DMFS – A Data Migration File System for NetBSD

William Studenmund
Veridian MRJ Technology Solutions
*NASA/Ames Research Center**

Abstract

I have recently developed DMFS, a Data Migration File System, for NetBSD[1]. This file system provides kernel support for the data migration system being developed by my research group at NASA/Ames. The file system utilizes an underlying file store to provide the file backing, and coordinates user and system access to the files. It stores its internal metadata in a flat file, which resides on a separate file system. This paper will first describe our data migration system to provide a context for DMFS, then it will describe DMFS. I also will describe the changes to NetBSD needed to make DMFS work. Then I will give an overview of the file archival and restoration procedures, and describe how some typical user actions are modified by DMFS. Lastly, I will present simple performance measurements which indicate that there is little performance loss due to the use of the DMFS layer.

1 Introduction

NAStore 3 is the third-generation of mass storage systems to be developed at NAS, the Numerical Aerospace Simulation facility, located at NASA/Ames Research Center. It consists of three main parts: the volume management system (volman), the virtual volume management system (vvm), and the DMFS (data migration file system) system.

A complete description of the NAStore 3 system is beyond the scope of this document. I will give a brief overview, and then descrive DMFS's role in NAStore 3.

1.1 volman system

The volman system is responsible for keeping track of the tapes, and for bringing them on line when requested. It was designed to support the mass storage systems deployed here at NAS under the NAStore 2 system. That system supported a total of twenty StorageTek NearLine tape silos at two locations, each with up to four tape drives each. Each silo contained upwards of 5000 tapes, and had robotic pass-throughs to adjoining silos.

The volman system is designed using a client-server model, and consists of three main components: the volman master, possibly multiple volman servers, and volman clients. The volman servers connect to each tape silo, mount and unmount tapes at the direction of the volman master, and provide tape services to clients. The volman master maintains a database of known tapes and locations, and directs the tape servers to move and mount tapes to service client requests. The client component consists of a set of programs to monitor activity and tape quotas, and a programmatic interface so that programs can make requests of the volman master and servers.

1.2 vvm system

The vvm system provides a virtual volume (or virtual tape) abstraction and interface for client programs. Its main utility is to improve tape usage efficiency. Many streaming tape technologies do not handle writing small files efficiently. For instance, a seven kilobyte file might take up almost as much tape space as a one megabyte file. To better handle such cases, vvm clients (such as the DMFS system) read and write files out of and into virtual volumes (vv's), which are typically between 50 and 300 megabytes in size. These vv's are read and written to tape. Thus all tape operations involve large reads and writes which are much more efficient.

The vvm system consists of a server, client programs, and a tape server daemon. The vvm server is responsible for remembering which vv's have been allocated, and on which tape volume they reside. When a vvm client requests a vv, the vvm server contacts the volman system (the volman master specifically) to arrange for the needed tape volume to be mounted. The vvm tape

*Present address: Zembu Labs, 445 Sherman Avenue, Palo Alto CA 94306, wrstuden@zembu.com

server daemon, which runs on the same machine as the volman tape servers, then reads the tapes and makes the vv available to the client. The clients obviously use the vv's to store variable-length files. There are client programs which are able to monitor the status of vv's, and there is a programmatic interface which lets programs make requests of the vvm system.

1.3 DMFS system

The final component of the NAStore system is the DMFS system. This consists of *dmfsd,* the DMFS data migration daemon, system utilities to assist file migration, and the DMFS file system (the focus of this paper). *dmfsd* is the data migration daemon, and it runs on each host supporting a DMFS file system. It is responsible for responding to requests from the DMFS file system for file restoration. The userland system utilities in the DMFS system are responsible for archiving files and for making them non-resident. There are also utilities to permit users to force the archival or restoration of specific files.

2 Description of DMFS

The DMFS file system is the part of NAStore 3 with which most end-user interaction happens. It maintains as much of a traditional UNIX user experience as possible while providing data migration services.

2.1 Layered file system

One of the main differences between DMFS and other migration file systems (such as RASHFS, the NAStore 2 file system, or HighLight[3]) is that it is a layered file system. These other file systems merge the data migration code into the file system code, while DMFS uses the layered file system formalism to place its migration code between most kernel accesses and the underlying file system.

