7

Framework for Implementing File Systems in Windows NT

by

Danilo Almeida

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998

Author ....
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by ....[illegible]..
M. Frans Kaashoek
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Framework for Implementing File Systems in Windows NT

by

Danilo Almeida

Submitted to the Department of Electrical Engineering and Computer Science on
May 22, 1998
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents FIFS (Framework for Implementing File Systems), a framework that facilitates academic 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 Common Internet File System (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.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to thank everyone who helped make this happen, especially Frans, Costa, George, Max, Dave, Derek, Rusty, and Joanne.

Contents

List of Figures

List of Tables

1 Introduction

This thesis presents FIFS, a 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. Access to Windows NT source is more restricted than many UNIX implementations. As a result, no significant academic research file systems are implemented on NT today. The motivation for FIFS is to give file system researchers a mechanism that enables them to use a new and increasingly popular operating system to do their work. This will enable file system researchers to reach a wider user base and to use popular application workloads with their file system implementations. In addition, they will be able to explore the architectural implications of NT on their work. With FIFS, almost anyone should be able to write file systems for Windows NT.

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. NT has a hierarchical kernel namespace where all machine resources, including devices and drivers, are named. When a user issues an I/O request to a given name, NT translates the beginning of the name into a file system driver name and dispatches the request along with the untranslated portion of the name to the driver. The file system driver then interacts with the NT I/O manager, cache manager, and virtual memory manager to satisfy the request. Figure 1-1 illustrates the basic structure of the IFS driver architecture.

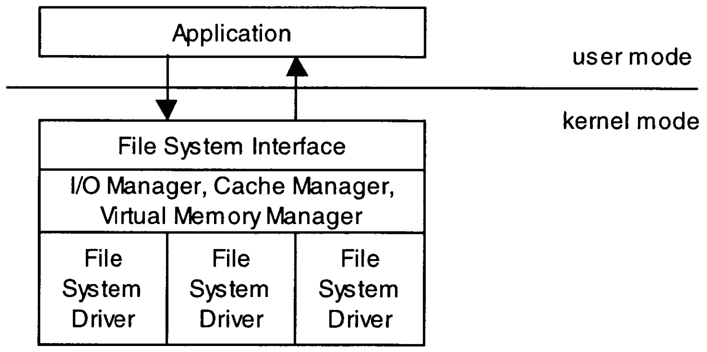


Figure 1-1: Overview of NT IFS Architecture

Unfortunately, writing file system drivers for Windows NT is difficult. In general, device drivers for NT tend to be more difficult to write than UNIX drivers [2] due to the extra complexity of NT's asynchronous I/O request packet-based architecture, the lack of quality kernel-mode development documentation, and the additional difficulty in obtaining an academic license for kernel source. The situation for file system driver development is even worse because these drivers are the most complex type of driver in NT but are the least documented. Thus, NT file system driver development is a difficult and time-consuming undertaking [18].

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 [17]. 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 the 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 [17, 18, 19].

The NT operating system includes several local file system drivers, including those that support NTFS, FAT, and CDFS. The only remote file system[2] driver that

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

[2] For the rest of this document, the term *remote file system* is being used to denote a file system accessed over the network.

comes with the operating system is a Common Internet File System (CIFS) client, which uses the Server Message Block (SMB) protocol[3] [12].

1.2 Issues

Several important criteria affect file system development. Any file system development framework must address these issues. These criteria are price, performance, portability, richness of semantics, ease of programming, and ease of use. Here is a summary of these issues:

Price	The framework must be inexpensive so that researchers on a low budget can still implement file systems.
Performance	A file system developed under the framework must be able to perform at speeds comparable to a kernel-mode file system driver implementation.
Portability	The framework and file systems developed within it 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 an operating system supports byte range locking, a good framework should allow users to take advantage of a file system's 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. If a file system programming interface is difficult to use, file system development time increases, and easier alternatives will be sought.
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.

[3] The terms *CIFS* and *SMB* will be used interchangeably throughout the rest of the document. Both terms refer to a single protocol. Servers (or clients) implementing the protocol are known as both CIFS and SMB servers (or clients).

1.3 Goals

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.

1.4 Solution

Our solution is FIFS, a user-mode file system architecture that plugs into NT's Common Internet File System (CIFS) client. The framework provides a user-mode CIFS loopback server that can be configured to serve a file system implemented via a Windows dynamic link library (DLL). The server is multi-threaded so as to handle requests asynchronously. It calls into a simple but functionally rich file system interface and, given a full implementation, is capable of performing all file system operations available to NT user-mode programs, including file locking, byte range locking, and directory notification.

1.5 Contributions

This thesis makes several contributions to file system development under Windows NT: prototype implementations of FIFS and the FSWIN32, FSMUNGE, and FSNFS user-mode file system drivers. This implementation of FIFS is the first user-mode file system development framework for Windows NT that is fully implemented in user-mode. 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.

