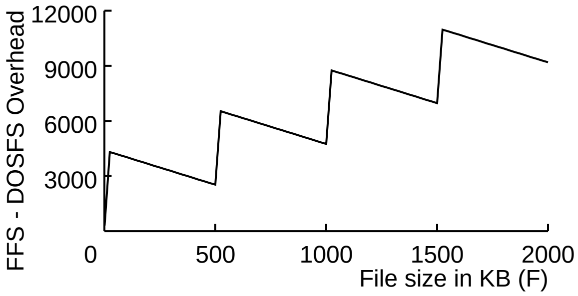

Figure 6-2: File System Overhead In Bytes, As A Function Of The File Size

If we look at directory entries alone, FFS would score better than it does overall. DosFs uses a fixed 32 bytes per entry, including information that FFS stores in the inode table. FFS instead uses a minimum of 8 bytes plus the file name, and will typically be able to fit more directory entries per block. In Figure 6-3 we plot the directory entry costs **D** for the two file systems, and their difference. By growing the number of files per directory the initial cost is amortized and the remaining cost is dominated by the file name length (we have assumed file names of length 12).

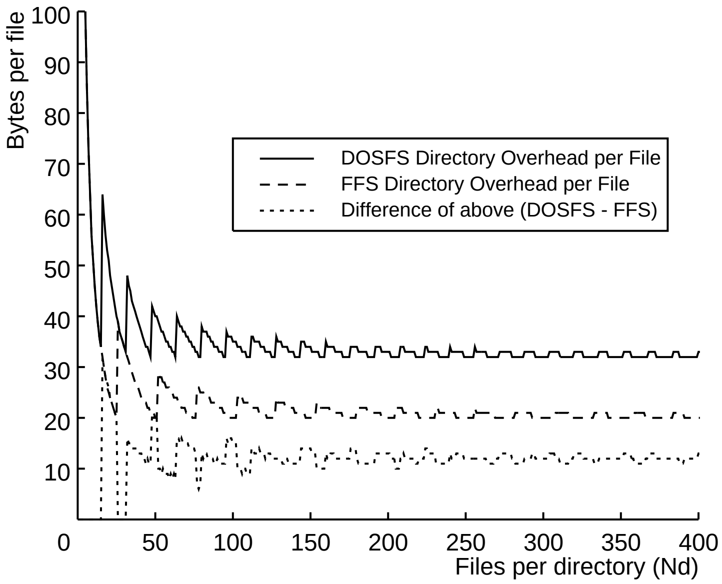

Figure 6-3: Costs Of Directory Entries

Note that the block allocation algorithms used in the two file systems have no effects on disk utilization, but they do affect performance as indicated in Section 7.

7. Performance Evaluation

John Ousterhout published, as part of the Sprite OS sources, a set of file system tests [11], which we use to summarize the performance of our file system at the micro-benchmark level. The **open** test opens (and immediately closes) the same pathname a large number of times, and reports the average time per iteration. The **crtdel** test repeatedly creates, writes to, and deletes a file. The **read** test reads from stdin in 16 KB sizes, until the end of the file. The file is then reset and the cycle is repeated at least 100 times, for timing purposes. Similarly **write** writes the given number of KB to stdout, in no more than 16 KB sizes. This is also repeated at least 100 times.

Most of the well-known macro-benchmarks are UNIX-specific and do not have an identical implementation and meaning under MS-DOS and UNIX. This is true for the Andrew benchmark [4], for which we do not have a native MS-DOS version. Similarly for the unpacking of the tar file described in Section 6; we can only compare the times to do the unpacking between FFS and DosFs. The Iostone test is a synthetic benchmark that models a timesharing load. The test was written by Arvin Park and Jeff Becker at UC Davis, and it is based on the traces John Ousterhout collected on a general-purpose machine at UC Berkeley [10]. For MS-DOS, we report on typical operations a user would perform on a large number of files, such as copying or deleting file trees.

The test machine is a Gateway 2000 486/33C running Mach 3.0 (MK83+UX42), with a 33 Mhz Intel i486 processor, 16 MB of main memory and a 400 MB Maxtor SCSI disk interfaced via an Adaptec 1540c SCSI HBA. Effective memory bandwidth is about 10 MB/sec. The disk is split into a 60 MB initial MS-DOS partition and the remaining disk space for UNIX partitions. Both the MS-DOS and FFS partitions are over two years old. The MS-DOS partition is about 40% full, the cluster size is 2 KB. The FFS partition is about 60% full, fragsize is 512 bytes, block size is 4 KB, rotdelay is 4 ms, maxcontig is 1, there are 62 cylinder groups. Using fresh file systems would lead to a more repeatable experiment, but using old file systems shows what happens to the performance numbers when the file systems age. The discrepancy between our results and the results presented in [7] seems to indicate that DosFs has a better degradation than FFS.