We perceived two main advantages and one potential disadvantage with this design choice. The first advantage we perceived is that it would be much easier to create and maintain a layered file system with data migration facilities than to add those facilities to a specific file system and maintain them over the long term. For instance, the Berkeley Fast File System (FFS)[4, 5] is undergoing change with the addition of soft updates[7]. By maintaining the file migration functionality separately, we decouple DMFS from any other FFS-related development. Secondly, we leave the choice of underlying file system to the site administrators. If the access patterns of a site are better suited to the Berkeley Log-structured File System (LFS)[4, 6], then a site can use that as the underlying file store rather than LFS. The only limitations are that the underlying file system support vnode generation numbers, and the current metadata database requires knowledge of the maximum number of vnodes at the time of initial configuration.

The one potential disadvantage is that the use of layered file systems incurs a certain amount of overhead for each file system operation. In [2], Heidemann measures layered file system overhead to be on the order of a few percent. Section 7 will describe the limited testing we have done to date. The conclusion we have reached is that any performance penalties due to layered file system technology are certainly worth the benefits in maintainability and flexibility.

2.2 Functions

The primary function of the DMFS file system layer is to provide user processes transparent access to migrated data. It determines if an operation would access non-resident portions of the underlying file. If so, it requests that the *dmfsd* daemon restore the file, and then blocks the operation until the file is sufficiently restored so that the operation may proceed. One feature present in DMFS is that when a process is blocked awaiting file restoration, it may be interrupted. As a practical matter, a user may kill (ˆC) a blocked process.

Another main function of the DMFS layer also is to preserve the integrity of the metadata it and the userland *dmfsd* databases keep regarding files. It does so in two ways.

First, it records the generation number of the underlying vnode. When an operation (such as name lookup) causes the allocation of a DMFS vnode, the recorded generation number is compared with the generation number of the underlying vnode. In case of a discrepancy, the information in both the kernel and userland databases is invalidated. Likewise these databases are invalidated when the last reference to an unlinked vnode is released.

The second area of metadata preservation keeps a file from getting in an inconsistent state. The present metadata format permits recording that a portion of an archived file is resident, but does not permit noting that only a portion of a file is archived. The possibility of file inconsistency would arise if the on-disk portion of a non-resident file were permitted to be modified. For simplicity, such cases are forbidden at present. Thus any writes or truncations to a non-resident file are blocked until the file is fully restored.

2.3 Metadata

The DMFS layer keeps a small amount (80 bytes) of metadata for each vnode in the underlying file system. At present these data are kept in a flat file database indexed by vnode number which is stored on a separate file system. The metadata file contains a header which records the version number of the metadata file and of the nodes, their size, and the byte order in which they are stored. The metadata include:

flags
: Flags indicating state of the DMFS node, including: metadata valid, file is archived, and file is (partially) non-resident.

generation number
: The generation number of the current file using this vnode number.

archive size
: The size of the file (including any non-resident portion).

byte barrier
: How much of a non-resident file is present on the underlying file store. Reads of non-resident files which access only data that is present on the underlying file store are not blocked.

bfid
: "Binary File IDentifier" a 16-byte endian-independent tag assigned to the file which is intended to uniquely identify it. The *dmfsd* databases use this tag as the main index for the file.

atime and **mtime**
: Access and modification time stamps (with nanosecond resolution) for the file. These times are not updated due to either archive or restore agent operation.

archive time and **restore time**
: Time stamps (with second resolution) of last archive and restore events.

For files with valid metadata, the **atime** and **mtime** values above are reported in a VOP_GETATTR() operation, masking archive and restoration agent activity from view by users. When a file is flagged as non-resident, the **archive size** value is also reported in a VOP_GETATTR() operation. These elements together serve to preserve a more traditional user experience with files on the DMFS layer. The activities of the archive and restore agents are not noticeable by userland processes such as `ls -l`, `find -amin`, `find -mtime`, etc., other than as described in section 3.9[1].

The archive and restore time stamps are recorded to help the policy engine in the userland migration agent know more of a file's access history.

2.4 Comparison with other projects

Some of the design goals of the DMFS layer are best understood in terms of some of the data migration efforts which have come before it.

The DMFS layer is based on NASA's experience with the RASHFS file system used in the NAStore 2 system. NAStore 2 was deployed at NAS from 1991 until June, 1999. It was initially deployed on an Amdahl 5880 running UTS. Later it was deployed on two Convex 3820 computers running ConvexOS. RASHFS was a modified version of the native FFS file system implementation. The changes needed to support data migration were grafted into the FFS implementation, extra space in the inode was used to store metadata, and a few system calls were modified to call into the RASHFS implementation to ensure that a file was restored before proceeding. (A file had to be fully restored before it was executed.) This implementation was very successful. The system deployed at NAS had about 4 TB of disk in the file systems and a couple of petabytes of on-line tape.