1.6 Organization

The remainder of this thesis is organized as follows: Chapter 2 examines related work, including previous efforts in file system development under Windows NT. Chapter 3 discusses the design of FIFS. Chapter 4 describes some of the aspects of the initial implementation of FIFS and the FSWIN32, FSMUNGE, and FSNFS file system drivers.

Chapter 5 investigates 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. Chapter 6 suggests future enhancements to 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, 10]. 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, 9]. There has been no similarly simple interface to NT file system development. It is our hope that FIFS will change this situation.

One of the most well-established file system development groups for Windows NT is Open Systems Resource (OSR). OSR's File Systems Development Kit (FSDK) is based on source code licensed from Microsoft and is considered to be an excellent NT kernel-mode file system development kit [19]. It provides wrappers that attempt to isolate the file system driver developer from the complexity of the NT kernel [see Figure 2-1]. 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 [21].

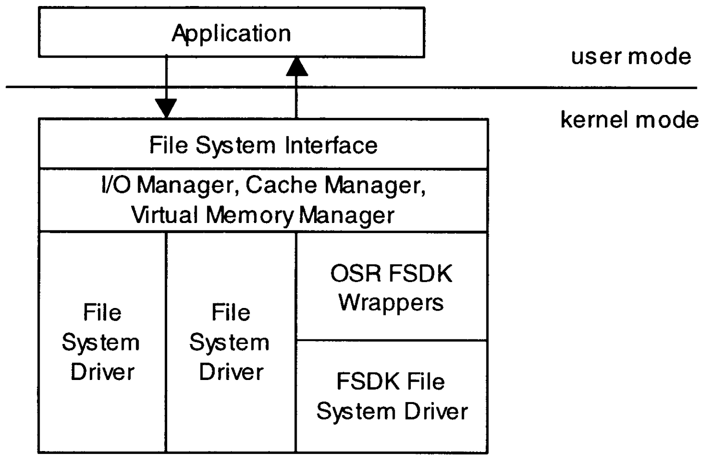


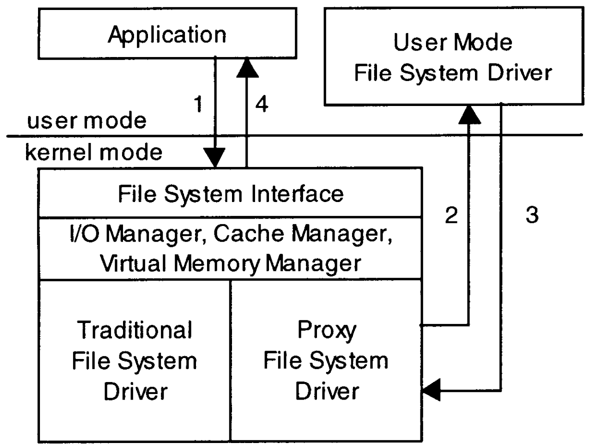
Figure 2-1: Overview of the OSR FSDK

More recently, NT Core Services (NTCS) has announced a beta version of their File System Veneer Toolkit, which is also a kernel-mode file system development kit. We requested documentation and licensing information for this product, but never received any.

In late 1997, the first book about NT file systems was published [18]. It explains the complex, highly asynchronous NT I/O architecture in more detail than does the only other NT device driver book [1]. Due to the complexity of developing file systems under NT, the sample source from the book does not attempt to provide a working file system. 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). The book does not contain enough information to completely implement a file system driver.

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. No freely available file system drivers exist at this time.

User-mode file systems have also been implemented under Windows NT. In mid-1997, Galen Hunt at Microsoft Research developed a kernel-mode proxy driver that calls back into user-mode and allows for the development of user-mode file systems [7, 8]. The principle of his proxy driver is illustrated in Figure 2-2. Since it is a generic device driver proxy, the driver simply proxies I/O request packets into user-mode. Such an implementation is guaranteed to be slower than a full kernel-mode driver implementation due to additional context switching. However, the user-mode environment provides a rich feature set and is more forgiving in that programming errors will not cause the system to crash. The user-mode environment is also better documented. In addition, file systems developed in user-mode are more portable across versions of Windows NT (though the proxy driver itself may not necessarily be so).


Application
User Mode File System Driver
1
4
user mode
kernel mode
File System Interface
I/O Manager, Cache Manager, Virtual Memory Manager
Traditional File System Driver
Proxy File System Driver
2
3

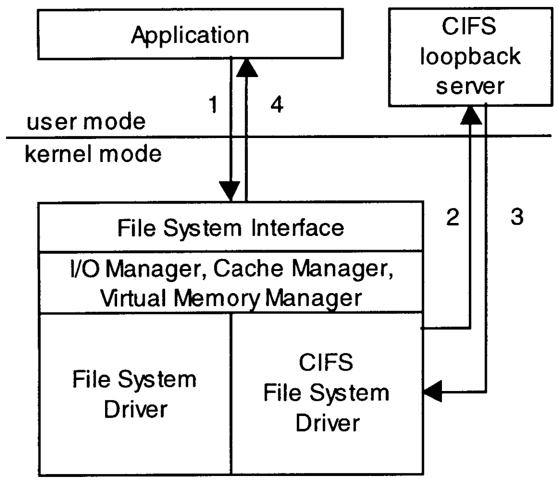
Operation:

