FIFS: A Framework for Implementing User-Mode File Systems in Windows NT

Danilo Almeida
dalmeida@mit.edu

Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

This paper presents FIFS, a framework that facilitates file system research under Windows NT. FIFS addresses the high cost of file system development under Windows NT by providing a simple user-mode development environment. The environment is a Common Internet File System (CIFS) loopback server that seamlessly integrates with NT's Installable File System (IFS) architecture via the CIFS client included in the operating system. As such, it can provide full NT remote file system semantics. Initial performance measurements of the prototype FIFS implementation show FIFS capable of achieving good performance. Our prototype non-caching user-mode NFS implementation performs at about 70% the speed of a commercial non-caching kernel-mode NFS implementation.

1 Introduction

This paper presents FIFS, a user-mode framework for implementing file systems in Windows NT. For many years, file system research has been conducted on UNIX, which is popular in academic research environments. However, there has recently been increasing interest in Microsoft's Windows NT as a research operating system. Unfortunately, Windows NT does not a have a well-documented, inexpensive file system development environment. Its I/O architecture is radically different from the traditional UNIX architectures and is thus not as widely understood. FIFS is an effort to make it easy to write file systems for Windows NT.

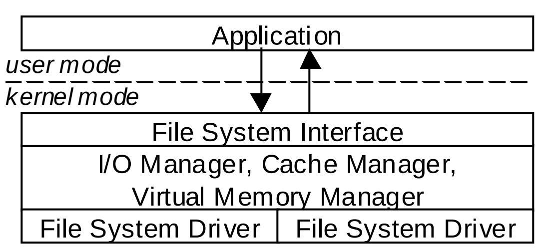

Figure 1-1: Overview of NT IFS Architecture

1.1 Background

The file systems shipped with Windows NT are implemented as kernel-mode file system drivers that plug into the NT's Installable File System (IFS) architecture. They interact with the NT I/O manager, cache manager, and virtual memory manager to satisfy file I/O requests as shown in Figure 1-1.

Unfortunately, writing file systems for Windows NT has been hard for a variety of reasons. As a kernel-mode device driver, a file system driver cannot access traditional operating system services available to user-mode programs. It must also comply with kernel paging restrictions. Due to NT's highly asynchronous I/O architecture, device drivers must not block and must be fully re-entrant. The driver is also hard to debug because it runs in a single address space with all other kernel-mode components and must be debugged with special kernel-mode debuggers rather than the generally more familiar user-mode tools. A file system driver is an especially difficult type of device driver to write. While traditional device drivers interact mostly with just the I/O manager, a file system driver must also engage in complex interactions with the cache manager and the virtual memory manager when satisfying file I/O requests.

While there is some support for writing general device drivers for NT, there is almost no support or documentation to aid in file system driver development. Microsoft has limited support for file system development via the NT Installable File System (IFS) Development Kit. At the beginning of 1997, Microsoft started selling the NT IFS Development Kit for US$1,000 [16]. The kit includes no documentation of the IFS architecture, but instead provides a single half-megabyte header file and source code for the FAT and CDFS[1] file systems. The sample code is fairly optimized and complex, and, without documentation, requires that developers reverse engineer the IFS architecture. In addition, there are no assurances that the file system-related portions of the driver environment will remain stable between releases of Windows NT [16, 17, 18]. In late 1997, the first book about NT file system development was published [17]. While the book provides a great deal of help in kernel-mode file system driver development, it does not make the development process easy or rapid. In fact, the author stresses that developing file systems under NT is a time-consuming process (more so than traditional operating systems, i.e., UNIX).

[1] CDFS provides ISO-9660 (CD-ROM file system) support in Windows NT.

1.2 Goals

Several important criteria affect file system development. Any file system development framework must address the following issues:

price - The framework should not impose a high additional cost barrier to file system development.

performance - File systems developed under the framework must perform at speeds close to traditional implementations.

portability - The framework and file systems should be easily portable across operating system revisions and perhaps even across different operating systems.

richness of semantics - File systems developed using the framework should support full operating system file system semantics. For example, if the operating system and file system both support byte range locking, a framework should allow users to take advantage of the byte range locking capabilities through the operating system's standard file system interfaces.

ease of programming - The file system development interface provided by the framework must be easy for programmers to use.

ease of use - A file system should be easy to use. Programs should not need to be recompiled or re-linked to take advantage of a new file system.

Our goal is to provide a file system development framework that addresses these issues in a way that makes it easy for developers to implement file systems. Thus, we prefer ease of programming rather than the absolute highest performance.

1.3 Solution

FIFS is a free experimental file system development framework that addresses all of these issues. It is the first file system development framework for Windows NT that is fully implemented in user-mode. By running in user-mode, FIFS addresses the issues of portability and ease of programming. Rather than forcing the developer to write to the more volatile kernel-mode programming environment, FIFS allows the developer to write a simple user-mode dynamic link library (DLL), using traditional operating system facilities, programming libraries, and development tools. As a user-mode driver, a FIFS driver is much easier to debug. It is thus very well suited for experimental file systems. A file system developed with FIFS is easy to use as it plugs transparently into the Windows NT namespace. It has the potential to perform all file system operations available to NT user-mode programs, including file locking, byte range locking, and directory notification. Our initial performance tests show that our user-mode NFS FIFS file system driver performs comparably to a commercial kernel-mode NFS implementation.

In addition to the file system development framework itself, FIFS includes the FSWIN32, FSMUNGE, and FSNFS user-mode file system drivers. FSWIN32 is a FIFS file system driver that makes Win32 file system API calls to provide access to whatever is accessible via the local machine. It is mainly intended as a testing and performance measurement tool. FSMUNGE is a pathname dissection filter driver that allows simpler file system driver implementations to plug directly into FIFS. FSNFS is a simple NFS file system driver that works with FSMUNGE.