One of the difficulties with the RASHFS system is that it basically was RASH-FFS. To add the same migration abilities on top of a different file system type would require a substantial amount of re-implementation. As there is less unused space in FFS inodes in modern (4.4BSD and later) FFS implementations, it is possible that the metadata will no longer fit. This is one of the main reasons we went with a layered file system when designing DMFS. While untested, DMFS should work with NetBSD's LFS, and possibly even EXT2FS file systems, about as well as with the FFS file system. Additionally, at the same time as DMFS was being devel-

[1] the fact that accessing non-resident portions of a file will block until the needed tape can be mounted and read. That will depend on the tape silo robotics, and, for tapes in off line storage, a human finding and mounting the needed tape. This delay, especially for off line storage, can be on the order of tens of minutes.

oped, there was a tremendous amount of file system development going on within the *BSD community. Kirk McKusick was working on adding "soft updates" to FFS, a technology which works to give FFS the resiliency of a journalized file system without having to have a journal. Konrad Schroder has been working to make NetBSD's LFS implementation quite robust and functional. By not integrating DMFS into a file system, DMFS stays independent of the above technology progressions – a site could choose to follow them or not, using the same DMFS code either way.

NAStore 3 is not the first tertiary storage system built on top of 4.4BSD. One predecessor was HighLight, a migration file system built above LFS. The major difference between it and DMFS is that HighLight is like RASHFS in that it represents melding data migration into a file system in a very intimate manner. However, HighLight uses certain aspects of the LFS structure to its advantage. LFS operates in terms of written segments[4, 6]. A file is described as having contents in specific segments. HighLight implements data migration by including in the file system block space (where these segments may reside) the tapes in the robotics assigned to the file system. Thus when a file is migrated to tape storage, the inode is modified to point to the tape block. When a migrated file is read, portions of the on-tape segments are read onto disk. Otherwise, the file system behaves much as a non-migrating LFS. One advantage of this methodology is that it is possible to create dense files larger than the disk storage allocated to the migration file system (something not possible with DMFS).

The DMFS layer does have a few advantages over HighLight. One is that it has no knowledge of where migrated files are stored. As mentioned above, HighLight marks the blocks stored on the tape as part of the file system. Thus it requires knowledge of how many tapes of which size (in which robots) are available at file system creation time. While it certainly would be feasible to extend this knowledge base (add new tape robotics for instance) for an operational file system, keeping this knowledge separate from the migration file system is simpler. Additionally, by decoupling the tape storage information, we permit more sophisticated tape policies. For instance, NAStore 3 will by default archive a file into two independent vv's (which will reside on two separate tapes) before making it non-resident (deallocating its blocks on disk). This behavior permits a level of redundancy not readily obtained with HighLight.

3 NetBSD vfs changes

A number of changes were required to the NetBSD vnode and vfs interfaces in order to get DMFS to work. All of these changes were designed to be applicable to all file systems, not just DMFS.

3.1 The OVERLAY file system

One change was the addition of a new file system, the OVERLAY file system. The OVERLAY file system is functionally similar to the NULLFS[4] file system except that it does not export the directory hierarchy into another part of the file name space. It places itself between the file name space and the overlayed file system[2]. It is intended for certain specific circumstances, such as that of DMFS, where the layered file system has a strong need to block access to the underlying file system. DMFS does so to prevent metadata inconsistencies. One other variant of the OVERLAY file system might be a file system which attempts to validate executables with a checksum before permitting their execution. This file system would need to block access as that is very key to its security model. Unless there is a strong need for interposition, all future layered file systems should be based on the NULLFS layer rather than the OVERLAY layer as it simplifies the fcntl(2) interactions described in section 3.8.