1. Application issues file system request to the kernel
2. Proxy file system driver forwards request to user mode file system component
3. User mode file system component fulfills request and returns results to the proxy driver
4. Application receives results from the kernel

Figure 2-2: Operation of proxy file system driver

Hunt successfully implemented an HTTP and FTP file system driver using this scheme. He indicated that his proxy driver is expected to be included in the Windows NT Device Driver Kit. However, the latest Windows NT Device Driver Kit, which was released several months after the publication of his paper, contained no such driver [16]. While his proxy driver has fairly good performance, the programming environment it provides is not specialized for file systems. Rather, it exposes the generic device driver I/O request packet architecture to the programmer. 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.

Fortunately, it is possible to use an existing network file system driver to forward file system requests to a user-mode file server. The user-mode server can run on the same machine and serve as a loopback server. This was the approach used in creating a portable UNIX SFS (secure file system) client implementation in [13]. 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 have a CIFS client. A CIFS loopback server would therefore be more suitable than NFS for NT both because NT's CIFS client comes at no additional cost and because some of the SMB calls are exactly the same as NT system calls. In fact, we reverse-engineered Transarc's commercial AFS client for Windows NT and discovered that their AFS client is implemented as a CIFS loopback server. This is the only instance of such a file system implementation of which we are aware. One problem with the CIFS loopback server approach is that there is not a convenient way to set file system-specific information, such as access control lists, through the SMB protocol. Transarc addresses this issue by providing several utilities that communicate with the loopback server to perform these special functions. The communication takes place via a special invisible file in the root of the file system. Unfortunately, their implementation requires that the file be opened in exclusive mode, preventing more than one communication session from occurring at a given time (thus making users of this AFS implementation vulnerable to denial of service attacks). Figure 2-3 shows how a CIFS loopback server file system implementation works. FIFS uses a CIFS loopback approach to provide its file system framework in a fairly portable manner.



Operation:

1. Application issues file system request to the kernel
2. Kernel mode CIFS client issues corresponding SMB request(s) to the user mode CIFS loopback server
3. User mode CIFS loopback server fulfills request(s) (possibly by going out over the network and/or reading the local disk) and returns results to the CIFS client as an SMB response
4. Application receives results from the kernel

Figure 2-3: Operation of CIFS loopback server file system

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 [11, 25].

3 Design

We chose a user-mode CIFS loopback server[4] 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. 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 [15] 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 or any other trusted administrative domain. 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. Subsequent file system operations for that user are invoked through that dispatch table. Figure 3-1 illustrates the basic operation of the server.

The functionality supported via the loopback server is only as rich as the functionality provided by the SMB protocol. 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 this feature directly. However, this additional functionality can be provided over SMB via IOCTLs.

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

3.2 Namespace

A user names files on a FIFS file system via universal naming convention (UNC) names of the form `\\FIFS_NetBIOS_name\share_name\path`. 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 the share (`\\FIFS_NetBIOS_name\share_name`) portion of a pathname by associating the share with a drive letter.

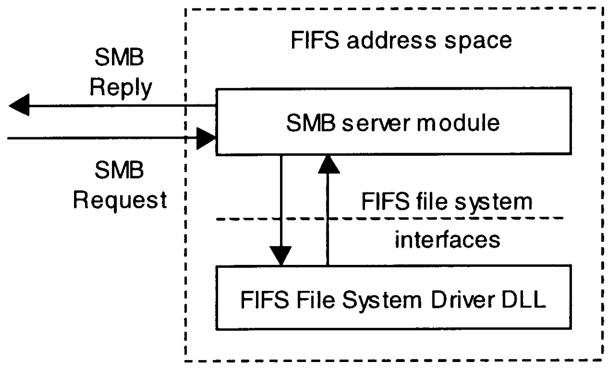


Figure 3-1 : FIFS Architecture

3.3 File System Drivers

The FIFS server uses user-mode FIFS file system drivers to satisfy SMB requests (see Figure 3-1). File system drivers are 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, COM-like interfaces. This is so that a multi-threaded server can be easily used with the interfaces. As COM interfaces, they perform their own reference counting via `AddRef()` and `Release()`. 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. Thus, 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, in pseudo-code, is defined as follows:

```
FsDispatchTable = FileSystem::connect(principal)
```

The server passes a principal identifier string for the user into `connect()` to get 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 (which are fully prototyped in Appendix A):

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 with given attributes and returns a handle. Action depends on flags specified.
`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 this missing feature 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.

