FUSE-NT: Userspace File Systems for Windows NT

Evan Driscoll Jonathan Beavers Hidetoshi Tokuda

University of Wisconsin-Madison, Department of Computer Sciences
{driscoll, beavers, hidetosh}@ cs.wisc.edu

Abstract

This paper presents our progress towards FUSE-NT, a Windows port of Filesystem in Userspace (FUSE). FUSE is a service that allows users to implement functional file systems in userspace. It provides a simple but sufficiently powerful set of APIs which allow users to design and implement original file systems without dealing with the complications of kernel programming. Currently, FUSE itself is available on Linux, BSD, and Mac OS, but not Windows. Implementing FUSE on Windows brings about new challenges due to different file system design principles and ethics of the two systems, and a good amount of complexity in the Windows interfaces. This paper describes our unsuccessful attempts at writing a Windows port of FUSE using a native file system driver and some of the difficulties we faced as well as design decisions we had to make. It then describes our prototype that uses a server for the Common Internet File System (CIFS) to provide the FIFS interface.

1. INTRODUCTION

Designing and implementing file systems has traditionally been the realm of people with extensive knowledge of operating systems and kernel mechanisms. Porting a file system to another operating system can require reimplementing it in the context that OS. If a designer is terribly unlucky, they could potentially find themselves redesigning large portions of their original code.

Recently, there has been increased interest in being able to make file system implementation easier. There are a number of possible reasons for doing this. First, CPU speed has been rising dramatically which reduces the overhead of context switches between kernel and userspace. Second, there has been a rise in demand for some "odd" file systems, such as ones that go over a network or the Internet using a protocol that wasn't designed to be a networked file system. (SSHFS [22] is perhaps the canonical example in this category now.)

One common approach to solve this problem is to implement a framework that enables developers to implement a file system in userspace with a simple and powerful set of APIs that are common amongst most operating systems. There have been a number of efforts to forward requests to userspace. Common approach is to implement a mechanism in a file system to forward the system call to userspace and process the calls without existing restrictions caused by the operating system. The number of production-quality systems that provide a standardized API for developers to design a unique file system in userspace is still small, but there is one commonly used and well deployed system called FUSE, part of the Linux kernel since version 2.6.14.

FUSE is the most well known example of a userspace file system implementation [9]. As shown in figure 1, FUSE uses the Virtual File System interface (VFS) in Unix to receive file I/O requests from applications and forward them to the FUSE kernel module. The kernel module provides a device `/dev/fuse` which a userspace daemon[1] uses to interface with the driver. The server reads this device, and the driver uses that read call as a conduit to send a request up to the server. The server can satisfy the request however it pleases, so long as it then returns the result to the original application by writing to `/dev/fuse`.

Easy prototyping and application friendliness are two of this design's key advantages. Developers implementing a file system in userspace no longer have to recompile the kernel or worry about crashing the operating system during development. Additionally, FUSE's relatively loose policy of implementing file system APIs allows developers to run file systems with only a few functions implemented. The overall friendliness of FUSE may be largely attributed to the VFS interface. It presents an application with a well known, standardized, and native file system that accepts regular system calls. This means that applications can use interesting and cutting edge file systems on FUSE without changing any code inside the application. These two characteristics of FUSE's design clearly encourage not only file system developers but also other people who are not familiar with kernel programming to challenge themselves

[1]We will refer to the userspace portion of a FUSE file system as a server, inspired by microkernel terminology.

by implementing their own file systems.

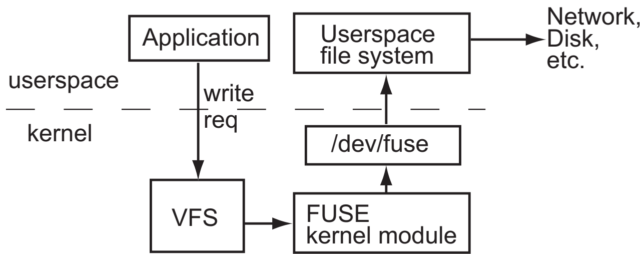

Figure 1: Overview of how fuse works

This architecture also gives FUSE another interesting benefit, which has become one of the project goals. Under this system, as long as the system administrator loads the FUSE kernel module, unprivileged users can safely mount their own file systems, even ones they make themselves. To set this up it is simply enough to set the SETUID bit on the program `fusermount` (assuming it's owned by root), which is distributed with FUSE. The server will be running as the unprivileged user so will only have access to what the user has access to, so safety is assured.

The main advantage of FUSE over other similar projects is its large and active user community. The FUSE user community has developed several dozen file systems to date, several of which provide significant functionality to the platforms supported by FUSE. Among the more interesting FUSE file systems are Wayback [5], NTFS-3g [21], and SSHFS [22]. These provide a versioning file system, safe read and write support for NTFS volumes, and a file system based on secure communications over SFTP, respectively.

While FUSE has a large and active user community, FUSE remains limited to Unix-based operating systems. To date, FUSE has either been ported to Linux, Mac OS X, and BSD. There appears to be much desire for a Windows-based FUSE implementation, and there have been several mailing list discussions about it from people not involved in our project. To date, these efforts seem to have progressed little beyond discussion. There are some projects that attempt to provide a facility to implement a Windows file system in userspace, but they each have different APIs and thus are not compatible with FUSE. FIFS [2] is one such project for Windows, and will be discussed in section 4.

Our goal for this project is to provide an interface that is source-compatible with FUSE file systems for Linux, which would instantly make available much the work that has been done using FUSE available to Windows users.