3.2 Layering works

The most dramatic change needed to get DMFS to work was to get layered file systems working with NetBSD. The layered file system research of Heidemann [2] describes many of the advantages of layered file systems. When we began this work, we found the layered file system implementations present in all *BSDs not to be robust enough for production work. For instance, the NetBSD 1.4 NULLFS certainly will replicate a file hierarchy in another portion of the system's file name space. However simultaneous access of this replication by multiple processes, for instance a parallel make invoked with `make -j 8`, would result in kernel locking panics. Obviously a production-quality migration system

[2]file system mounted on `/home`, and an OVERLAY file system subsequently mounted on `/home`. Access to files under `/home` in the file name space (`/home/user1/.profile` for example) now go to the OVERLAY file system. It in turn can pass these to the underlying file system initially mounted on `/home` as it needs to. Contrast this to the case of mounting a second leaf (non-layered) file system onto `/home`. In this case, no operations would be passed to the file system initially mounted on `/home` until the second file system was unmounted.

needed a more robust implementation.

The changes described below, especially those of sections 3.3 and 3.4, were implemented at the same time, in addition to a modification to how null_lookup() behaved[3]. As such, it is not clear which change (if any) was singularly responsible for the increase in robustness, though I suspect the change to how null_lookup() behaves represents most of it.

After these changes were implemented, I performed a few simple tests to ensure their effectiveness. The first one was to perform a multi-process kernel make in a directory on a NULLFS file system (`cd` into a kernel compile directory and type `make -j 8`). Before these changes, such an action would promptly panic the kernel. After these changes, it did not. Additionally, simultaneous access of multiple NULL layers and the underlying layer, such as the parallel make above combined with recursive finds in the other layers, have not been observed to panic the system. The DMFS layer we have built based on this NULLFS layer has shown no difficulties (panics or otherwise) due to simultaneous multi-user access in almost one year of operation.

It should be noted that the NULLFS and UMAP layered file systems have benefited from these changes, while the UNION file system has not. It is still not considered production quality.

3.3 Most file systems do real locking

To better support the vnode locking and chaining described in the next section, all file systems (except for UNIONFS and NFS) were changed to do vnode locking. Previously only file systems with on-disk storage actually did vnode locking, while the others merely returned success. As of NetBSD 1.5, the NFS file system is the only leaf file system which does not do real vnode locking. This defect remains as lock release and reacquisition during RPC calls to the NFS server has not been implemented. As mentioned above, the UNION file system has not been updated with these locking changes.

While it is not likely that one would want to layer a file system above many of the file systems which gained true locking (such as the KERNFS layer), this change makes it easier to impliment layered file systems. Lock management is one of the keys of getting layered file systems to work. By being able to rely on all leaf vnodes doing locking, the layered locking becomes much easier. Additionally the changes to add this support were not difficult.

3.4 New vnode lock location and chaining

One change made in how vnodes are locked was to implement vnode lock chaining similar to what Heidemann described in [2]. After this change, each vnode contains a pointer to a lock structure which exports the lock structure used for this vnode. If this pointer is non-null, a layered file system will directly access this lock structure for locking and unlocking the layered vnode. If this pointer is null, the layered file system will call the underlying file system's lock and unlock routines when needed, and will maintain a private lock on its vnode. As described in the preceding section, all leaf file systems other than NFS now do locking, and thus export a lock structure. Once NFS has been fixed, the only file systems which should not export a lock structure would be layered file systems which perform fan-out (such as the UNIONfile system) as they need to perform more complicated locking.

As all file systems should be doing locking and exporting a pointer to a lock structure, I decided to add a lock structure to the vnode structure and remove it from all of the leaf file systems' private data. I was concerned about a whole stack of vnodes referring to file system-specific private data, and felt it cleaner for vnodes to refer to memory contained in vnodes. In retrospect (and having completed the change), it now seems wiser to leave the lock structure in the leaf file system's private data and just require the file system be careful about managing the memory it exports in the lock structure pointer in its vnode. This change would improve memory usage in a system with multiple layered file systems by not allocating a lock structure in the layered vnodes which would go unused.

The effect of this change is that a whole stack of vnodes will lock and unlock simultaneously. At first glance this change seems unimportant, as any locking operation can traverse a vnode stack and interact with the underlying leaf file system. The advantage is three-fold. One advantage is conceptual. By having all layers use the same lock structure, the commonality of the stack is reinforced. Secondly, layered nodes do not need to call the underlying file system if the lock structure has been exported – it may directly manipulate it itself. This lack of stack traversal becomes quite advantageous in the case of multiple layered file systems on top of each other.

[3] directly rather than using the bypass routine. It also checks for the case of a lookup of ".", where the returned vnode is the same as the vnode of the directory in which the lookup was performed and handles it explicitly.