Test	*Native MS-DOS*	*FFS*	*DosFs*
open "foo"	**0.50 ms**	**0.75 ms**	**0.75 ms**
open "a/b/foo"	**0.82 ms**	**0.82 ms**	**0.82 ms**
open longpath	**1.16 ms**	**0.95 ms**	**0.95 ms**
crtdel 0 KB	**32 ms**	**3 ms**	**3 ms**
crtdel 10 KB	**521 ms**	**36 ms**	**20 ms**
crtdel 100 KB	**3671 ms**	**107 ms**	**87 ms**
read 1 KB	**0.1 MB/s**	**1.1 MB/s**	**1.1 MB/s**
read 8 KB	**0.5 MB/s**	**2.2 MB/s**	**2.5 MB/s**
read 100 KB	**1.2 MB/s**	**3.0 MB/s**	**3.3 MB/s**
write 1 KB	**0.1 MB/s**	**1.3 MB/s**	**1.3 MB/s**
write 8 KB	**0.5 MB/s**	**3.7 MB/s**	**3.7 MB/s**
write 100 KB	**0.5 MB/s**	**5.8 MB/s**	**7.4 MB/s**

Table 7-1: John Ousterhout’s Basic File System Tests.

The results of the micro-benchmarks are illustrated in table 7-1. DosFs is faster than FFS on cached read and write operations, but this is simply the result of the way data is moved between the UNIX server and the user process. Mach's Virtual Memory works best on large amounts of data, and in the case of DosFs it can move the entire 16 KB segment by remapping it at once. In the FFS case, data is split across 4 buffers and disjoint virtual addresses.

The lack of caching and/or prefetching in MS-DOS makes it run slower, on average, than the UNIX file systems. We modified the native MS-DOS *read* test to read different amounts of data with various blocking factors; the best bandwidth was 1.2 MB/sec when reading a 100 KB file in 16 KB blocks. In order to explain the results of the **open** test we must assume that native MS-DOS is doing some caching anyway. This was confirmed with the Mach 3.0 MS-DOS emulator.

To evaluate the performance on uncached operations we use the **read** and **write** tests above, on much larger files. The tests include both networked and local cases. Read and write performance of DosFs over NFS are nearly identical to the FFS performance, as indicated in table 7-2. The client was a Decstation 5000/200, the server was a DEC Alpha workstation, both running Mach 3.0. The user-to-user bandwidth (measured by **ttcp**) between the two machines over a UDP transport was 860 KB/sec. Both the NFS and the networking code in the UNIX server are old, and these numbers show it.

Test	*FFS*	*DosFs*	*MS-DOS Native*
Uncached read 5MB	**170**	**850**	**1200**
Uncached write 5MB	**178**	**382**	**624**
NFS read 5MB	**377**	**379**	**n/a**
NFS write 5MB	**749**	**749**	**n/a**

Table 7-2: Uncached Data Transfers, For Local Disk And Over NFS, In KB/sec

The UNIX server has disabled mapped files, therefore we incur data movement overheads that make local performance lower than in a monolithic kernel. To quantify the maximum disk performance, we wrote a program that accesses the disk directly through the Mach 3.0 kernel. This is the closest equivalent of the raw character device for traditional UNIX. This program uses multiple threads to guarantee that a request for the next sequential block is always ready in the disk driver queue by the time a request completes. This was verified with a SCSI bus analyzer. On the test machine this program obtains a 1.5 MB/sec read rate at a block size of 16 KB. Higher block sizes obtain bandwidths in excess of 1.6 MB/sec[1]. By contrast, a dd(1) of the raw device via the UNIX server only obtains 1.2 MB/sec, which indicates a 20% overhead in the UNIX emulation code along the I/O path.