1.4 Organization

In this paper, we examine related work, discuss the design of FIFS, describe some aspects of the prototype implementations of FIFS and the FSWIN32, FSMUNGE, and FSNFS file system drivers. We also investigate the performance of the initial FIFS implementation by comparing FSWIN32 to local file system access and comparing FSNFS to a commercial kernel-mode NFS implementation. Finally, we suggest future work that needs to be done with FIFS.

2 Related Work

The vnode interface originally developed by Sun is one of the most common mechanisms for adding file systems to UNIX [5, 9]. It is simple compared to the interfaces used by NT's IFS architecture and has enjoyed a fair amount of use in research file system implementations [4, 23]. Until recently, there has been no similarly simple interface to NT file system development. It is our hope that FIFS will help change this situation.

There are several commercial file system development kits for Windows NT. One such kit is Open Systems Resource's (OSR) File Systems Development Kit (FSDK), which is based on source code licensed from Microsoft and is considered to be an excellent NT kernel-mode file system development kit [18]. It provides wrappers that attempt to isolate the file system driver developer from the complexity of the NT kernel. Unfortunately, the kit is sold for US$95,000. According to OSR, the price of the FSDK reflects the current cost of NT file system driver development [20].

There are several kernel-mode file system drivers available to users in addition to those that are shipped with NT. Most are commercial NFS clients. Some of these include Intergraph's DiskAccess, Xlink's Omni-NFS Enterprise, FTP Software's InterDrive, and Hummingbird's NFS Maestro. Since FIFS was conceived, there has been one free kernel-mode implementation of ext2fs released.

User-mode file systems have also been implemented under Windows NT. In mid-1997, Galen Hunt developed a kernel-mode proxy driver at Microsoft Research. This proxy forwards I/O requests from the kernel into user-mode and allows for the development of user-mode file systems

[7, 8]. Hunt provides a library that is specifically geared towards writing file systems with his proxy. Hunt successfully implemented HTTP and FTP file system drivers using his proxy. One novel feature of Hunt's proxy driver is that the user-mode drivers are developed as Component Object Model (COM) objects. FIFS uses a similar object-oriented approach. At the time FIFS was developed, Hunt's proxy driver was not available to the general public. Today, however, it can be downloaded from Microsoft Research. We have not had time to evaluate the framework since it was released to the public.

An alternative approach is to use an existing network file system driver to forward file system requests to a user-mode file server via a standard network connection. This server services the request and sends the response back to the network file system driver via the connection. When the user-mode file server runs on the same machine as the client network file system driver, the server is called a *loopback server*. This was the approach used in creating a portable UNIX SFS (secure file system) client implementation in [12]. In that case, an NFS loopback server was used because most UNIX kernels contain an NFS client. While Windows NT does not include an NFS client, it does include a Common Internet File System (CIFS) client, which uses the Server Message Block (SMB) protocol[2]. A CIFS loopback server is particularly suited to implement file systems on NT because some of the SMB calls are exactly the same as NT system calls. An example of a CIFS loopback server file system implementation is Transarc's commercial AFS client for Windows NT.

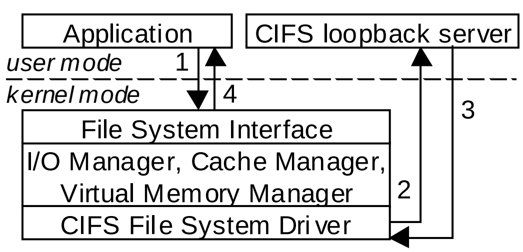

Figure 2-1: Operation of CIFS loopback server file system

Figure 2-1 shows how a CIFS loopback server file system implementation works. First, the application issues file system request to the kernel (1). Then, the kernel-mode CIFS client issues one or more corresponding SMB requests to the user mode CIFS loopback server (2). The user-mode CIFS loopback server fulfills the requests (possibly by going out over the network and/or reading the local disk) and returns results to the CIFS client as an SMB response (3). Finally, the application receives the results from the kernel.

A user-mode file system can also be implemented as a file system library. However, each program that wishes to make use of the new file system needs to be statically or dynamically linked to the library. Programs that use the file system interface exported directly by the kernel will simply be unable to use the new file system unless they are recompiled. No work on such a file system implementation has been done on NT. However, some work has been done in providing UNIX libraries on NT which provide users with UNIX-like semantics for file systems under NT [10, 25].

3 Design

We chose a user-mode CIFS loopback server[3] design for FIFS. This gives the framework full integration into the NT namespace as well as the potential for full NT remote file system semantics[4]. As a user-mode server using a standard protocol, the loopback server is portable across operating system revisions. To facilitate file system programming, the loopback server calls into a straightforward file system dispatch table.

3.1 Server

The loopback server is a user-mode process that listens on a NetBIOS name [14] and responds to CIFS requests from the local machine's CIFS client. (To prevent connection hijack attacks, requests are accepted only from the local client.) The requests can be divided into 2 categories: connection management and file system dispatch operations. Connection management requests consist of setting up the CIFS session to the local machine and performing user authentication. File system dispatch operations are the standard open, close, read, write, etc. operations.

When a user attempts to connect to a particular file system, the CIFS server can use pass-through authentication to the local machine. After establishing the identity of the user, the server passes the user's principal identifier to an underlying user-mode file system driver and receives a dispatch table for the user. The dispatch table is a set of file system functions (like read, write, etc.) with an associated user security context. Subsequent file system operations for that user are invoked through that dispatch table.

The functionality supported via the loopback server is only as rich as the functionality provided by the SMB protocol.

[2] The terms *CIFS* and *SMB* are used interchangeably throughout the rest of the paper. They refer to the same protocol. Servers (or clients) implementing the protocol are known as CIFS and SMB servers (or clients).