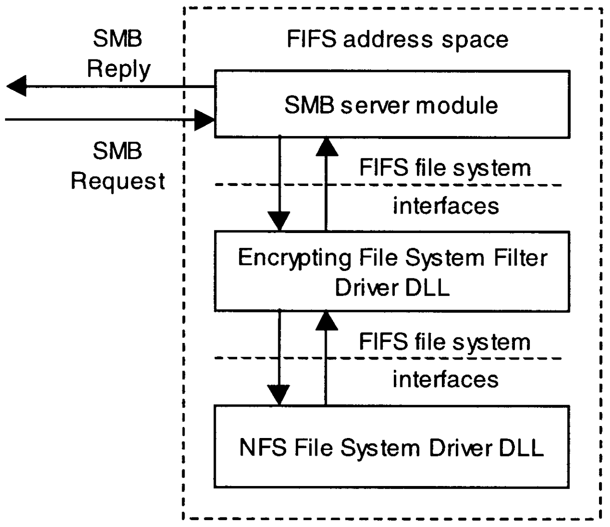

SMB Reply
SMB Request
FIFS address space
SMB server module
FIFS file system interfaces
Encrypting File System Filter Driver DLL
FIFS file system interfaces
NFS File System Driver DLL

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 [12]. 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 [12] is used. It provides the richest semantics without the more obscure NT-specific features of the more recent `NT LM 0.12` dialect [12]. 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 [12] 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`

[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.

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 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 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. Thus, since CIFS often passes full 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 breaks down the pathname and opens each directory component of the pathname 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 driver was to allow us to develop FSNFS without having to handle multi-part pathnames. The file system driver 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. The purpose of this file system driver is to validate that the framework is easy to use and that it can achieve reasonable performance. The development of FSNFS showed that the framework is easy to use. 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 ONC/Sun RPC library implementation freely available 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. 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 inspired by the Andrew benchmark to understand of how these file systems compare under some typical workloads.

5.1 Setup

The FIFS loopback server and NFS server for these experiments are 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 runs Windows NT Server 4.0 with Service Pack 3 while the NFS server run OpenBSD. 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.

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

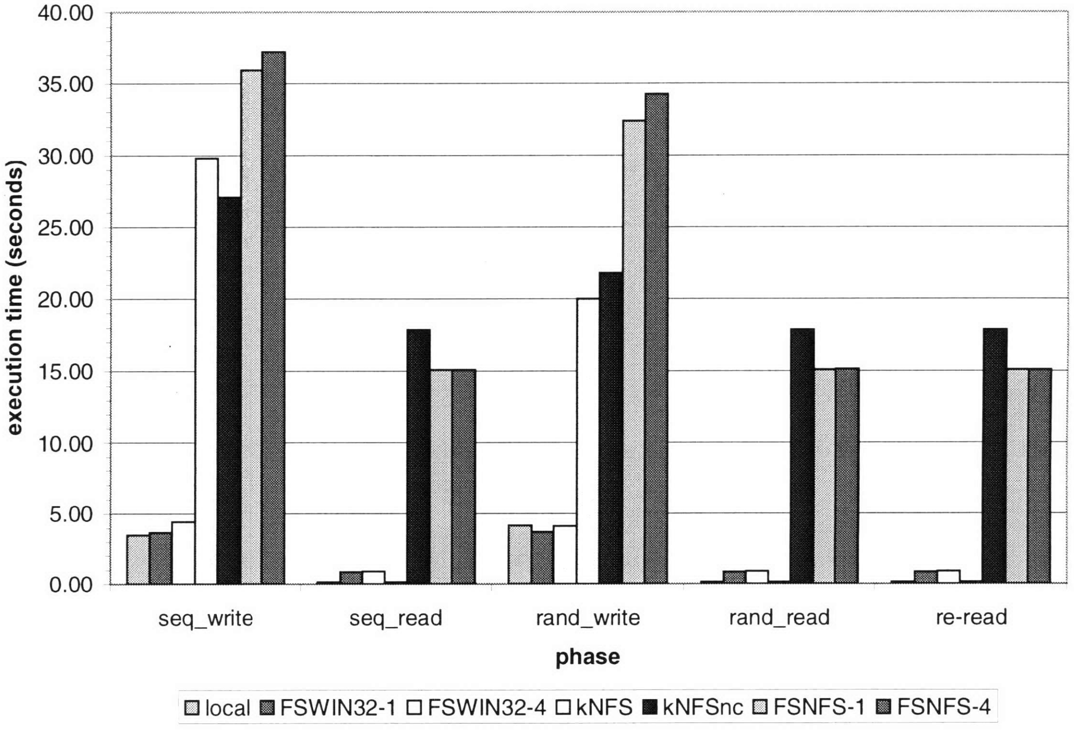


Figure 5-1 : Large File Micro-Benchmark Results (8MB file using 256KB I/O)

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. It may be a result of the random number generator, but we are not certain. 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 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 8KB I/O requests more quickly than 256KB I/O requests.