In this paper, we first review some related work. In section 3 we discuss our main implementation efforts on a file system driver. In this section we discuss Windows driver design in brief, contrast it with file systems in Unix, and discuss some design decisions. In section 4, we discuss our second, last-ditch efforts to implement FUSE on top of the previously mentioned FIFS. Finally, we conclude.

2. RELATED WORK

FUSE is a recent example of a general trend in operating system design: reducing the complexity of the kernel. This simplification has been the impetus for two related trends in operating system research: microkernel implementations such as Mach [20] and the MIT Exokernel [8]. Both approaches remove all but the most basic operating system services from the kernel moving it to programs residing in userspace.

Direct Access File System (DAFS) developed by Magoutis et al. is an example of file system implemented in a userspace [12]. The main purpose of DAFS is to increase the performance of network file sharing by reducing the delay of reading and writing by reducing kernel involvement. As a solution, Magoutis et al. designed a userspace file system that uses a virtual I/O interface that directly connects to the network card so there are no system calls occurring. This approach improved both performance and flexibility of network based file systems.

Kantee's Pass-to-Userspace Framework File System (PUFFS) is an example of forwarding file system calls to userspace [11] that is similar in spirit to FUSE. PUFFS is implemented on BSD. It provides API compatibility with FUSE via a library called librefuse.

Arla is an implementation of AFS that resides partly in userspace [23]. It consists of two parts, a small kernel module called xfs which hooks on to VFS and transfers the call to the userspace component, called arlad. Arlad directly communicates with the network interface. Arla has implementations for BSD, Linux, MacOS, and Windows.

FUSE is probably the most well known example of such a file system framework. However, as mentioned in the introduction, it is not available for Windows; this is what our project fixes.

One of the reasons that it isn't available is that file system development on Windows has somewhat of a checkered history compared to other systems. Microsoft has offered the Installable File System (IFS) Kit [14] for some time now that allows third-party file system development; however for the first years of it being offered, it cost $1,000 and consisted of a header with function prototypes and the code to Microsoft's CDFS and FAT drivers with no documentation. The situation has improved since then, and the IFS Kit is now included with the Windows Driver Kit (WDK) for free, along with API documentation [13]. In 1997, to help fill the documentation gap left by Microsoft, Nagar published a

book on NT file system development [15] that remains perhaps the best reference on the topic, despite being about a decade out of date. OSR, Inc. offers the File Systems Development Kit [17] which is a toolkit that makes kernel-mode file system development easier. Unfortunately for us, it is a commercial product which, as of the printing of [2], was priced at $95,000.

However, there is a small literature of projects that have redirected file system requests back up to userspace under Windows. Galen Hunt created a "proxy device" which directs requests up to a service running in userspace [10]. He used this service to implement a FTP file system. It is unclear from his paper how it is implemented, and the project seems to be no longer available. The Arla AFS implementation also made efforts toward a Windows IFS driver plus a large user-mode component, which mirrors their architecture on Linux and BSD [1]. However, even in their latest release, they warn that the windows port is still essentially not working [3].

The more common approach to dealing with the complexities of Windows file system development for research projects is to ignore it entirely and implement what [2] calls a loopback server. This is a server for the Common Internet File System (CIFS; often referred to by the name SMB or Samba) protocol that runs in userspace. The kernel component is already present in Windows, and simply connects to the CIFS server on localhost. Danilo Almeida used this technique in [2] to implement what he called FIFS (A Framework for Implementing User-Mode File systems) which is very similar to FUSE. In fact, we used FIFS in our second effort discussed in section 4. In FIFS, file systems are implemented using Component Object Model (COM) DLLs that are loaded by the server module. The most important way that our work is distinguished from this, as well as the efforts of Galen Hunt, is that we are aiming at compatibility with FUSE itself. In addition, our original goal was to forgo this approach and implement a file system driver directly. However, our current implementation uses FIFS as its back end, building a FUSE-compatible layer above the FIFS API. Finally, the other notable work in this area is OpenAFS [18], which uses a local loopback server for its Windows implementation.

3. IFS IMPLEMENTATION

Most of our efforts throughout the semester have been directed at implementing a solution using the IFS Kit from Microsoft.

We considered a few options for this project. The most obvious is a native IFS file system; this is what we ended up choosing to start. The second option also uses the IFS kit in a less obvious way to create what is called a network redirector. The third option was something like FIFS which uses a CIFS loopback server. It is not clear how the second is used because of the dearth of good material for learning. The third option is not ideal, as will be discussed in section 4.

3.1 Overview of the Windows NT Architecture

Drivers for Windows are implemented using an array of function pointers. These function pointers are analogous to virtual functions in a class or interface from Java or C++. (It's more or less the same as Unix.) These functions are called driver entry functions or major functions. Those of interest to Windows file system developers are [15]

1. Create — the Windows version of Unix's `open` system call, this both creates and opens files
2. Read and Write — self explanatory
3. Directory Control — among other things, this is called to get directory listings, in one of four different formats representing different data
4. File Information — this performs a whole myriad of functions, including deleting, renaming, and linking files; and the equivalent of Unix's `stat` system call, returning information about a given file (in one of 10 formats as of [15] that describe different properties)
5. Cleanup and Close —the former is called when the userspace-held handle to a file is closed; the latter is called when the last NT executive-held handle is closed
6. Flush
7. Volume Information — returns (or sets) information such as the label of a volume, the amount of unused and total space, and attributes of the file system itself (such as whether it is case sensitive or allows compression)
8. FS and Device Control — again a myriad of functions, but in particular used when mounting volumes
9. Fast I/O path functions (see below)
10. Byte Range Locking and Unlocking
11. Opportunistic Locks (oplocks) — used for performance boosting over networked file systems

The main thing to know about how Windows NT drivers are structured is that the I/O system is packet based. Requests are sent to drivers in the form of an I/O Request Packet (IRP), which is a packaged form of the request and supporting data gathered as it goes through the driver stack. The IRPs tell the driver what

to do, and the IRP representing the current request is passed to each driver entry function.