The third advantage is due to the lock semantics of the lookup operation on "..". To avoid deadlock, directories are locked from parent to child, starting with the root vnode. To preserve this when looking up "..", the child directory must be unlocked, the parent locked, and the child then re-locked. When looking up ".." in a layered file system, with a unified lock, the leaf file system can unlock the entire stack and re-lock it while trying to obtain the parent node. If the locks were not unified and there were separate locks in vnodes stacked above the leaf one, the leaf file system would either need to somehow unlock and re-lock those locks during the lookup of ".." or to run the risk of deadlock where one process had the upper lock(s) and not the lower, while another had the lower and not the upper.

3.5 New flag returned by lookup: PDIRUNLOCK

One other change has been to plug a semantic hole in the error reporting of the lookup operation. In case of an error, the lookup operation in a leaf file system is supposed to return with the directory vnode in which it was looking locked. One potential problem with this is that it is possible that the error being returned resulted from not being able to reacquire the lock on the examined directory when looking up "..". In this scenario, the lookup routine has little choice but to return an error with the examined directory unlocked. It signals this behavior by setting the PDIRUNLOCK flag which is returned to the caller. When a layered file system is maintaining its own parallel locks (if the underlying file system did not export a lock structure), the layered file system must adjust its locks accordingly.

3.6 "layerfs" library added

Before NetBSD 1.5, most NetBSD layered file systems other than UNIONFS were based on copies of NULLFS. Typically the NULLFS files were copied and renamed, the routine names were changed (the "null_" prefix changed to reflect the new file system name), and then new features were added. This behavior represents a duplication of code. In order to reduce this duplication, there is now a library of common files, "layerfs," which provide most of the NULL layer functionality. For instance the NULL layer now consists of a mount routine and vnode interface structures. The rest of the routines are in the layerfs library. The UMAP layer now shares most all of these routines, with the only difference being that it has some customized routines (bypass, lookup, and a few others) which perform its credential mapping. DMFS also uses this library of routines. Of the 19 vnode operations handled by DMFS, 20% consist solely of calls into this library.

3.7 File Handles usable in the context of a local filestore

When communicating with the userland daemon *dmfsd*, DMFS uses file handles to refer to specific files. It does this because on all of the occasions where it needs to communicate with *dmfsd* (for instance in a VOP_READ() attempting to access non-resident data) none of the potentially multiple paths to this file are available.

To make file handles truly useful in this manner, two changes were made to the kernel. First, three new system calls were added: fhopen(2), fhstat(2), and fhstatfs(2). For security reasons, all three calls are restricted to the superuser. fhopen(2) is similar to open(2) except that the file must already exist, and that it is referenced via a file handle rather than a path. fhstat(2) and fhstatfs(2) are similar to lstat(2) and statfs(2) except that they take file handles rather than paths.

The other change modified the operation of file handle to vnode conversion. In 4.4BSD, the only use of file handles was with network-exported file systems, specifically by the NFS server module. For convenience, the VFS operation which did file handle to vnode conversion also did foreign host export credential verification. Obviously that combination is not appropriate for a tertiary storage system. So I changed the VFS_FHTOVP() operation to just do the file handle to vnode conversion, and added a separate VFS_CHECKEXP() operation to handle the export verification.

3.8 VOP_FCNTL added

One feature which was needed by *dmfsd* and the other utilities was an ability to perform arbitrary operations, such as start archive, finish restore, etc., on the file(s) which it was manipulating. At the same time as we were addressing this need, there was a desire to add the ability to add the capability to perform file/inode operations. One such example is adding the ability to manipulate access control lists on file systems which support them.

These operations would be similar at the VFS level to ioctl(2)s except that they would always reach the file system, rather than possibly being dispatched to device drivers as is the case for ioctl(2)s.

```
Arw-r--r-- 1 wrstuden mss 2202790 Sep 14 15:14 Inside_AT.pdf
arw-r--r-- 1 wrstuden mss    7537 Nov  9 17:27 dmfs.h
-rw-r--r-- 1 wrstuden mss   36854 Dec  6 15:28 ktrace.out
```

Figure 1: Sample directory listing

While there was no objection to adding this extension to the VFS interface, the form of its programmatic interface generated controversy and much discussion on NetBSD's kernel technology email list. The desired interface would consist of a file descriptor, a command code, and a data argument (`void *`). There were three options: add a new system call with this parameter signature, overload the fcntl(2) interface, or overload the ioctl(2) interface.