[3] For the remainder of this paper, the term *FIFS server*, *loopback server*, and *CIFS server* refer to the FIFS loopback server. Unless otherwise noted, other references to *server* also refer to the FIFS loopback server.

[4] The CIFS semantics are very similar to NT file system semantics. For details on their similarities, see [11] and [16].

Since SMB does not support such things as hard links, creating symbolic links, and setting standard UNIX ownership information and mode bits, the framework cannot support these features directly. However, this additional functionality can be provided over SMB via IOCTLs.

3.2 Namespace

A user names files on a FIFS file system via universal naming convention (UNC) names of the form `\\server\share\path`, where `server` is the NetBIOS name of the FIFS server and `share` can be anything. The NT CIFS client will direct requests to names of this form to the FIFS loopback server, passing the `share` identifier and `path` portion of the name to the server. A user can avoid having to specify a UNC pathname by mapping the `\\sever\share` portion of the pathname to a drive letter.

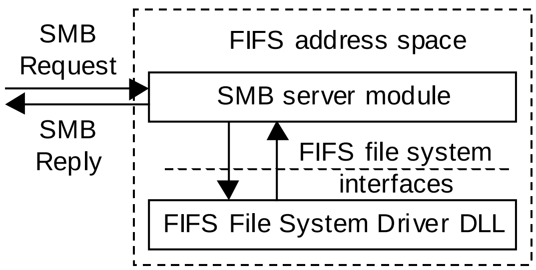

Figure 3-1 : FIFS Architecture

3.3 File System Drivers

The FIFS server uses user-mode FIFS file system drivers to satisfy SMB requests as shown in Figure 3-1. A file system driver is implemented in a Windows DLL. The DLL exports a single function, `FileSystemCreate()`, that allows the server to request an interface pointer to a `FileSystem` interface. `FileSystemCreate()` takes in a file system name, a configuration string, and an interface version. It returns a pointer to the desired version of the `FileSystem` interface for the corresponding file system, configured according to the configuration string. In pseudo-code, the function is defined as follows:

```
FileSystem = FileSystemCreate(fs_name,
        fs_config_path, version)
```

This design is based on the Component Object Model (COM) [3]. The `fs_name` argument allows a DLL to act as a driver for multiple file systems. The `fs_config_path` argument allows the file system to be dynamically configured by the server. If a new version of the file system interface is developed, a different `version` number can be used for the interface. Then, the server can try to use the latest version of the file system interface supported by the file system DLL.

3.4 File System Interfaces

The file system interfaces are thread-safe and provide COM-like reference counting. As thread-safe interfaces, they can be readily used in a multi-threaded loopback server. As in COM, reference counting is achieved via `AddRef()` and `Release()` methods in the interfaces. Programmers using these interfaces must therefore call `AddRef()` whenever they assign an additional reference to the interface and `Release()` whenever they release a reference to the interface object. This is so that the memory allocated for the interface can be automatically de-allocated by the interface object itself once its reference count is zero. Any function that allocates an interface object and returns an interface pointer (such as `FileSystemCreate()` above and `FileSystem::connect()` below) must ensure that the reference count for the interface is equal to one.

The initial version of the file system interfaces is `FS_VERSION_0`. The main interface is the `FileSystem` interface. Aside from reference counting, the only function that this interface provides is `connect()`, which, given a principal identifier for a user, returns a `FsDispatchTable` interface that is associated with the user's security context.

The `FsDispatchTable` interface is a simple, handle-based file system interface derived from the Win32 interface and the vnode interface. Aside from `AddRef()` and `Release()`, it contains the following functions:

Function	Description
`get_principal`	Returns principal associated with this dispatch table
`get_root`	Returns handle to root of file system.
`create`	Creates/opens files, opens directories and returns a handle, depending on flags.
`lookup`	Looks up a name in a directory and returns its attributes.
`set_attr`	Given a handle, sets file/directory attributes.
`get_attr`	Given a handle, returns file/directory attributes.
`close`	Closes a handle.
`write`	Writes data to a file handle at the specified offset.
`read`	Reads data from a file at the specified offset.
`read_dir`	Given a directory handle and cookie, returns directory entries.
`statfs`	Returns file system attributes, including volume name and size information.
`remove`	Removes file with given name from a directory.
`rename`	Renames a file.
`mkdir`	Creates a directory with the specified attributes.

`rmdir`	Removes a directory.
`readlink`	Given a symbolic link handle, returns path to which it points.
`symlink`	Creates a symbolic link.
`link`	Adds a hard link.
`ioctl`	Performs an IOCTL on a file handle.
`flush`	Returns after putting file on stable storage.

Table 3-1: Summary of FsDispatchTable interface

This interface allows all standard directory and file operations to be performed. However, it does lack locking and callback notification facilities. Section 6.3 discusses these missing features in more detail.

3.5 Layering: Filters and Adapters

The framework provided by FIFS is flexible. A file system driver can call into any other file system driver in the same way that the server can. Thus, the framework can achieve layering of file system drivers. Figure 3-2 illustrates this principle.

There are a wide variety of applications for file system driver layering. For example, a simple encrypting layer can be used as a filter to encrypt and decrypt all data written to and read from a given file system. A separate filter driver might audit all file accesses in a particular directory that contains confidential information.

A file system driver can also be used as an adapter from the FIFS server component to a simple file system implementation. For example, a simple FIFS server can be implemented such that it is not aware of symbolic links and never calls `readlink()` and `symlink()`. To make this server work with a file system driver that returns symbolic link attributes and does not automatically traverse symbolic links, an adapter layer can be written that transparently traverses symbolic links. Similarly, if a file system driver is case sensitive, but the names provided by the CIFS client are not, a file system adapter can mask the mismatch between the server and the client.