That said, Microsoft found that constructing IRPs for each request produced a very high overhead. To regain performance, they established the *Fast I/O* path. This is a separate set of entry functions more along the lines of the traditional Unix functions, where the information required to carry out the request (e.g. the offset and length of a read and the buffer to put the data into) is passed as parameters. Implementing the Fast I/O path is optional; if a driver doesn't provide Fast I/O entry points, the I/O subsystem simply falls back to using IRPs.

Finally, a note on terminology. When referring to the portion of Windows NT that runs in supervisor mode, this paper will use the term *NT executive*. (What Microsoft calls the *kernel* proper is a much smaller piece, leaving out components such as the I/O Manager, Cache Manager, and Object Manager.)

3.2 Unix vs. Windows

While the more we learn about Windows the more we learn some things are not all that different from their Unix equivalents, there are still a large number of differences between the design tradeoffs and aesthetics of the two systems. Almost all of these differences tilt towards making Windows file system development more complex than Unix. This section summarizes many of these differences.

Without these differences, it would likely have been easier to port FUSE, because (except for licensing issues) we would have been able to make use of some of the existing code for the Linux kernel module.

In general, Windows tends to provide richer, more complex abstractions abstractions than Unix, and these abstractions tend to fall to the device drivers to implement. In other cases, there doesn't seem to be an overall design philosophy that causes the other differences. In addition, Windows file systems have to implement much more functionality in their path name parsing functions, which is a particularly notable point for those wishing to design file systems on Windows. We will start by describing the differences in path name parsing.

3.2.1 Path name parsing

There are two things that Windows file systems have to consider that Unix ones do not. First, Windows file systems do pattern matching themselves. Second, Windows file systems must parse paths into component directories themselves.

In Windows, the work of file name pattern matching moves into the file system. In Unix, when someone issues a command such as `ls file*`, first the shell issues a series of `readdir(3)` calls that go through he directory list to find those that match. Those files that do match are passed as command line arguments to `ls(1)`. `ls` will call `stat(2)` for each of its arguments; those that are successful are displayed to the user. By contrast, in Windows, the equivalent command `dir file*` works rather differently. In this case, the shell does no interpretation of the command line arguments. In addition, and notably, neither does `dir`; it simply passes the string `‘‘file*’’` to the Windows API call `FindFirstFile`. This makes its way into the NT executive and is passed uninterpreted to the file system driver. The FSD then interprets the pattern in whatever way it chooses, and returns one or more files up the chain to `dir`.

This approach has both advantages and disadvantages, mostly that show up when working with very large directories. First of all, the Unix approach removes the ability of the file system to optimize the query. Many file systems (including NTFS and ReiserFS [19, 16]) store directory information in trees; in these systems, doing a linear scan of the directory is suboptimal. For instance, in the example with `file*`, such a file system could find the first and last files that match that pattern in logarithmic time, but the Unix approach doesn't give the file system the ability to optimize.

Over a network, performance can suffer even more, because there is likely one `stat` request per file and thus at least one round trip per file in the directory (that is not in the cache). With the Windows approach, it would be possible to have a protocol such that it sends the pattern to the remote server for matching. The fact that `dir` itself would be issuing requests (as opposed to just interpreting the command line arguments that could have either been typed by hand and thus may be inaccurate) means that it knows that all the returned file names are valid, and there is no need to stat each one individually. (Indeed, even if the user was requesting additional information, say the equivalent of `ls -l`, a FSD can return all that information at the same time it's returning the list if files.)

The second difference in how Windows handles path names is that file system drivers must parse paths themselves in Windows. In Unix, opening a path such as `foo/bar/baz` results in three calls to file system functions; one to open `foo` relative to the current directory (or relative to `/` if an absolute path name is given), one to open `bar` relative to the return from the first call, and finally one to open `baz` relative to the second. In Windows, the entire absolute path name is passed to the FSD; if an application opens a relative path, that is physically concatenated to the current working directory before it is passed to the file system.

The main cost of these differences is increased complexity in the FSD. Instead of the shell implementing pattern matching; the FSD must. Instead of the next

layer up in the kernel doing command line parsing, the FSD must. In also decreases flexibility in some arenas. If you've ever wondered the *real* reason that Windows doesn't allow * or ? in a file name, it's because the FSD needs to be able to distinguish a pattern-matching request from a request for that particular file. Also, what happens if Microsoft wants to add regular expression matching to the `dir` command? It's possible to place it into `dir` itself, of course, but then `dir` should distinguish when it should use its pattern matching vs. the FSD's. The alternative is to move it into the FSD; but then it needs to be moved into *each* FSD, and more reserved characters would need to be taken away.

3.2.2 Abstraction differences

There are a number of places where the actual abstractions Windows provides are slightly different. Windows provides a richer concept of locking than is present in Unix, additional APIs such as the Fast I/O path and defragmentation

Unix implements advisory locking at the granularity of files, whereas Windows (typically) supports mandatory locking on arbitrary ranges of a file, called byte-range locks. While FSDs are not required to support byte-range locks, or can implement them in an advisory nature, Nagir suggests that this is not a good idea as "most Windows-based applications expect locks to be mandatory." Even though byte-range locks can be implemented in a fairly generic manner, and in fact are done so in the Windows code, these APIs are not publicly known so it's up to the FSD to implement locks itself. Windows also supports something called opportunistic locks ("oplocks"), though these are intended for networked file systems, and even then only an opportunity for performance improvement. (One node on a network can take out an oplock on part of a file and know that it can cache it until the oplock is released.)