In the end, I opted for using the fcntl(2) system call. The main reasoning is that we should allocate the entire command space reserved for the new vnode operation now – it would not be advisable to reserve some now and then add more later. There are far fewer fcntl operations than ioctl operations in NetBSD, and they are less frequently added, so it is less likely that using fcntl(2) as the programmatic interface and reserving a sizable command space now will impede future kernel development. Additionally, in the case of overlay-type layered file systems, different layered file systems will need to choose unique codes, and pass codes they do not understand to underlying file systems, so that tools designed to operate on one particular file system type will operate regardless of the depth of layered file systems. This requirement is easier to satisfy with a spacious reservation of command space.

The fcntl(2) system call takes an integer as its command code. If the most significant bit is set, the operation is now considered a request for a file system-specific operation – a VOP_FCNTL() call. This division leaves approximately 2ˆ31 commands available for traditional fcntl-type operations. The remaining 31 bits encode whether a value or a structure are read into or out of the kernel (3 bits), the size of data so transferred (12 bits), and the actual command code (16 bits). Half of the command space (32768 commands) is reserved for NetBSD use, while the remaining space is for a file system's private use.

3.9 New ls -l reporting

One user feature added to support the data migration system is a set of additional flags which indicate archive state and which are displayed via the `ls -l` command. Files typically have a "-" in the left-most column. With this change, an archived file has an "a", and an archived, non-resident file is indicated with an "A". This change was implemented by increasing the functionality of the strmode(3) subroutine, and by adding two extra flags to the mode value returned by a stat(2) call. For instance in the directory listing shown in Figure 1, it is clear that the file `dmfs.h` has been archived, and the file `Inside_AT.pdf` has both been archived and also been made non-resident.

These flags are available for use by all file systems which possess the concept of file archival attributes. For instance, the NetBSD FAT (MS-DOS) file system implementation has been extended to indicate archival state using this mechanism.

4 Archival

There are two ways for an archival to be initiated. One is for a user to request a file be archived (and possibly made non-resident) using the forcearc(8) utility. Another would be for an archival scan process to determine that the file should be archived. After opening the file, the first step is for the archiving process to perform the DMFS_SETAIP fcntl(2) operation, which sets the internal DM_AIP (archive in progress) flag. This operation will succeed if the process has root privileges and if no other process is in the process of archiving the file. The process ID for the file is noted for reference.

The second step of the archival process is for the archiver to add the file to the *dmfsd* databases, and to copy the file to tape storage. Adding the file to the *dmfsd* databases might also involve initializing some of the metadata fields such as the **bfid**, the unique identifier for this file.

The third step is for the archiving process to set the file's DM_ARCH flag, which indicates that the file has been archived. The DMFS_SETARCH fcntl(2) operation sets this flag.

Finally, if the file is to be made non-resident, the archiver performs the DMFS_SETNONR fcntl(2), which takes the amount of initial data which are to remain on the underlying file system. The DMFS layer determines the size of the underlying file, and, assuming it is greater than the requested size, it notes the underlying file size in the **archive size** field, and truncates it to the desired length. The DM_NONR flag is set for the file which indicates that it is not totally resident. From this point, the file size reported will be that which was noted during this operation rather than the actual size of the remaining portion on disk.

NAStore 2 truncated all files to zero residency. In NAStore 3 we have added the ability to retain a portion of the file on disk. From our analysis of some new storage architectures, such as MCAT/SRB, files will actually be managed by storage agents rather than directly by the generating program. Often such systems will add a header of metadata to the beginning of the file which serves to describe the whole file. By leaving this portion on disk, we permit whatever agent is managing the storage of these data to examine the file's header without triggering a restore event. As the exact amount of storage to leave on disk is a dependent on site and storage broker configuration, we permit the archiving process to determine how much of the file should remain.

5 Restoration

Like archival, restoration can also be triggered in one of two ways. The most common method is for a user's access of a file to trigger the restore, which is performed by the *dmfsd* daemon. Another method is the use of the frestore(8) program, which performs the restoration directly.

In either case, the restore agent opens the file, using either the fhopen(2) or open(2) system calls, with the O_ALT_IO flag set. This flag requires root privileges, and will permit the process to access the underlying file directly, rather than being blocked when accessing a non-resident portion.