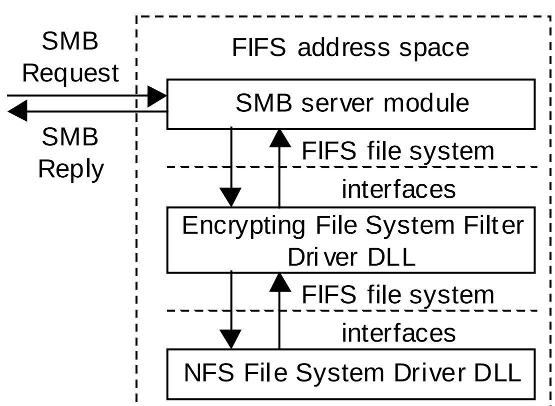

Figure 3-2 : Example of File System Layering in FIFS

This layered architecture allows developers to write simple file system drivers that can be matched to whatever server implementation is being used through any number of layered drivers. Thus, if desired, a filter or adapter can be simple and do a single task but several can be combined to perform complex data transformations and actions.

4 Implementation

The initial FIFS implementation consists of the CIFS loopback server, two file system drivers, and an adapter driver.

4.1 Server Implementation

The SMB protocol used in CIFS supports several different dialects [11]. The first step in implementing the FIFS loopback server is the selection of an SMB dialect. The newer dialects have more advanced functionality (e.g., file locking, better user authentication, etc.) but also support all requests in the older dialects. The CIFS specification suggests that new clients using a new protocol should not use older-style SMB messages so that, in the future, new SMB servers will not have to support the older messages. However, new SMB servers are currently supposed to support old-style messages in their dialect. This makes writing SMB servers more cumbersome than necessary.[5] In this implementation, the `LM1.2X002` dialect [11] is used. It provides the richest semantics without the more obscure NT-specific features of the more recent `NT LM 0.12` dialect [11]. Because all current SMB dialects include older dialects, an `LM1.2X002` implementation can be used as a stepping stone to an `NT LM 0.12` implementation.

A drawback of not using `NT LM 0.12` is that IOCTLs are not supported in previous SMB dialects (or, at least, are undocumented). Therefore, this FIFS implementation does not support IOCTLs.

For the initial FIFS implementation, we chose not to support SMB's opportunistic locking [11] so as to simplify the implementation. We also do not implement pass-through authentication. Sections 6.1 and 6.3 have more details on these topics.

The server is multi-threaded and maintains a minimal amount of global state that is protected from concurrent access. (This state mainly consists of `FsDispatchTable` pointers.) The access operations are fast and allow the server to be highly concurrent. The only time-consuming operations that a thread might do are calls into a dispatch table. In that case, the file system driver is responsible for

[5] The correct solution is to define a new dialect that only supports the newer-style SMB messages. Then, writing servers that speak the newest SMB dialect would be a less cumbersome task. Since the SMB loopback server needs to work with the current NT CIFS client, it cannot define a new SMB dialect and must instead use one of the supported dialects.

safely maintaining its internal state and achieving as much concurrency as desired.

The server does not directly call any of the symbolic or hard link functions in underlying drivers because SMB does not know about symbolic links. Instead, it relies on the file system driver to provide transparent access to symbolic links (directly or via a layered driver).

4.1.1 Configuration

Server configuration parameters are specified via a Windows NT registry pathname argument to the server. The configuration includes NetBIOS name (which by default consists of the local machine name and a few extra characters), the desired NetBIOS buffer size, the number of worker threads desired, the file system driver DLL to use, the file system name to ask for, and the file system configuration information string. The server currently runs as a regular process rather than as a Windows NT service.

4.1.2 Pathnames

An important feature of this server is that it passes pathnames to the underlying file system driver without interpreting them. Since such pathnames are often absolute pathnames, the underlying file system driver must be prepared to handle a backslash-delimited pathname. An advantage of this approach is that the FIFS server does not have to split up the name and traverse the pathname by calling `create()` multiple times. Rather, it can just pass the full name to the underlying file system driver, which can do whatever it wants to do.

One problem that we discovered while implementing the server is that the NT CIFS client sometimes passes uppercase pathnames to the loopback server. This is a problem if the underlying FIFS file system driver is case-sensitive. In some cases, the NT client requests all entries in a directory from the loopback server. However, there are cases where the NT client passes an uppercase string as a filter to an SMB directory enumeration request. Our loopback server optimizes lookups for such a filter by calling the `lookup()` function on the given name instead of calling `read_dir()` and filtering the results. One problem with this approach is that `lookup()` does not return a name. So, the server fills in the name information in the SMB directory enumeration reply with the uppercase string that it received from the CIFS client. This can be a problem if the client then uses the name to open a file on a case-sensitive file system. In order to circumvent this idiosyncrasy, some future work can be done either in the main file system driver itself or as a layered driver (see Section 6.6).

4.2 FSWIN32

FSWIN32 is the first file system driver implemented for FIFS. It allows the user to access a subtree of the local machine's namespace. It simply converts its arguments and calls directly into the Win32 API. The implementation uses coarse locking and thus exhibits little concurrency. Most of the file system functions in this driver lock the user's entire dispatch table object. The purpose of this file system driver was to do initial framework validation as well as to get an idea of the overhead of FIFS when accessing parts of the Win32 namespace. Its only interesting performance feature is that it pre-fetches and caches directory information.

4.3 FSMUNGE

FSMUNGE is a file system adapter. Its configuration information specifies the underlying file system driver to which it will serve as an adapter. Whenever FSMUNGE receives a request with a multi-part pathname, it parses the pathname and opens each directory component using the underlying file system driver. It then fulfills the request by calling the underlying file system driver with the resulting directory handle and final pathname component. This filter driver is fully asynchronous. It will block only if the underlying file system driver blocks.