There are also a number of other optional features that make FSDs more complicated. For instance, file system drivers may implement Fast I/O entry points for improved performance. As another example, FSDs can implement a set of defragmentation APIs by way of special defrag IOCTLs. Under Unix, the task of defragmentation is done (if it can be done at all) by a userspace program. Under Windows, this functionality goes into the FSD, but if it is provided, any program that uses the defragmentation APIs (including the standard Windows one) will be able to work with that volume immediately.

Finally, as a relic from the dark times of MS-DOS, file system drivers are encouraged (though not required) to support DOS-style 8.3 character names. These names provide essentially a second name for the files, and the file and directory information functions return these alternate names.

3.2.3 Other differences

There are other differences that make Windows development more difficult. For instance, Windows drivers are not supposed to block. When a request is submitted to the FUSE kernel module in Linux, it blocks awaiting a reply from the userspace server. In Windows, we had to find another way to accomplish this task. Basically, instead of blocking, Windows driver threads return a code of `STATUS_PENDING`. At a later time, when that request is finished, the driver completes it separately. In some sense this didn't really complicate our design over FUSE's, but it's just one more place where we have to do things differently. (One of the reasons that the asynchronous design of Windows is important for FSDs in general is that when the system is under memory pressure and Windows starts paging data to disk, it does so using a limited number of threads. If those threads block, then the paging rate is slowed and it takes longer to relieve the pressure. This is a consequence of using page files instead of a separate swap partition, thus forcing paging I/O through the file system.)

The fact that Windows is closed source (despite access to the Windows Research Kernel) also makes things difficult of course.

3.3 Kernel-Userspace Interface

There are a number of possible ways that the userspace portion of FUSE-NT and the file system driver can communicate. This section will provide an overview of some of the possibilities.

3.3.1 Local Procedure Calls (LPC)

Windows NT provides a mechanism that is at least conceptually similar to lightweight remote procedure calls [4] in that it provides a faster implementation of remote procedure calls amongst local processes. Windows itself uses LPC for communication amongst different modules, for instance between security components such as Winlogon and the Windows authentication server process LSASS [19]. In fact, Microsoft describes the architecture of Windows NT as a "hybrid microkernel," in that it is structured similar to a microkernel with modules that are isolated in part by LPC interfaces instead of direct procedure calls, but with most services running in kernel mode.

This method would be ideal for communication between the user and kernel mode portions of the code if only it were a documented interface. Enough of it has been reverse engineered that we could probably get away with using it, but we decided that targeting an undocumented, unsupported, and potentially moving interface would be a bad option given our timetable. Should this interface be documented in the future, this is a possible "best option," particularly if LPC calls can be asynchronous.

3.3.2 Named Pipes and Sockets

Another option would be to do communication through a named pipe. One design for FUSE-NT we developed would proceed as follows:

1. The server creates a pipe using the Windows API `CreateNamedPipe`
2. The server passes the name of the pipe to the driver using a custom IOCTL
3. The driver opens the pipe, and communication proceeds from there

The design of named pipes is one area where Windows would actually help us out in comparison to Unix. In Unix, there are two kinds of pipes: named and unnamed. Unnamed (or anonymous) pipes wouldn't be possible as the file descriptor the server creates wouldn't have meaning within the driver. (There might be some way to make this work, but we suspect that any such solution would be more complex than the following alternatives for little benefit.) In Unix, named pipes exist in the file system. As such, without special care some cleanup would be required in case of a server crash. In addition, both types of pipes in Unix only allow one-way communication, so two pipes would be necessary.

Windows differs on all of these points. First, though there are Windows APIs for creating anonymous pipes, they are implemented behind the scenes as named pipes with random names. (This doesn't affect our solution, it's just a curiosity.) Second, in Windows, named pipes exist only in the object namespace, not in the file system. They are reference counted and automatically deleted when the last reference goes away, removing the need to explicitly collect garbage. Third, named pipes (not anonymous pipes) can be full-duplex, which removes the need for two pipes.

Although we did not choose this option, we feel that it would not be unreasonable to implement a communication mechanism using this in Windows. (Under Unix the cleanup problem provides an argument, albeit a small one, against this method.)

3.3.3 Read/Writes to the Device

This is the approach that FUSE takes on Linux. The server opens `/dev/fuse`, and after "introducing" itself, the sequence of events is:

1. The server calls `read(2)` on the file descriptor to the device
2. The read call makes it to the FUSE kernel module
3. The thread serving this read request blocks on a semaphore
4. An application makes a request for a file provided by FUSE
5. The application's request makes it to FUSE
6. FUSE gathers the data from the application's request that the server will need to satisfy it, and marshals it into a structure the server understands
7. FUSE wakes up the server's read thread, and returns the marshaled request as the data
8. The application's request thread is blocked on a semaphore
9. The server wakes up and services the request
10. The server calls `write` to return the result to the kernel module
11. FUSE wakes up the application's request thread, and returns the data back to the application
12. The server calls `read` again for the next request

Conceptually, the same approach is possible on Windows. The device name would change to `\Devices\Fuse` and the API calls would change names, but the biggest difference is the fact that the Windows kernel threads cannot block. For a discussion of how to handle this, see section 3.3.4.