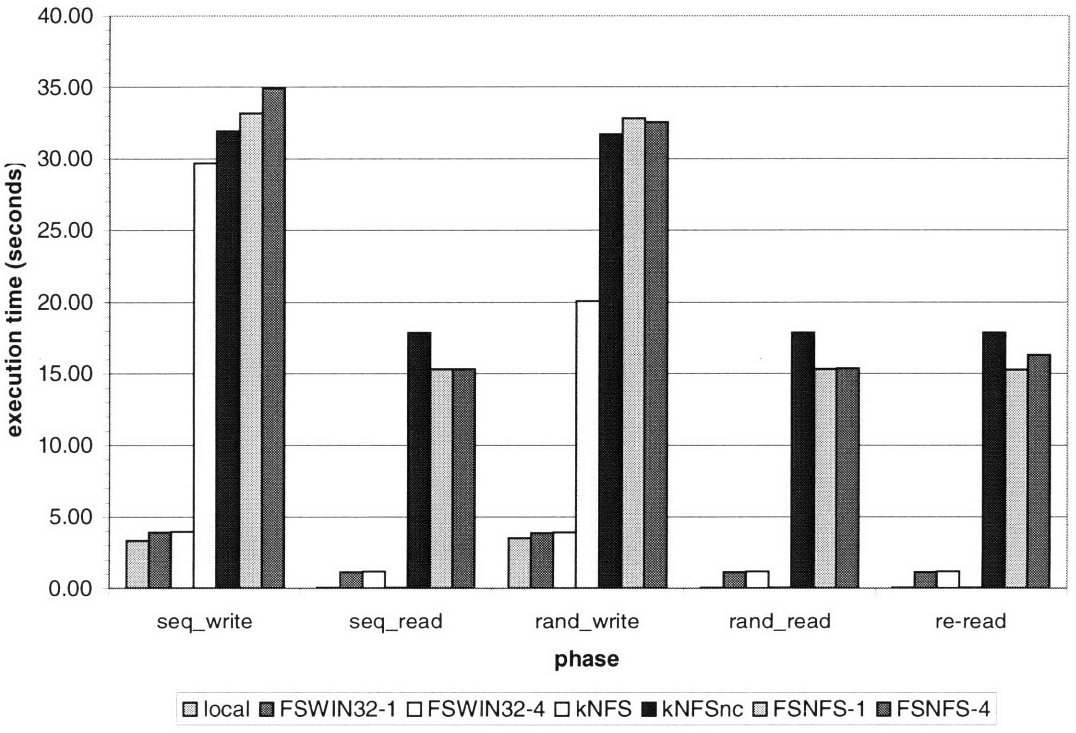


Figure 5-2 : Large File Micro-Benchmark Results (8MB file using 8KB I/O)

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.

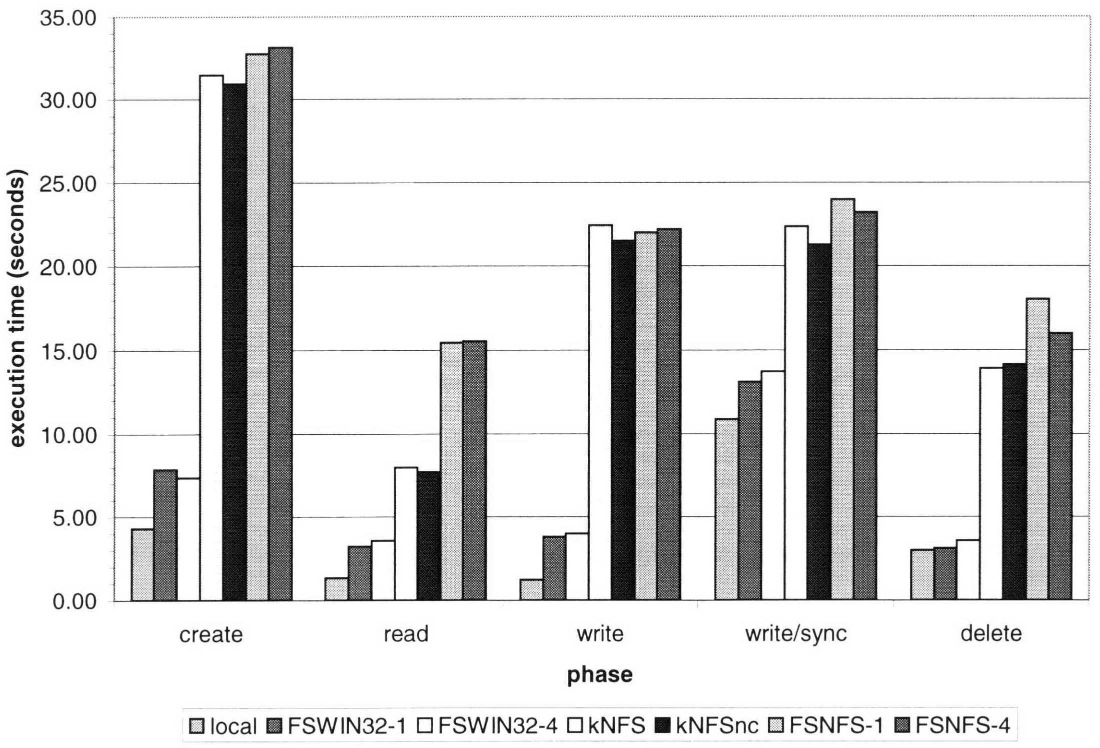


Figure 5-3 : Small File MicroBenchmark Results (1000 1KB files in 10 directories)

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.