In table 7-2 we indicate the results of reading or writing a 5 MB file for each of the three file systems. The buffer cache size was set at 1.3 MB and the file systems were unmounted between each of five test runs. We report the best results, but there is little variation even if the machine was running in multi-user mode on a regular Ethernet. Extra data copies make DosFs slower than MS-DOS. But the most remarkable result here is the five fold speedup of DosFs over FFS. This result can be explained in the same way as in [7]: DosFs performs I/Os on a file that is contiguous, whereas FFS allows for presumed rotational delays and interleaves blocks on the disk. Every track is only 1/4 utilized by FFS, losing most of the benefits of the track-level caching done by the disk. The speedup is a little higher than four because the FFS file system is not empty and occasionally FFS cannot allocate its blocks optimally. DosFs is still capable of reaching optimal allocation in its partition, despite the utilization.

We obtain another important result if we disable the large I/O optimizations, e.g. if we force DosFs to perform I/O at the 2 KB cluster size. The DosFs bandwidth becomes 480 KB/sec, almost three times higher than FFS which

[1]Poor cabling prevented us from making use of the faster SCSI synchronous transfer mode.

does I/O in 4 KB blocks. The disk compensates for the small access size by read-caching an entire track, but it cannot compensate for the extra seeks that FFS incurs. Adding track caches on the disk drives is quite common with SCSI disks.

Table 7-3 shows the results of our macro-benchmarks. The modified Andrew benchmark is commonly used to simulate the load of a programming environment. The test has five phases which mainly involve the creation of various directories (phase I), copying of files (phase II), doing many stat(2) over the file tree (phase III), sequentially scanning all files (phase IV) and performing some compilations (phase V). DosFs is faster than FFS in all phases, the larger gains are obtained in the phases that actually access file data, e.g. phases IV and V. Among all the tests we ran, this is the one with the least controlled execution environment. It uses scripts and system programs, for instance, which on the test machine are paged in from a different partition but on the same disk. The distance between the DosFs partition and the FFS partition from system data and code is not the same, and DosFs is penalized. To use this test effectively we would need a different test setup.

We report three cases for the Iostone test, in each one we used a different number of instances of the test, running in parallel directories. Each instance uses approximately 5 MB of disk space, to overflow the buffer cache. We selected this test to get a worse-case behavior from DosFs, because it invalidates the assumptions about a single-user load. When we checked the block allocations for the larger files we found them very scattered, as expected. But DosFs is still from 13% to 20% faster than 4.3 FFS when large I/O operations are enabled. If we disable large I/Os (times in parentheses) DosFs becomes at most 20% slower than FFS.

Test	*Native MS-DOS*	*FFS*	*DosFs*
Andrew - Phase I	**n/a**	**0.5**	**0.5**
Andrew - Phase II	**n/a**	**9**	**8**
Andrew - Phase III	**n/a**	**8**	**6**
Andrew - Phase IV	**n/a**	**12**	**9**
Andrew - Phase V	**n/a**	**49**	**40**
Iostone-1	**n/a**	**233**	**195 (287)**
Iostone-2	**n/a**	**654**	**550 (700)**
Iostone-3	**n/a**	**1089**	**963 (983)**
Unpack Tar File	**n/a**	**187**	**150**
Copy Windows	**48**	**76**	**58**
Delete Windows	**11**	**5.4**	**1.2**

Table 7-3: Results For Selected Macro-Benchmarks, All Times In Seconds

We had expected DosFs to be as much as twice as slow as FFS in the Iostone test, due to the smaller I/O size (2 KB versus 4 KB). This is not the case, and the explanation is not immediate. The majority of the files used by the test are small, only 12 of the 404 work files are larger than 16 KB, 192 files are one sector or less, and 40 files are between 4 KB and 16 KB in size. Even the larger files are not totally fragmented. Two distinct phenomena help explain the results of this test. DosFs keeps the many little files close to each other, making better use of the disk-level cache, and when using the large buffers DosFs has a higher hit rate in main memory.

We measure the unpacking of a 1 MB tar file to a floppy disk for two reasons. It is an example of a common activity in UNIX, and the floppy is extremely slow when compared to a disk and does not have any read or write caches to help performance. Here the FFS optimizations show much gain, the difference versus DosFs is no longer a factor of two as in the uncached write test but a much smaller 25%. The test involves creating many files, therefore FFS pays the extra costs discussed in the **crtdel** test above.

For the last two tests we also have native MS-DOS times. The first one is a recursive copy of a relatively large tree of MS-DOS system files, the Windows 3.1 directory. The tree is larger than the buffer cache, and the test was performed on a cold cache to guarantee pure uncached behavior. The results confirm the trend of the previous uncached micro-benchmarks, but the gap between systems is reduced. Deleting the Windows tree shows once again the advantage of delayed writes, as was partially indicated in the **crtdel** test.