Again, this is a very reasonable approach (evidenced not the least by the fact that FUSE itself uses it). However, at least one of the authors doesn't like the fact that the server needs to make both read and write calls into the driver and the fact that the terms "read request" and "write request" become ambiguous. (Does a read request refer to an application request or server request?) These minor thorns can be fixed by the following design.

3.3.4 IOCTL to the Device

This approach is very close to the previous one, except that instead of calling `ReadFile` or `WriteFile`, the server would call `DeviceIoControl` with a custom IOCTL. Because data can be sent both directions during an IOCTL, one system call can be used to both return the result of a previous request and get the next request; in other words, steps 10 and 12 merge.

This is the approach that we initially took, which meant that we needed to address how to handle the non-blocking aspects. As it turns out, this is not difficult. Instead of blocking above, we place the IRP onto a queue and return `STATUS_PENDING`. Instead of unblocking, we remove the IRP from the queue and complete it. The main drawback to this approach as compared to the previous is that it would make it impossible to use the existing libfuse library without modification. (This is unlikely in any case, but changing the interface from read and write to IOCTLs would increase the amount of churn needed.)

A diagram of our approach (slightly simplified) appears in figure 3.3.4.

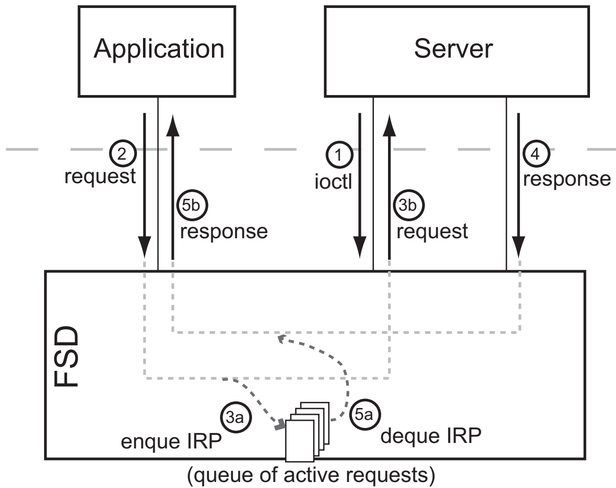

Figure 2: *The flow of data in FUSE-NT.* The server makes an IOCTL request of the driver (1); the IRP is remembered so it can be responded to later. When an application makes a request (2), the FSD adds that request to a list of outstanding requests (3a) and completes the IOCTL remembered from before (3b). The server does what it needs to do to satisfy the request, then makes another IOCTL with the results (4). The FSD finds the corresponding request IRP in the queue (5a) and completes it (5b).

3.4 Other Compatibility Issues

The goal of this project is to provide source compatibility with existing FUSE file systems. While it's possible to provide the same interface through the technique described above or in section 4, this isn't enough. Almost any non-trivial file system made for FUSE would also use Unix system calls, so we have to provide a way for these to work.

There are two main possibilities. The first possibility is to leverage the work of the Cygwin project [7] which provides Unix API and system calls for Windows. FUSE-NT compatible programs would be compiled against Cygwin headers that do a translation; for instance, from a `open` call to `CreateFile`. The programs that are produced are true Win32 binaries; this may be good or bad. On the positive side, it would be possible to create a "file system" that actually used some Windows APIs itself, perhaps to provide a GUI for some reason. On the negative side, it means that the Unix emulation has leaks, which means that the file system could make untrue assumptions. (For instance, the Windows API always presents a case-insensitive view of file systems. If the FUSE Server expected a case-sensitive view, it will fail.)

The second possibility is to use Microsoft's Subsystem for Unix-based Applications[2] (SUA). This is a "subsystem" for the NT kernel that provides POSIX system calls, and comes with a collection of utilities such as `ls` and an NFS server [6].

The Windows NT system call interface is not provided by the Windows API; instead, the NT system call interface, known as the Native API, is largely undocumented and completely unsupported for use by applications. The Windows API is provided by the *Windows subsystem*, which provides a translation from the Windows API to the Native API when it needs to make a system call. (Not all Windows API functions require system calls; some potentially require more than one.) The SUA subsystem is much the same thing, except that instead of providing translation from the Windows API, it translates POSIX system calls to the Native API.

SUA implements POSIX.1 and POSIX.2[3] [6], including features that the Windows subsystem hides. For instance, SUA can use case-sensitive file names (when on an underlying file system that supports it, which includes NTFS but not FAT) and implements the setuid bit on applications. The downside is that Windows API functions are unavailable from SUA applications.

If FUSE-NT were to be implemented as a FSD, the server's approach is largely irrelevant. Possibly excepting the LPC approach[4], all of the ideas mentioned above for communication would work equally well with both approaches. We were planning to use SUA first, but there's no compelling reason why we should have chosen this.

However, once we changed to the FIFS approach we are constrained to use Cygwin. FIFS itself is a Windows program, and it's impossible to run half of a program in the Windows subsystem and half in SUA, which means that our program needs to run entirely in the Windows subsystem. (The alternative is to set up communication, say through a named pipe, from a SUA process to the FIFS process.)

With only minor modifications (there are a couple function prototypes missing in each case and one must stub out two FUSE functions since we ran out of time to integrate it with FIFS), SSHFS compiles under both Services for Unix and Cygwin.

[2]This has gone by many names in the past: OpenNT, the POSIX Subsystem, and Interix; it is a part of Services for Unix (SFU), which adds common Unix command line utilities.

[3]Note however that this does not mean that they are implemented entirely usefully; POSIX allows functions to return a "not implemented" error. However, a scan of the support system calls suggests that enough are there to run many useful file systems, and there are a number of packages available that run on SUA, including OpenSSH.