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.

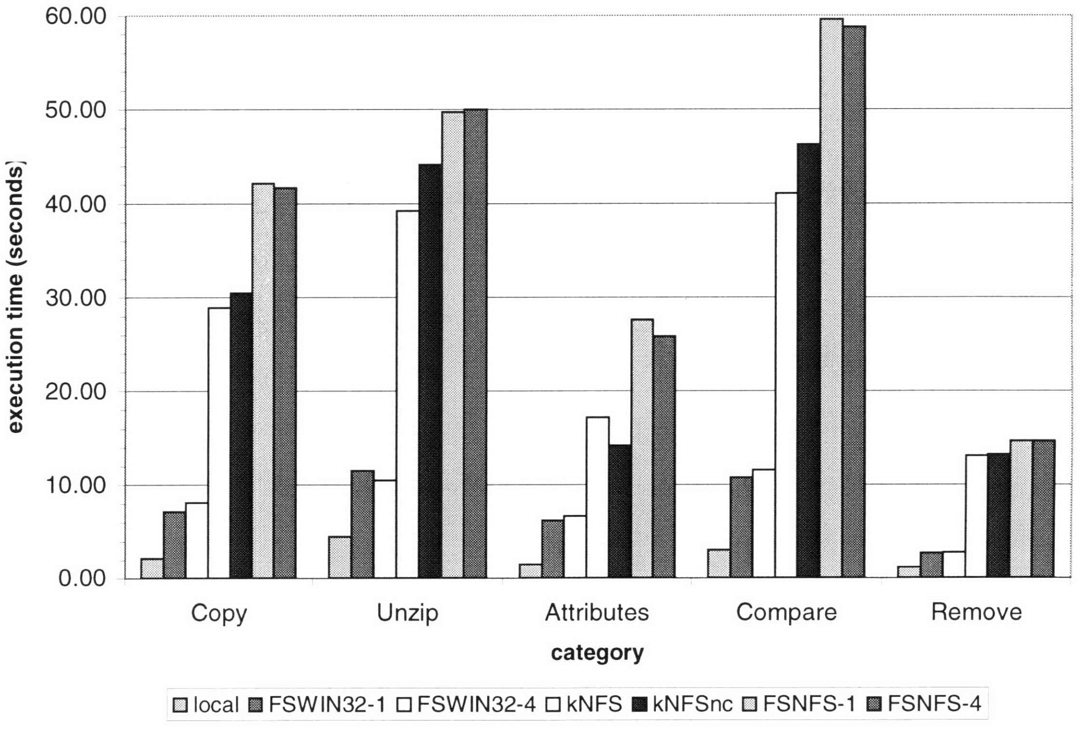


Figure 5-4: Application Benchmark Results – Part 1

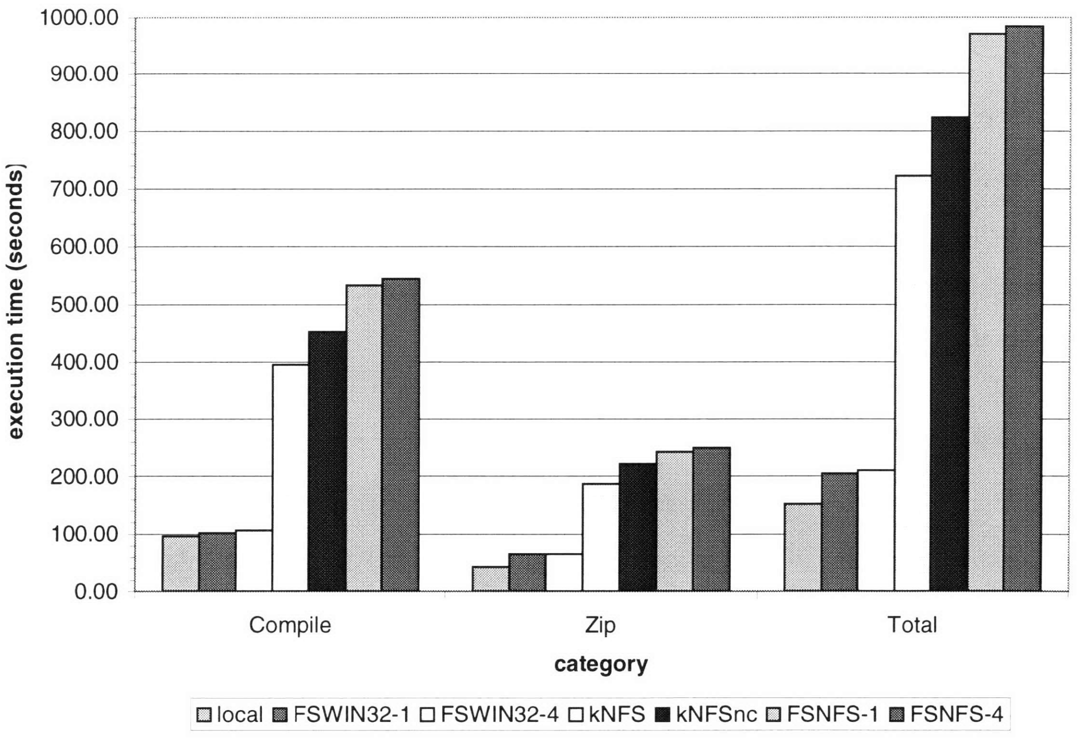


Figure 5-5: Application Benchmark Results – Part 2

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 Chapter 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 system 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 length of the returned file names. Thus, the loopback server needs 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 form 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 cache 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 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.