The purpose of this file system adapter was to allow us to develop FSNFS without having to handle multi-part pathnames. FSMUNGE was easy to develop. In fact, it only took about an hour of development time, most of which was spent writing a pathname dissection class. The ease of development is a good indication of how easy it is to write FIFS drivers. FSMUNGE was later revised to make pathnames lower case so that the underlying FSNFS driver did not have to handle uppercase names sent by the NT CIFS client. (A side effect of this is that the FSNFS driver is unable to access files with names containing upper case letters. A fix to this problem is suggested in Section 6.6).

4.4 FSNFS

FSNFS is an NFS version 2 file system driver. Like FSMUNGE, this driver took relatively little time to develop. Its was implemented in a day by someone who had never implemented an NFS client or server and who had never written a significant piece of code using the ONC/Sun RPC library. Due to lack of time, we did not add any caching optimizations to this driver. So FSNFS must always make one or more NFS remote procedure calls to satisfy requests.

Because the freely available ONC/Sun RPC library implementation for NT is not thread-safe, we use a coarse locking discipline for FSNFS. One big drawback to the low FSNFS concurrency is that large read and write requests get broken down into 8KB chunks that are issued synchronously rather than asynchronously.

Though NFS supports symbolic links, this first FSNFS implementation does not. FSNFS does not make any provisions to handle case-insensitive names passed into it. The driver could support this functionality by reading the directory where the name lookup is taking place and doing a case-insensitive match. Since FSNFS does no caching, such functionality can just as easily be implemented as an adapter driver. Section 6.6 discusses the issues that must be addressed to fix this problem.

5 Experiments and Results

We compared the performance of accessing the local NTFS file system directly versus access through FSWIN32 to obtain a rough measurement of the overhead of FIFS. We also compared FSNFS to NFS Maestro Solo, a commercial NFS client implemented as a kernel-mode file system driver created by Hummingbird Communications Ltd. We chose NFS Maestro because it has been rated as one of the best-performing NFS clients for NT. By default, Maestro uses the NT cache manager. However, it does have an option to disable caching. We believe that it may be possible to integrate FIFS with the NT cache manager (see Section 6.3). However, this initial implementation of FIFS does not include cache manager integration. To obtain a clearer picture of the FIFS overhead, we ran Maestro both with and without caching. We ran the LFS large and small file micro-benchmarks to help understand the performance characteristics of these systems. We then ran an application benchmark so as to understand how these file systems compare under a particular type of workload.

5.1 Setup

The FIFS loopback server and NFS server for these experiments were both 200MHz Pentium Pro machines with 256KB L2 cache, 64MB RAM, 2GB disks, and 10Mb/s SMC Ultra Ethernet cards connected via a 100Mb/s switch. The FIFS machine ran Windows NT Server 4.0 with Service Pack 3 while the NFS server ran OpenBSD 2.2. The tests were run 3 to 5 times to verify that they generated similar results. Neither machine was loaded. While no special care was taken to isolate the machines from broadcast traffic and other connections, the network was monitored to verify that the tests ran under similar conditions. The loopback server was configured to run with a 16KB buffer size. We configured NFS Maestro to use NFS version 2. In addition, as per its performance optimization configuration tool, NFS Maestro was configured to use a 4KB read size and an 8KB write size with no parallelism on reads and 8-way parallelism on writes.

In the results, we indicate the FSWIN32 driver running in a single FIFS server worker thread as FSWIN32-1. Similarly, the FSNFS driver running in a single worker thread is indicated by FSNFS-1. FSWIN32-4 and FSNFS-4 indicate four-threaded runs of the FIFS server with each of these drivers. The kernel-mode NFS Maestro results are listed as kNFS. kNFSnc indicates the results of running kNFS without caching enabled.

5.2 Large File Micro-Benchmark

The large file benchmark sequentially writes a large file, reads it sequentially, writes it randomly, reads it randomly, and then sequentially re-reads it. Data is flushed to disk after each write. For these tests, we used an 8MB file with I/O sizes of 256KB and 8KB. The results are shown in Figure 5-1 and Figure 5-2.

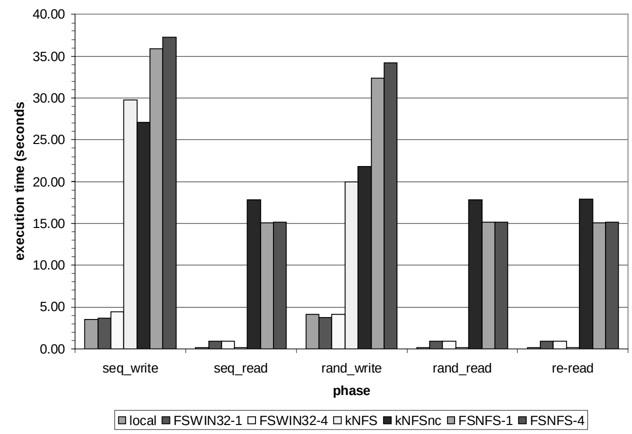

Figure 5-1 : Large File Micro-Benchmark Results (8MB file using 256KB I/O)

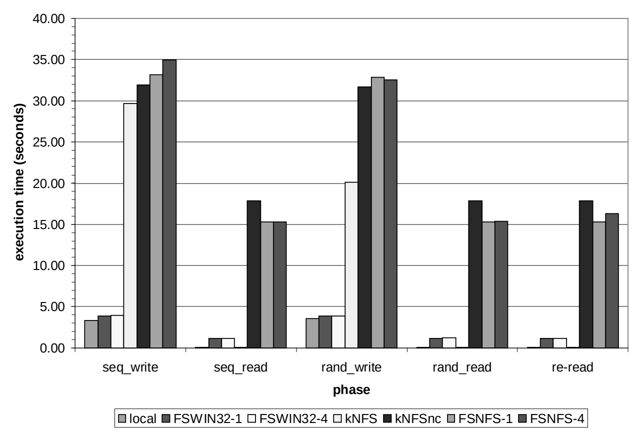

Figure 5-2 : Large File Micro-Benchmark Results (8MB file using 8KB I/O)

The benchmark shows that the FIFS file system driver implementations are slightly slower when running with multiple worker threads. This is expected as the current FSWIN32 and FSNFS implementations block the entire file system dispatch table on each call.