[4]Remember, "undocumented" means anything can happen.

4. FIFS IMPLEMENTATION

Our switch to a FIFS-based implementation can be largely attributed to the general complexity of implementing a Windows FSD that mimics the behavior of a Unix-based kernel module. It is not only the differences between Unix and Windows that made a port difficult, but also accurately understanding and retaining knowledge about the Windows API and how to use it in the context of a FSD.

Examples of these complexities have something of a taxonomy. The particular problems that we were having can mostly be grouped into the following categories: complex function interfaces, structure overload, and documentation issues. Some representatives from each group: the Windows function `CreateFile` takes seven parameters and allows for countless different ways to create or open a file, the `VOLUME_LOGICAL_OFFSET` structure has only one field (surely the structure could be eliminated), and functions like `NtCreateIoCompletion` are not even mentioned in Microsoft's public documentation.

Our team ultimately decided that we could not realistically implement even a rudimentary Windows FSD within the given amount of time. Accordingly, we investigated other options that might allow us to avoid kernel programming entirely. The most promising option that we know of is the FIFS project [2]. This now serves as the basis for our prototype implementation.

We originally had trouble running FIFS on Windows XP[5], so we decided to try to mimic the environment that Almeida had. We surmised this was Windows NT 4.0, so this is what we tested under. It is likely that it works under Windows 2000 as well, but this is untested.

4.1 FIFS Overview

Our current implementation is built upon the FIFS project[6] implemented by Danilo Almeida for his master's project at MIT [2]. It actually provides a service very similar to FUSE, but with a radical different presentation.

Instead of having a kernel module that forwards requests, FIFS implements a network loopback server for the Common Internet File System[7] (CIFS). Accessing a FIFS file system just consists of opening a network path like one would for a typical remote file system in Windows, except that it will be connecting to the local machine.

FIFS file systems run entirely within one process, however the file system code proper (the equivalent of FUSE's userland servers) is stored in DLLs. During startup, FIFS looks at a registry entry to determine what DLL to load, and links it into its address space with the `LoadLibrary` function.

Our design simply adds a new layer on top of FIFS that does translation between the API that FIFS provides and the API that FUSE file systems expect. Figure 4.1 illustrates this design.

Figure 3: The flow of data in the FIFS version of FUSE-NT.

4.2 FUSE Adapter

The main part of implementation is to translate the API that FIFS file systems use to the one that FUSE file systems use. The set of APIs are functionally similar, but the way they organize and manage the pointers, option, and state of files/directories are different.

The foremost difference between FIFS and FUSE are the way that they open and manage files and directories. The FIFS CIFS server indicates the location of the file or directory using an internal file handle called `fhandle_t`. On the other hand, FUSE-based file systems accept a path string as a location and use a unique data structure called `fuse_file_info` (we will abbreviate this as `ffi`) to store state of a file. While the `ffi` structure contains a file handle that the file system can use, this value is opaque to the actual FUSE implementation. This is in contrast to FIFS, where the supporting code slightly manipulates the handles. Therefore, the FUSE adapter must have a way of mapping the `fhandle_t` values to pairs of pathnames and `ffi` structures. Figure 4.2 is pseudocode for how we adapt a FIFS `create` call to a FUSE `open` call. In this code, the `handle_info_t` serves this purpose.

Ironically, the situation in FIFS is actually almost exactly the reverse of what goes on in the kernel. FIFS is prepared to open files by the name of the file relative to the `fhandle_t` of the containing directory (though it seems to always give a full path relative to the root directory, file handle 0), while FUSE file systems are provided the full pathname. As it turns out this seems to be a good choice for FUSE; many FUSE file systems would need to reconstruct the full path name if it were

[5] It appears that XP uses a newer version of CIFS than FIFS is prepared to handle.

[6] "FIFS" stands for *A Framework for Implementing User-Mode File Systems*.

[7] Also known as the Server Message Block protocol (SMB).

```
DWORD
FsDT_Win32::create2(
    string path,
    UINT32 flags,
    fattr_t* fattr,
    fhandle_t* phandle
    )
{
    // Changes \ to / and upper to lowercase
    translatePathname(path);

    DWORD error = NO_ERROR;

    fhandle_t h;
    handle_info_t hinfo;

    if(path is a file and not a directory) {
        if(flags are invalid) {
            error = ERROR_INVALID_PARAMETER;
            goto cleanup;
        }

        hinfo.ffi = new fuse_file_info;

        // Turn Windows into Unix flags
        hinfo.ffi->flags =
          translateAccessMask(flags) |
          translateCreateMask(flags);

        // Calls into the actual FS
        int ret = Fuse::open(hinfo.pathname,
                             hinfo.ffi);

        if(ret != 0) {
          error = translateOpenError(ret);
          goto cleanup;
        }
    }

    // Remember the path and ffi for later
    h = get_new_handle();
    handles[h] = hinfo;
    handles_used++;
    *phandle = h;

    return ERROR_SUCCESS;

cleanup:
    return error;
}
```

Figure 4: Translating a Create/Open Call

```
unsigned translateFIFSModeMask(UINT32 mask) {
    unsigned short fuseMask = 0;

    if ( ATTR_SYMLINK & mask )
        fuseMask |= S_IFLNK;
    if ( ATTR_DIRECTORY & mask )
        fuseMask |= S_IFDIR;

    if ( ATTR_READONLY & mask )
        fuseMask |= S_IRUSR|S_IRGRP|S_IROTH;
    else
        fuseMask |= S_IRWXU|S_IRWXG|S_IRWXO;

    return fuseMask;
}
```

Figure 5: Translating Attributes

not provided. For instance, SSHFS would want the full path to pass to `scp`. One of the example file systems that comes with FUSE simply concatenates the given path name onto the end of an existing path it is mirroring, then issues the appropriate system call on the resulting file name.