Appendix A: fsinterface.hxx

This appendix contains the header file for the file system interfaces.

```
#ifndef __FS_INTERFACE_HXX__
#define __FS_INTERFACE_HXX__

// note: we assume DWORD and DWORDDLONG are defined (from windows.h)

#ifndef IN
#define IN
#endif

#ifndef OUT
#define OUT
#endif

// disposition:
namespace FsInterface {

    typedef DWORDLONG UINT64;    // a 64-bit unsigned value
    typedef DWORD     UINT32;    // a 32-bit unsigned value
    typedef UINT64    TIME64;    // in units of 100ns since Jan 1, 1601 (AD)
    typedef UINT64    fhandle_t; // a file handle -- a bit wide

    const UINT64    INVALID_UINT64    = ((UINT64)(-1));
    const UINT32    INVALID_UINT32    = ((UINT32)(-1));
    const TIME64    INVALID_TIME64    = INVALID_UINT64;
    const fhandle_t INVALID_FHANDLE_T = ((fhandle_t)(-1));

    // disposition:
    const UINT32 DISP_CREATE_NEW        = 0x10000000;
    const UINT32 DISP_CREATE_ALWAYS     = 0x20000000;
    const UINT32 DISP_OPEN_EXISTING     = 0x30000000;
    const UINT32 DISP_OPEN_ALWAYS       = 0x40000000;
    const UINT32 DISP_TRUNCATE_EXISTING = 0x50000000;
    const UINT32 DISP_DIRECTORY         = 0x60000000;
    const UINT32 DISP_MASK              = 0x70000000;

    // access:
    const UINT32 ACCESS_READ          = 0x00010000;
    const UINT32 ACCESS_WRITE         = 0x00020000;
    const UINT32 ACCESS_MASK          = 0x00030000;

    // sharing:
    const UINT32 SHARE_READ           = 0x00100000;
    const UINT32 SHARE_WRITE          = 0x00200000;
    const UINT32 SHARE_MASK           = 0x00300000;

    // flags = dispositions | access | sharing
    const UINT32 FLAGS_MASK = DISP_MASK | ACCESS_MASK | SHARE_MASK;

    // attributes:
    const UINT32 ATTR_SYMLINK         = 0x00002000;
    const UINT32 ATTR_DIRECTORY       = 0x00000010;
    const UINT32 ATTR_READONLY        = 0x00000001;
    const UINT32 ATTR_HIDDEN          = 0x00000002;
    const UINT32 ATTR_SYSTEM          = 0x00000004;
    const UINT32 ATTR_ARCHIVE         = 0x00000020;
    const UINT32 ATTR_COMPRESSED      = 0x00000800;
    const UINT32 ATTR_OFFLINE         = 0x00001000;
    const UINT32 ATTR_MASK            = (ATTR_SYMLINK    | ATTR_DIRECTORY |
                                         ATTR_READONLY   | ATTR_HIDDEN    |
                                         ATTR_SYSTEM     | ATTR_ARCHIVE   |
                                         ATTR_COMPRESSED | ATTR_OFFLINE
                                         ); // 0x00003837;

    const size_t MAX_FS_NAME_LEN = 64; // somewhat arbitrary...

    // file system attributes
    struct fs_attr_t {
        char fs_name[MAX_FS_NAME_LEN];
        UINT64 total_bytes;
        UINT64 free_bytes;
    };
```

```
// file/dir attributes
// - we do not use gid/uid because SMB does not deal
// - passing in INVALID_xxx (-1) for any of the values below makes it
//   unspecified.

struct fattr_t {
    // sizes
    UINT64 file_size;
    UINT64 alloc_size;
    // times
    TIME64 create_time;
    TIME64 access_time;
    TIME64 mod_time;
    // mode/attr
    // - a la Win32, but wider so that we can pass in -1 safely
    UINT32 attributes;
};

const size_t MAX_NAME_LENGTH = 256;

typedef struct {
    char name[MAX_NAME_LENGTH];
    struct fattr_t attribs;
    UINT32 cookie;
} dirinfo_t;

class FsDispatchTable {
public:
    virtual ULONG AddRef()  = 0;   // returns new reference count
    virtual ULONG Release() = 0;   // returns new reference count

    // returns string for principal owning this interface
    virtual const char* get_principal() = 0;

    // returns immutable root handle, which can only be used by create()
    virtual fhandle_t   get_root()      = 0;

    virtual DWORD statfs(
        IN     fhandle_t   handle,
        OUT    fs_attr_t*  attr
        ) = 0;
    // create() can take an empty string (a NULL byte) as its name
    // argument to signify opening the given dir again.
    virtual DWORD create(
        IN     fhandle_t   dir,
        IN     const char* name,
        IN     UINT32      flags,
        IN     fattr_t*    attr,
        OUT    fhandle_t*  handle
        ) = 0;
    virtual DWORD lookup(
        IN     fhandle_t   dir,
        IN     const char* name,
        OUT    fattr_t*    attr
        ) = 0;
    virtual DWORD set_attr(