The large file write performance for FSWIN32 versus direct NTFS access shows that FIFS does not add much overhead to writes. We are uncertain as to why FSWIN32 exhibited slightly better performance in the random write phase of the benchmark. Varying the I/O size does not significantly affect the FSWIN32 results.

FSWIN32 read performance is poor compared to direct NTFS access. The difference is 0.87 seconds for FSWIN32 versus 0.15 seconds direct NTFS using 256KB I/O and 1.14 seconds versus 0.07 seconds for 8KB I/O. This is because the NT I/O subsystem cannot satisfy FSWIN32 read requests by looking directly at the NT cache. Instead, NT must use FIFS to satisfy the request. FSWIN32 takes more time for the 8KB I/O than the 256KB I/O because it needs to satisfy more individual I/O requests. It is unclear why NT is able to satisfy the 8MB I/O more quickly in 8KB rather than 256KB I/O requests.

For the most part, FSNFS performs comparably to kNFSnc. In reading, kNFSnc is actually slower than FSNFS. It may be the case that the supposedly optimal 4KB read size and 1-way parallelism of NFS Maestro really is not optimal. For reads, the 256KB and 8KB I/O performance does not differ significantly. For writes, kNFSnc significantly outperforms FSNFS for 256KB I/O. While FSNFS handles its I/O synchronously, kNFSnc uses 8-way parallelism during writes and can thus issue 64KB of a 256KB write at once. For 8KB writes, kNFSnc outperforms FSNFS by less than 10 percent.

With NT cache manager integration enabled, kNFS is significantly faster than FSNFS. In reads, kNFS is as fast as NTFS since the data is in the NT cache. Unlike kNFSnc, kNFS can issue writes to the NFS server asynchronously and thus achieve slightly better performance than even kNFSnc does with 256KB I/O. However, kNFS does not achieve this same level of performance when doing sequential I/O. It is unclear why its performance suffers there.

5.3 Small File Micro-Benchmark

The small file micro-benchmark creates 1000 1KB files across 10 directories. It then reads the files, re-writes them, re-writes them flushing the changes for each file, and deletes them. Because this benchmark operates on 1000 files, the read and write phases of the benchmark must open the files before performing I/O. Thus, the times for these benchmarks reflect the time to lookup each file. Figure 5-3 shows the benchmark results.

FSWIN32 performance for create, read, and both types of write takes an additional constant amount of time compared to direct NTFS access. This is because the SMB reply to the opening of a file includes some file attribute information. So, FIFS needs to get attributes for each of the 1000 files that are opened. For delete, however, there is no such overhead, so the execution times are nearly the same.

For the NFS clients, the small I/O size prevents I/O overlap while the lookups and deletes synchronize access to the NFS server. FSNFS performance is not as good as kNFS and kNFSnc performance because FSNFS does a lookup on every pathname component when a file is opened. kNFS and kNFSnc cache the directory lookups and thus save the lookup times. Even with some network tracing, we are unable to explain why the read difference is large while the write and delete differences are small. A more detailed study is needed.

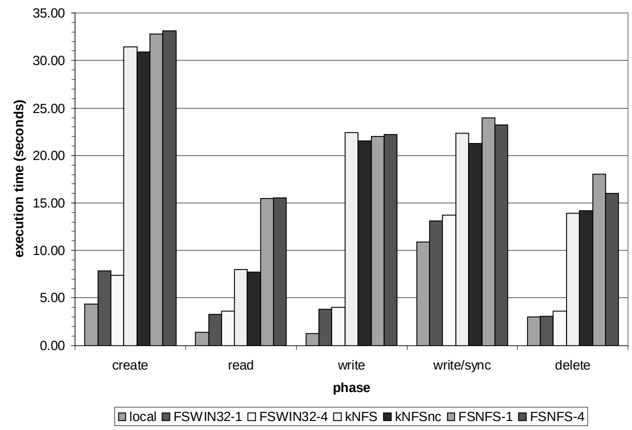

Figure 5-3 : Small File MicroBenchmark Results (1000 1KB files in 10 directories)

5.4 Application Benchmark

Our application benchmark represents a program build scenario. The source tree built in the benchmark is an older version of the FIFS source tree. It contains 209 files with an average file size of approximately 4.5KB.

The benchmark first copies a zip file containing the source tree. It then unzips the source tree and copies it into a new directory tree. It recursively checks the size of every file in the source tree using `du`. Next, it compares the two trees using a recursive `diff`. It then builds the source tree and recursively checks the size of the built source size using `du`. Next, it compares the built source tree with the original copy using a recursive `diff`. The build tree is then zipped into a new archive. It is also zipped into the original zip archive. Then, the trees and original archive are removed.

The benchmark results are divided into 6 categories: copy, unzip, attributes, compare, compile, zip, and remove. The copy, unzip, compare, compile, zip, and remove categories consist of the corresponding operations above. The attributes category consists of the `du` operations. The results are summarized in Figure 5-4 and Figure 5-5.