Another difference is the way file attributes are handled. FUSE file systems use Unix attributes (e.g. `S_IFDIR` to indicate that the given file is a directory), while FIFS uses Windows flags (e.g. `FILE_ATTRIBUTE_DIRECTORY` for the same thing). The basic set of attributes, such as symbolic links, directories, and to some extent read-only files are the same between Windows and Unix, though some may require extra work; for instance, read-only requires looking at the RWX bits, for instance. However, there are some special attributes that Windows and hence FIFS support that Unix and FUSE do not. These include hidden, compressed, and archived files. We currently ignore these attributes that do not exist in the FUSE interface; the function that retrieves attributes for a given file will never set them. Figure 4.2 gives an example of how we translate attributes.

Similarly, option values passed to the file system when opening or creating a file are also different. While most flags have nice correspondences between FIFS and FUSE, there are some mismatches. For instance, the Windows `CREATE_ALWAYS` flag will always create a new file, overwriting the existing one if necessary; Unix doesn't have exactly equivalent behaviorFor these cases we approximate as best as we can. (In some cases, such as `FILE_FLAG_RANDOM_ACCESS`, we just ignore them.) Figure 4.2 shows how we translate some of the flags that the Windows `CreateFile` function accepts.

Finally, we also changed the architecture FIFS uses a bit. Instead of starting the FIFS server and letting it load the file system as a DLL, we build the file system and server into one monolithic EXE. The file sys-

```
int translateCreateMask(DWORD mask) {
    int fuseMask = 0;

    switch (mask & CREATE_MASK) {
        case CREATE_ALWAYS:
            // This flag is unsupported in Unix
            // This is pretty close though
            fuseMask = O_CREAT | O_TRUNC; break;

        case CREATE_NEW:
            fuseMask = O_CREAT | O_EXCL; break;

        case OPEN_ALWAYS:
            fuseMask = O_CREAT; break;

        case OPEN_EXISTING:
            fuseMask = O_EXCL; break;

        case TRUNCATE_EXISTING:
            fuseMask = O_TRUNC; break;
    }
    return fuseMask;
}
```

Figure 6: Translating File Creation/Opening

tem's `main` function begins execution, and it then calls `fuse_main`. This function is now implemented to start the server by calling the original FIFS entry point. We did this so that the file system could receive command line arguments. An alternative would be to build the FIFS server as a DLL and have `fuse_main` load that and start it that way.

4.3 Hello World FUSE FS

We used the *Hello World* FUSE example file system as a testing and demonstration mechanism. It defines four functions: `getattr`, `readdir`, `open`, and `read`. Despite its simplicity, the Hello World example provides our FIFS and FUSE translation function with good tests while leaving the actual Hello World code almost unchanged. While it is far from a fully-featured file system, we feel that the core elements are there and they can demonstrate our implementation. The remaining operations would be largely straightforward to implement.

The Hello World file system is very simple. In the directory under which it's mounted, there is a single file called `hello`. Reading the file produces the contents `Hello World!`.

This file system runs almost unmodified with our FUSE implementation. In fact, really no substantive changes at all were needed. The only modifications from the original example to our current implementation are:

1. A change in the name of the `fuse.h` header file
2. Additional debugging output using macros from FIFS
3. A workaround for the use of GCC/C99 syntax that Visual C++ 6 does not support

The last one simply changed the following declaration which sets up function pointers to the specified operations:

```
static struct fuse_operations hello_oper = {
    .getattr = hello_getattr,
    .readdir = hello_readdir,
    .open = hello_open,
    .read = hello_read,
};
```

so that the members were initialized in an `init()` function called from `main()`.

4.4 Problems With the FIFS Approach

There are a few reasons why the CIFS loopback server approach is not ideal.

One problem that we ran into was that the CIFS client will sometimes capitalize names before sending them. For instance, in the Hello World example, when opening `/hello`, the CIFS server sends a request for `/HELLO`. The Hello World function `hello_open` does a case-sensitive comparison between `/hello` and `/HELLO`, determines that they are not the same, and returns a file not found error. Almeida provides an explanation in [2] about exactly when and why this occurs. As a stop-gap measure, we take the same approach the examples in the original FIFS work, and simply change all file names to lowercase. This means that we would be unable to open files containing capital letters on a case-sensitive FUSE file system.

Another potential problem is performance. While we did not measure the performance of our system, Almeida did, and found that the read performance was hurt tremendously. He attributes this to NT not using the buffer cache on reads, but always going up to FIFS. (It's possible that enhancing the FIFS SMB server to use oplocks would help this.)

One other significant problem is that, at least with the implementation FIFS uses, a network cable must be plugged into the computer. FIFS uses NetBEUI names, and the NetBIOS interface will return an error if the computer is disconnected. Judging from the configuration of Windows machines run by the University of Wisconsin's Computer Science Department, OpenAFS creates a new virtual network interface called AFS. It's possible that the purpose of this is to avoid this problem.

There was also another problem caused by the specifics of the Hello World file system, which is that it is read-only. When trying to read `/hello`, the request always came over requesting `GENERIC_ALL` permissions,

which includes both reading and writing. (This was true even for "obviously" read-only utilities such as the Windows shell's `type` command. Because it wasn't opened `O_RDONLY`, the Hello World server would fail with an access denied error. To get around this, we simply disabled the access check by explicitly passing `O_RDONLY` to the open routine. (If the user tries to write to the file at a later point, the write produces an error.)