        IN     fhandle_t   handle,
        IN     fattr_t*    attr
        ) = 0;
    virtual DWORD get_attr(
        IN     fhandle_t   handle,
        OUT    fattr_t*    attr
        ) = 0;
    virtual DWORD close(
        IN     fhandle_t   handle
        ) = 0;
    virtual DWORD write(
        IN     fhandle_t   handle,
        IN     UINT64      offset,
        IN OUT UINT64*     count,
        IN     void*       buffer
        ) = 0;
    virtual DWORD read(
        IN     fhandle_t   handle,
        IN     UINT64      offset,
        IN OUT UINT64*     count,
        OUT    void*       buffer
        ) = 0;
        // to use readir(), we do a create() with DISP_DIRECTORY
        // returns ERROR_NO_MORE_FILES when done...
        virtual DWORD read_dir(
            IN      fhandle_t   dir,
            IN      UINT32      cookie,
            OUT     dirinfo_t*  buffer,
            IN      UINT32      size,
            OUT     UINT32*     entries_found
            ) = 0;
        virtual DWORD remove(
            IN      fhandle_t   dir,
            IN      const char* name
            ) = 0;
        virtual DWORD rename(
            IN      fhandle_t   fromdir,
            IN      const char* fromname,
            IN      fhandle_t   todir,
            IN      const char* toname
            ) = 0;
        virtual DWORD mkdir(
            IN      fhandle_t   dir,
            IN      const char* name,
            IN      fattr_t*    attr
            ) = 0;
        virtual DWORD rmdir(
            IN      fhandle_t   dir,
            IN      const char* name
            ) = 0;
        virtual DWORD readlink(
            IN      fhandle_t   handle,
            IN OUT  int*        size,
            OUT     char*       path_buffer
            ) = 0;
        virtual DWORD symlink(
            IN      fhandle_t   dir,
            IN      const char* name,
            IN      const char* path
            ) = 0;
        virtual DWORD link(
            IN      fhandle_t   dir,
            IN      const char* name,
            IN      fhandle_t   handle
            ) = 0;
        virtual DWORD ioctl(
            IN      fhandle_t   handle,
            IN      UINT32      code,
            IN      void*       in_buffer,
            IN OUT  UINT32*     out_size,
            OUT     void*       out_buffer
            ) = 0;
        virtual DWORD flush(
            IN      fhandle_t   handle
            ) = 0;
    };

    class FileSystem {
    public:
        virtual ULONG AddRef()  = 0;   // returns new reference count
        virtual ULONG Release() = 0;   // returns new reference count

        virtual DWORD connect(
            IN      const char* principal,
            OUT     FsDispatchTable** ppDT
            ) = 0;
    };

    typedef DWORD (WINAPI *FS_CREATE_PROC)(
        IN     const char*, // fs_name
        IN     const char*, // fs_config_path
        IN     DWORD,       // interface_version
        OUT    FileSystem** // interface pointer
                            // (i.e., FileSystem** if FS_VERSION_0)
        );

    const char FS_CREATE_PROC_NAME[] = "FileSystemCreate";

    const DWORD FS_VERSION_0 = 0;
}

#endif /* __FS_INTERFACE_HXX__ */
```

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] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3-25, 1992.

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

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

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

[13] 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

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

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

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

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

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

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

[20] Matt Pietrek. A Programmer's Perspective on new system DLL features in Windows NT 5.0, Part I. *Microsoft Systems Journal*, November 1997.

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

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

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

[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.

4[illegible]- 7