Looking at the total time shows that the FSWIN32 driver was overall not substantially slower than direct NTFS access. However, a big component of the test is the compilation phase, which has a high CPU utilization. The less CPU-intensive phases show FSWIN32's performance to be 2 to 4 times slower than direct access. The large file micro-benchmark suggests that these performance differences are due to reads rather than writes (see Section 5.2). The NT cache also caches directory information, so FSWIN32 suffers just as much when reading directory information as when reading data compared to direct NTFS access. If it is possible to support the NT cache through support for CIFS opportunistic locks (see Section 6.3), FSWIN32 may become more on par with direct NTFS access.

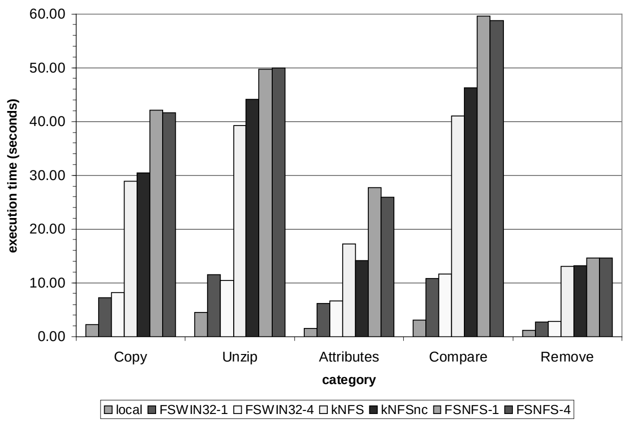

Figure 5-4: Application Benchmark Results – Part 1

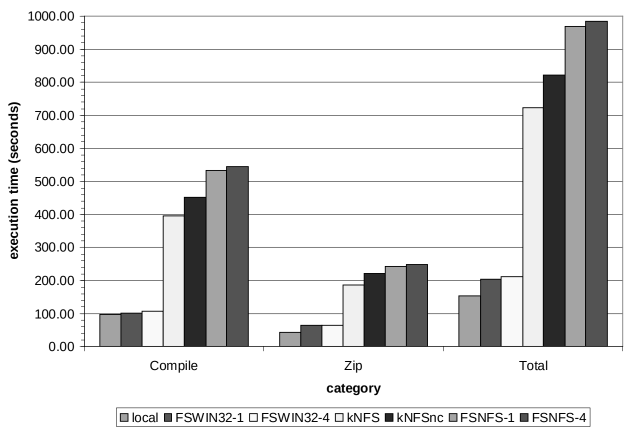

Figure 5-5: Application Benchmark Results – Part 2

The FSNFS driver performs reasonably well compared to NFS Maestro. The kernel-mode client performs at about 1.3 times the speed of FSNFS. The slowest application benchmark category for FSNFS is the attributes category. NFS Maestro is 1.6 times faster than FSNFS in that category. The problem is that FIFS and FSNFS interact poorly when reading directory entries. Section 6.4 contains a more detailed explanation of this problem and some possible solutions.

6 Future Work

While the current FIFS implementation does allow file systems to be implemented under NT, it is still missing some important functionality. The areas that need work are user authentication, IOCTL support, locking, `read_dir()` caching, symbolic link support, and achieving more concurrency in current FIFS file system drivers. We discuss each in turn.

6.1 Authentication

The server currently performs no real authentication and is thus unsuitable for multi-user use under NT. It should be straightforward to implement the pass-through authentication scheme described in Section 3.

6.2 IOCTLs

In order to support IOCTLs for non-CIFS file system semantics, the FIFS server needs to be updated to support the `NT LM 0.12` SMB dialect. This will require some additional work to implement the additional `NT LM 0.12` SMB messages. However, given the current `LM1.2X002` implementation, the change will be incremental in nature.

6.3 Locking, Change Notification, and Caching

FIFS currently has no support for locking and file and directory change notification. It currently lacks the necessary interfaces to support this functionality. In addition, the server currently lacks the support for SMB opportunistic locks that would be necessary to support such functionality. For locking and notification to be implemented, a callback needs to be passed into the file system so that the file system driver can notify the server of directory and file changes. This would require the addition of some extra lock status state to the server and some additional notification threads to the server and file system drivers. The callback mechanism would also support layering as filter drivers could store a higher level driver’s callback and pass its own call back to the underlying file system driver.

If this functionality is added, the framework will be able to efficiently support caching of callback and lease-based network file systems like AFS and SFS. With opportunistic locks enabled, the CIFS client will be allowed to cache files directly and will not need to go the loopback server on cache hits.

One of the most important areas to investigate in this area is whether the NT CIFS client will take advantage of the server’s opportunistic lock support and use the NT cache manager to cache data. If so, it may be possible to have a FIFS file system driver whose performance more closely matches kernel-mode file systems that use the NT cache manager.

6.4 Reading Directories

Some investigation of the network traffic indicates that the reading of directory entries could be made more efficient. Currently, a file system that does not cache directory entry information for an open directory (such as the current implementation of FSNFS) can suffer from unnecessary directory read overlap. The problem is that the size of each entry in the SMB directory enumeration response message depends on the length of the returned file names. Thus, the loopback server needs to know how many entries to read from the underlying file system driver to fill the SMB response buffer. Currently, the server will make a guess and keep calling the underlying file system driver until it fills up the response buffer. So, the loopback server ends up ignoring extra directory entries from the file system and must re-read them when the CIFS client asks for more directory entries. FSWIN32 does not suffer from this behavior because it caches directory responses. However, FSNFS goes out to the network each time.

The only way to fix this is to cache extra directory entries that get returned. This cache can be added at either the file system driver level (like FSWIN32) or the server level. While a server level cache would benefit all file system drivers, it might be redundant if a file system does its own caching across different directory enumeration operations. Therefore, it may be worthwhile to write a simple filter driver that caches directory entries during directory enumeration. Then the filter driver can be used with file system driver implementations that do not do their own caching of directory enumeration information.