It's unclear where these permissions originate from: the applications really opening files with all permissions, in the CIFS client, or in the FIFS server. In either of the latter cases, moving from a FIFS-like approach to a native FSD would solve the problem. In the former case, then a policy decision has to be made whether to fail such calls when they are opened (thus affecting a lot of applications unnecessarily) or to wait until a write request is given.

5. CONCLUSION

File system research is a slow process. Merely having a novel idea is a trivial first step, while implementing the design requires large investments of time spent researching, programming, testing, and experimenting. The majority of time spent implementing new operating system functionality, including file systems, is typically devoted towards implementing the new functionality in the context of the operating system and its APIs.

Operating systems researchers have long known of the inherent complexity of conducting their research, and efforts have been made to attempt to lessen the costs of implementation. One result of these efforts has been the idea of removing functionality traditionally seen encapsulated within a kernel The FUSE project has designed a service that allows file systems to reside outside of a Unix kernel, and it presents a greatly simplified API to researchers wishing to implement a file system.

The goal of our project this semester has been porting FUSE to Windows-based operating systems. In the course of doing so, our team has experienced the realities of file system implementation first-hand, in addition to the added difficulties introduced by the inherent differences between Unix-based operating systems and Windows. Our original implementation attempted to follow the FUSE design as closely as possible and unsurprisingly required a large amount of time spent becoming familiar with the Windows Driver Kit API. Our second implementation idea of implementing what amounts to a FUSE emulation service over CIFS was largely in response to time constraints.

Despite the inherent complexities and time investments, we believe that our original approach is conceptually and technically more correct than implementing FUSE-like functionality over CIFS.

Acknowledgements

We would like to thank everyone who helped working on this project, especially our advisor Michael Swift for giving us great advice and providing us with important and incredibly useful resources. Also, we would like to thank people who gave us a review of our rough draft with a critical and useful comments.

References

[1] Ahltorp, M., Hornquist-Astrand, L., and Westerlund, A. Porting the Arla file system to Windows NT. *MADE2000–Management and Administration of Distributed Environments Workshop, May* (2000).

[2] Almeida, D. FIFS: A Framework for Implementing User-Mode File Systems in Windows NT. *Proceedings of the third USENIX NT Symposium* (July 1999).

[3] Arla version 0.90. Online, March 2007. ftp://ftp.stacken.kth.se/pub/arla/arla-0.90.tar.gz.

[4] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. Lightweight remote procedure call. *ACM Trans. Comput. Syst. 8*, 1 (1990), 37–55.

[5] Cornell, B., Dinda, P., and Bustamante, F. Wayback: A User-level Versioning File System for Linux. *Proceedings of the USENIX Annual Technical Conference, FREENIX Track* (2004), 19–28.

[6] Corti, S. P. Introducing Microsoft Windows Services for Unix 3.5. Online, May 2007. http://www.corti.com/downloads/Microsoft/Events/2005-02_LOTS/lots_win_sfu_35_saschac.ppt.

[7] Cygwin. Online, May 2007. http://www.cygwin.com.

[8] Engler, D., and Kaashoek, M. Exterminate All Operating System Abstractions. *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems* (1995), 78–94.

[9] Filesystem in userspace. Online, March 2007. http://fuse.sourceforge.net.

[10] Hunt, G. Creating user-mode device drivers with a proxy. *Proceedings of the USENIX Windows NT Workshop* (1997), 55–59.

[11] Kantee, A. puffs - Pass-to-Userspace Framework File System. *Proceedings of the AsiaBSDCon 2007, Tokyo, Japan* (2007), 29–42.

[12] Magoutis, K., Addetia, S., Fedorova, A., Seltzer, M., Chase, J., Gallatin, A., Kisley, R., Wickremesinghe, R., and Gabber, E. Structure and Performance of the Direct Access File System. *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA* (2002), 1–14.

[13] Microsoft. About the windows driver kit (WDK). Online, March 2007. http://www.microsoft.com/whdc/devtools/WDK/AboutWDK.mspx.

[14] Microsoft. IFS kit — installable file system kit. Online, March 2007. http://www.microsoft.com/whdc/DevTools/IFSKit/default.mspx.

[15] Nagar, R. *Windows NT File System Internals*. O'Reilly, 1997.

[16] Namesys. Three reasons why ReiserFS is great for you. Online, May 2007. http://www.namesys.com/X0reiserfs.html.

[17] Open Systems Resources. File systems development kit (FSDK). Online, March 2007. http://www.osr.com/toolkits_fsdk.shtml.

[18] OpenAFS. Openafs for windows requested features and road map. online, May 2007. http://www.secure-endpoints.com/openafs-windows-roadmap.html.

[19] Russinovich, M. E., and Solomon, D. A. *Microsoft Windows Internals*, 4th ed. Microsoft Press, 2005.

[20] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating System Concepts*, seventh ed. John Wiley and Sons, 2004, ch. Appendix B. http://codex.cs.yale.edu/avi/os-book/os7/online-dir/Mach.pdf.

[21] Szakacsits, S. About NTFS-3g. Online, March 2007. http://www.ntfs-3g.org/about.html.

[22] Szeredi, M. SSH filesystem. Online, March 2007. http://fuse.sourceforge.net/sshfs.html.

[23] Westerlund, A., and Danielsson, J. Arla-a free AFS client. *Proceedings of the 1998 USENIX, Freenix track* (1998).