The next step is for the restoration process to set the DM_RIP (Restore In Progress) flag using the DMFS_SETRIP fcntl(2). As with the DMFS_SETAIP fcntl(2), this will only succeed if no other process is restoring the file.

Then the restore agent reads in the relevant virtual volumes, and writes the non-resident portions of the file to disk. As the file descriptor was opened with the O_ALT_IO flag set, the write(2) calls will not be blocked and will restore the file on the underlying file store.

At various points through the restore, the restore agent may perform a DMFS_SETBBOUND fcntl(2). This operation takes an `off_t` argument and adjusts the **byte barrier** to this new value. Typically, as the file is restored, this operation is used to move the **byte barrier**, permitting blocked reads to complete as the file is restored.

The final step for restoration is for the restore agent to perform the DMFS_FINISHARC fcntl(2) and then close the file. This fcntl(2) call will set a flag such that when the file descriptor is closed, the file is marked fully resident and all processes waiting for the restoration to complete will be unblocked.

Finishing the restore is done as a two-step process to better support executing non-resident files. Any process attempting to execute the file will be blocked until after the close operation of the restore agent is completed, ensuring that the file will not be noted as being opened for write while attempting to execute it.

6 Behavior of typical operations

The above sections have described how the DMFS support processes interacted with the DMFS layer. This section describes how typical user process operations interact with the DMFS layer.

6.1 read(2)

The read(2) operation is intercepted so that an attempt to read a non-resident portion of a file is blocked until either the data are restored, or the restoration fails. A failed restoration returns an EIO error to the calling process.

The behavior of the restoration-checking routine is fairly simple. If no restoration is in progress, a message is sent to the *dmfsd* daemon requesting a restore. Then, if the operation was initiated by the NFS server subsystem[4], the EAGAIN error is returned. For an NFSv3 mount, the NFS subsystem will return the NFSERR_TRYLATER error code[5].

Then the routine enters a loop. It flags that it is waiting,

[4] detected by the presence of the IO_KNONBLOCK flag to the read, a flag set only by the NFS subsystem.

[5] this error code, its NFS client does not. Thus NFS exporting a DMFS layer to NetBSD clients will result in access to non-resident files returning errors to the application. Other NFS clients, such as the Solaris NFS client, will block the access and periodically retry it.

	Without DMFS	With DMFS
Creating with 10,000 64k writes	72 ± 1	71 ± 1.4
Overwriting with 1000 1024k writes	116	116
Overwriting with 100,000 8k writes	88.4 ± 1.7	88.75 ± 0.5

Table 1: Average operation times with standard deviations (seconds)

unlocks the vnode, and sleeps. The sleep is interruptible, and if it is interrupted the system call will be retried. This behavior permits users to abort a process waiting for a restoration.

Upon re-awaking, the vnode is re-locked and the file state is re-examined. If the restoration failed, EIO is returned. If the blocked read can now progress, it is passed down to the underlying layer.

For all read operations on nodes with valid DMFS metadata, the completion of the read operation is noted in the **atime** metadata field.

6.2 write(2)

The write(2) operation is intercepted in much the same manner as the read operation, and it calls the same restoration-checking routine. The two main differences are that the write operation is blocked until the file is completely restored, and that the completion of a write operation on an archived file triggers a message to the *dmfsd* daemon that the file archive has been invalidated.

As mentioned above, the write operation is blocked until the file is completely restored because the metadata format keeps archived state for the whole file, not subsections of it. Were a restoration to fail after a write modified part of an archived file, portions of the file would still be archived and non-resident (needing to be restored from tape), and other portions would be unarchived (and thus must not be restored from tape). This limitation was not considered significant for NAStore 3's target applications.

6.3 truncate(2)

Like writes, truncate operations invalidate the archived copy of the file and are intercepted, and may be blocked while file data are being restored. If the file is shrinking so that all of the new size is presently resident, the truncate operation proceeds. Otherwise a file restoration is triggered. Unfortunately, at present there is no way to request that only a portion of a file be restored, so the whole file is restored and then truncated.

7 Performance

One drawback to the layered file design model is that it adds a certain amount of overhead to each file operation, even if the operation is merely bypassed to the underlying layer. We have performed only simple performance testing, but we have found little to no noticeable performance degradation due to the DMFS layer.

One operation which will be potentially slower on the DMFS layer is in-kernel vnode allocation, such as is the result of a lookup operation. The allocation of the in-kernel DMFS node requires reading the metadata for this node from the metadata file.