There is one optimization we did not apply to DosFs which is instead present in FFS: caching of inodes. The UNIX server, for instance, uses an in-memory inode table of about 1,200 FFS inodes. Even of released (usually because of namecache spills) FFS inodes stay around and can be reused without disk accesses. None of the tests above would be strongly affected by this optimization, but for instance the traversal of the whole file system name space probably would. Caching could be done at the Vnode layer, provided it benefits all file systems.

8. Related Work

Sun provides the *pcfs* file system in its UNIX offerings, which is a read/write Vnode based file system for MS-DOS disks, like our DosFs. Next has a similar file system in the NeXTStep product. Recently there was a posting on alt.sources of a "Filesystem Survival Kit" which provides the ISO and MS-DOS file systems for System-V UNIX systems. These file systems are strictly MS-DOS specific, and offer no protection extensions, or symlinks. They do not provide performance optimizations, and cannot be used as the root file system.

The SunOS 4.1 code lets the user change floppy disks without using umount/mount. It does this by serializing all operations and performing consistency checks between in-memory and on-disk data. As a result, the code is less effective at caching. NeXTStep uses the hardware to detect a change in floppy media, and auto-mounts floppies on insertion. The use of hardware detection should enable more optimizations.

In the non-UNIX world, IBM's OS/2 offers multi-user extensions to the PC-DOS FAT-based file system which they call *extended attributes*. These can support more than just protection information; they can be used for presentation purposes for instance. Extensions by the same name are defined in the ISO 9660 standard, and serve similar purposes.

Our optimizations for uncached accesses are similar, but unrelated to the optimizations described in [9, 7] for Extent Based File Systems. DosFs adds to McVoy's work the ability to specify the extent size at mount time. The implementation of large operations is similar to the BSD 4.4 *cluster* idea, the *bmap()* changes are almost identical. But while our optimizations are specific to DosFs, the BSD 4.4 clusters are not filesystem-specific and can be used by any file system code. We call our *bmap()* only from within DosFs code, in 4.4 we would call the DosFs *bmap()* from the filesystem-independent clustering code.

9. Conclusions

With DosFs we have demonstrated that it is possible to provide full UNIX semantics in a file system with simpler data structures than traditionally used in UNIX file systems. Removing inodes provides better disk utilization, better disk/power failure behavior, and eliminates many synchronous write operations in the handling of meta-data. The only cost is a more cumbersome and slow implementation of hard file links.

In the implementation of DosFs we have used a new block allocation algorithm, derived from MS-DOS [1]. The algorithm obtains both contiguous allocation of disk blocks and compact disk utilization. DosFs is both faster than 4.3 BSD FFS and more efficient in disk space. The performance gains are equivalent to Extent Based allocation, buth without the high fragmentation costs. DosFs allocates small blocks, but close to each other. In this way it can take advantage of disk-level read and write caches, and obtains a three fold performance improvement over FFS.

Moreover, DosFs takes advantage of the contiguous block allocation to perform large size I/O operations and obtains a five fold speedup over FFS.

An important factor in the DosFs performance profile is the effect of seek times, or more precisely the time spent by the disk to *service* a given I/O request. Today's disks use a variety of multi-track caching strategies that make the old FFS model [8] unrealistic. Unfortunately, the service time has become very difficult to model. In our SCSI analyzer traces we observed large variations in service times, anywhere from 3 milliseconds to 45 milliseconds, on the same disk, and not at all a simple function of the distance between disk blocks. When the disk drive uses a write cache, not only the distance, but the type of access and previous history affects the service time. Finally, the zoning techniques used in recent drives add another variable to the already complicated model. All of these factors contribute to make DosFs' performance better than FFS, even in cases where we did not expect it.

The only deficiencies in the MS-DOS permanent data structures are in handling large files and large disks. Large files create two problems, both related to the choice of a linked list to represent the clusters of a file. In the first place, is it is impossible to represent files with holes, all clusters must be allocated by DosFs and the "holes" truly must be zeroed for security reasons. The programmer does not have to issues the intervening writes, but there is no way to indicate in a FAT entry that the block ought to be zeroed when accessed. FFS uses a tree as its data structure, and can easily mark the holes as missing blocks, to be zeroed on access. But files with holes are not commonplace, and a copy of the file will fill the holes anyway. The second problem is we must traverse the whole list to get to the last cluster of a file. We have argued that it is feasible to cache the entire FAT in memory so that the cost is just in memory accesses, and much less than the case where FFS must actually fetch an indirect block from disk.