6.5 Symbolic Links

Symbolic links were not explored in any of the current FIFS file system drivers. To validate the framework's ability to easily deal with symbolic links, a simple symbolic link adapter driver should be developed to make symbolic links transparent to the CIFS loopback server. With this adapter, UNIX-style file system driver implementations such as FSNFS would simply have to return symbolic link attributes and implement `readlink()` and `symlink()` to provide transparent symbolic link support.

6.6 Case-Sensitive File Systems

The current FIFS framework does not transparently handle case-sensitive file systems (see Sections 4.1.2 and 4.3). This issue can be addressed via a filter driver that reads the directory where a case-insensitive name is being looked up and does a case-insensitive string compare to figure out the corresponding case-sensitive name. The filter can then pass the request through the underlying driver with the case-sensitive name.

Such an implementation is fairly naïve, however. It could suffer from poor performance. A high performance implementation could maintain a per-directory cache of directory entries for recently accessed directories. The cache could be kept up-to-date via the directory notification callback suggested in Section 6.3.

6.7 Achieving More Concurrency

The performance of initial file system driver implementations for FIFS could be improved through fine-grained locking. While the server framework can achieve high concurrency through the use of multiple threads, the current file system driver implementations cannot. It should be possible to retrofit FSWIN32 with finer locking without too much difficulty. This should allow FSWIN32 to have better large file performance for small I/O sizes. FSNFS would likely benefit a lot more from concurrency than FSWIN32 running against the local file system. FSNFS would be able to issue multiple I/O requests on the wire without waiting for the server to reply. However, for FSNFS to achieve this, it would need to use a thread-safe concurrent RPC library.

7 Conclusion

The FIFS prototype demonstrates the potential for a CIFS loopback server-based file system framework. While initial work shows that FIFS performance is comparable to kernel-mode performance in certain cases, it also shows that read performance in the FIFS prototype suffers from not being able to use the NT cache manager. The short time required to implement the FSMUNGE and FSNFS file system drivers shows that FIFS provides a framework where file systems can be easily developed. It is our hope that further work on FIFS and its file system drivers will yield higher performance and a more functional implementation.

Acknowledgements

I would like to thank everyone who helped me write the thesis that led to this paper, especially my advisor, Frans Kaashoek, and my colleagues and friends, Constantine Saputzakis, George Candea, Massimiliano Poletto, David Mazières, Derek Truesdale, Russel Hunt, Joanne Mikkelson, and Luis Sarmenta.

Bibliography

[1] Art Baker. *The Windows NT Device Driver Book: A Guide for Programmers*. Prentice-Hall, Upper Saddle River, New Jersey, December 1996.

[2] Richard Black. Report on the development, structure and performance of ATM device drivers for the personal computer operating systems Windows NT, and Linux. In *ATM Document Collection 4 (The Green Book)*. University of Cambridge, September 1995.

[3] Kraig Brockschmidt. *Inside OLE, 2nd. ed.* Microsoft Press, Redmond, 1995.

[4] David K. Gifford, Pierre Jouvelot, Mark Sheldon, and James O'Toole. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16-25, ACM, 1991.

[5] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Prentice Hall, 1994.

[6] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*. 12(1):58-89, 1994.

[7] Galen Hunt, Creating user-mode device drivers with a proxy . In *Proceedings of the USENIX Windows NT Workshop.* USENIX, 1997.

[8] Galen Hunt, Proxy Driver Home Page, from http://www.research.microsoft.com/os/galenh/proxy/, September 1997.

[9] S. R. Kleiman. Vnodes: an architecture for multiple file system types in Sun UNIX. In *Proceedings of the USENIX 1986 Summer Conference*. USENIX 1986.

[10] David Korn. UWIN – UNIX for Windows. In *Proceedings of the USENIX Windows NT Workshop*. USENIX, 1997.

[11] Paul Leach, Dilip Naik. A Common Internet File System (CIFS/1.0) Protocol. Internet-Draft, IETF, March 1997.

[12] David Mazières. Security and decentralized control in the SFS global file system. Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science Master's Thesis, August 1997

[13] Microsoft Corporation. Microsoft Windows-based Terminal Server, from http://www.microsoft.com/ntserver/guide/hydra.asp, December 1997.

[14] Microsoft Corporation. Platform Software Development Kit. *Microsoft Developer Network Library*, October 1997.

[15] Microsoft Corporation. Windows NT 4.0 Device Driver Kit. *Microsoft Developer Network Library*, July 1997.

[16] Microsoft Corporation. Windows NT IFS Development Kit. from http://www.microsoft.com/hwdev/ntifskit/, June 1997.

[17] Rajeev Nagar. *Windows NT File System Internals*. O'Reilly & Associates, September 1997

[18] Open Systems Resources, Inc. Open Systems Resources File Systems Development Kit, from http://www.osr.com/develkit/fsdk.htm, July 1997.

[19] Matt Pietrek. A Programmer's Perspective on new system DLL features in Windows NT 5.0, Part I. *Microsoft Systems Journal*, November 1997. (also available at http://www.microsoft.com/msj/1197/nt5dll.htm)

[20] Daniel Root, Open Systems Resources, Inc., from e-mail exchange, July 1997.

[21] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, Network Working Group, August 1995.

[22] R. Srinivasan. XDR: External Data Representation Standard. RFC 1832, Network Working Group, August 1995.

[23] David C. Steere, James J. Kistler and M. Satyanarayanan. Efficient User-Level File Cache Management on the Sun Vnode Interface. In *Proceedings of the USENIX 1990 Summer Conference*. USENIX 1990.

[24] Sun Microsystems, Inc. NFS: Network File System protocol Specification. RFC 1094, Network Working Group, March 1989.

[25] Stephen Walli. OpenNT: UNIX application portability to Windows NT via an alternative environment subsystem. In *Proceedings of the USENIX Windows NT Workshop*. USENIX, 1997.