We have performed extensive usage testing of our DMFS layer. We have deployed two internal beta-test systems using two different metadata storage formats (the one described here and a predecessor which stored metadata in a large-inode FFS). In this testing, which included generating over two million files on a 155 GB file system, none of the users complained that the DMFS layer felt slow. Obviously an attempt to access a non-resident file caused a delay, but that was due to the mechanical delay of the robotics retrieving a tape.

We have performed a limited amount of quantitative testing to compare the performance of reading and writing a fully-resident file both with and without the DMFS layer mounted. I performed three tests, all using `dd` to transfer a file either from /dev/zero or to /dev/null. The difference between the tests was the block size used for the operations.

All tests were performed on an IDE disk in an x86-based computer running a modified version of NetBSD 1.4. The file system was built using the default parameters of 8k blocks and 1k fragments. One test was to write a 640 MB file using a write size of 64k. I observed a strong disparity of new-file creation performance over time. Initially file creation took 72 seconds, while later creations took 82 seconds. As a comparable decrease was observed for creation with and without the DMFS layer, I attribute this degradation to disk and file system performance. The exact origin is not important, but this observation motivated all further testing to consist of overwriting an existing file.

I timed three creations without the DMFS layer, and two with. The average times are shown on the first line of Table 1. I do not believe that the presence of the DMFS layer actually improved the FFS performance, but that the variability of times reflects the simplicity of the tests. However the tests indicate that with 64k writes, the extra overhead of the DMFS layer was not noticeable and that I/O scheduling and device performance will add an amount of variability to the tests.

As shown on line two of Table 1, creating a 1000 MB file with 1024k byte writes took the same amount of time both with and without the DMFS layer. This example is similar to the previous one in that the extra overhead of the DMFS layer was not noticeable.

Both of the above tests used large write sizes to maximize performance. As such they minimized the number of times the DMFS layer was called and thus the impact of its additional computations. To better measure the call overhead, I also tried writing with a smaller block size. Line three of Table 1 shows the average times I observed when using 8k writes. Here too, no statistically significant difference was observed.

As I am using layered file system technology, I expect a certain amount of overhead when accessing fully resident files. The rule of thumb estimate I am familiar with is that this overhead should be on the order of one to two percent. My simple tests measured less, and I believe that the rule of thumb one to two percent is a good upper bound.

8 Conclusion

In this paper I have described DMFS, a layered file system developed for NetBSD to support a tertiary storage system. I have briefly described NAStore 3, the storage system of which it is a part, and described DMFS in more detail. I have described the changes to NetBSD needed to support this work. I have given an overview of how a system process interacts with the DMFS layer to archive and restore files, and I have touched on how some typical operations are affected by the DMFS layer. Finally, I presented some simple performance measurements which indicate that while DMFS might impose a performance degradation, it is not significant and that the rule of thumb value of 1 to 2 percent is probably a reasonable upper bound for the performance penalty.

Acknowledgements

This project would not have succeeded without the assistance of others. I'd first like to thank Jason Thorpe for his advice and assistance with many of the kernel design issues raised in this work. My fellow NAStore developers, Tom Proett and Bill Ross, helped me devise how the kernel and NAStore programs interoperate. Harry Waddell and John Lekashman, our management team, encouraged and supported us during the entire development cycle. Finally I would like to thank Chris Demetriou for his design suggestions and especially for his review comments regarding this paper.

References

[1] The NetBSD Project, http://www.netbsd.org/

[2] J. S. Heidemann, *Stackable Design of File Systems*, Ph.D. Dissertation, University of California, Los Angeles (1995).

[3] J. Kohl, C. Staelin, M. Stonebraker, "HighLight: Using a Log-structured File System for Tertiary Storage Management", Proceedings of the San Diego Usenix Conference, January 1993.

[4] M. McKusick, K. Bostic, M. Karels, J. Quarterman, *The Design and Implementation of the 4.4 BSD Operating system*, Addison-Wesley Publishing Company (1996).

[5] M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", ACM Transactions on Computer Systems 2, 3. pp 181-197, August 1984.

[6] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, "An Implementation of a Log-Structured File System for UNIX", Proceedings of the San Diego Usenix Conference, pp 201-218, January 1993.

[7] M. McKusick, G. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem", Proceedings of the Freenix Track at the 1999 Usenix Annual Technical Conference, pp 1-17, January 1999.