Large disks create the problem of representing cluster numbers, which are currently 16 bit integers in the permanent data structures. It is possible to use a new FAT type, with 32 bit entries. It is not possible to transparently change the format of the directory entries to hold a 32 bit starting cluster instead of the 16 bit one. A temporary solution can be to grow the cluster size, but this can only gain a few more bits, if the cluster size becomes 32 KB the fragmentation (most MS-DOS files are small) is unacceptable. If we ignore MS-DOS compatibility this problem disappears.

9.1. Status and Acknowledgements

The code described in this paper has been integrated in the CMU sources (releases UX42/43), and is available for general use at CMU and elsewhere. Our shepherd Margo Seltzer went way beyond the call of duty in helping us put the paper in shape. Mike Dryfoos, Mahadev Satyanarayanan, Robert Baron, Daniel Stodolsky, Howard Gobioff and Peter Dinda gave us many helpful comments on earlier versions of this paper. Thanks for her patience, and good luck to Kelly.

References

[1] Ray Duncan.
Advanced MSDOS Programming.
Microsoft Press, 1986.

[2] David Golub, Randall Dean, Alessandro Forin, Richard Rashid.
Unix as an Application Program.
In *Proceedings of the USENIX Summer Conference*, pages 87-95. USENIX, Anaheim, CA, June, 1990.

[3] David Golub and Richard Draves.
Moving the Default Memory Manager Out of the Mach Kernel.
In *Proceedings of the USENIX Mach Symposium*, pages 177-188. USENIX, Monterey, CA, November, 1991.

[4] Howard, J. et al.
Scale and Performance in a Distributed File System.
ACM Transactions on Computer Systems 6(1), February, 1988.

[5] ISO 9660 : 1988 (E).
Information processing - Volume and file structure of CD-ROM for information interchange.
International Organization for Standardization, 1988.

[6] Kleiman, S. R.
Vnodes: An Architecture for Multiple File System Types in Sun UNIX.
In *Proceedings of the Summer 1986 USENIX Conference,* pages 238-247. June, 1986.

[7] Margo Seltzer, Keith Bostic, Kirk McKusick, Carl Staelin.
An Implementation of a Log-Structured File System for Unix.
In *Proceedings of the USENIX Winter Conference*, pages 201-220. USENIX, San Diego, CA, January, 1993.

[8] Kirk McKusick, William Joy, Sam Leffler, R. Fabry.
A Fast File System for UNIX.
ACM Transactions on Computer Systems :181-197, August, 1984.

[9] Larry McVoy and S. Kleiman.
Extent-like Performance from a Unix File System.
In *Proceedings of the USENIX Winter Conference*, pages 33-44. USENIX, Dallas, TX, January, 1991.

[10] John Ousterhout, et al.
A Trace-Driven Analysis of the UNIX 4.2 BSD File System.
In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 96-108. December, 1985.

[11] John Ousterhout.
Why Aren't Operating Systems Getting Faster As Fast As Hardware?
In *Proceedings of the Summer 1990 USENIX Conference*. June, 1990.

[12] Richard Rashid, Gerald Malan, David Golub, Robert Baron.
DOS as a Mach 3.0 Application.
In *Proceedings of the USENIX Mach Symposium*, pages 27-40. USENIX, Monterey, CA, November, 1991.

[13] Rock Ridge Interchange Protocol, Version 1.
An ISO 9660:1988 compliant approach to providing adequate CD-ROM support for Posix file system semantics.
Rock Ridge Technical Group, 1991.

[14] Satyanarayanan, M.
Integrating Security in a Large Distributed System.
ACM Transactions on Computer Systems 7(3), August, 1989.

Dr. Forin is a Research Computer Scientist at Carnegie Mellon University, working on operating systems. He received the B.S. degree in Electrical Engineering in 1982 and the Ph.D. degree in Computer Science in 1987, both from the University of Padova, Padova, Italy.

Gerald R. Malan is a Research Programmer at Carnegie Mellon University, and has been working on the Mach project for the last four years. His research interests include multiple operating system personalities, remote debugging, file systems, and object oriented system composition. Mr. Malan received his B.S. in Computer Engineering from Carnegie Mellon University in 1990.

Table of Contents

List of Figures

List of Tables