Self-Certifying File System Implementation for Windows

by

David Euresti

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2002

Author .....
Department of Electrical Engineering and Computer Science
August 13, 2002

Certified by .....
M. Frans Kaashoek
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
David Mazières
Assistant Professor at New York University
Thesis Supervisor

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

Self-Certifying File System Implementation for Windows

by

David Euresti

Submitted to the Department of Electrical Engineering and Computer Science
on August 13, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

SFS for Windows is a Microsoft Windows 2000 client for the Self-Certifying File System. It is a direct port of the original using Cygwin. It uses a commercial NFS client to connect Windows to SFS. It provides support for symbolic links in Cygwin by converting these into Windows Shortcuts. It automounts file systems in Windows using a mountd daemon and the portmapper. The client also includes a new system to pass file descriptors between Cygwin processes. All this was accomplished with minimal code changes to the original UNIX code.

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

Thesis Supervisor: David Mazières
Title: Assistant Professor at New York University

Acknowledgments

I would like to thank David Mazières for his invaluable help and M. Frans Kaashoek for his support of this project. I would also like to thank Paul Hill, Danilo Almeida, and the entire WinAthena team for their expert advice. Finally, I am deeply grateful to my family for putting up with me during this process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Self-Certifying File System (SFS)[6] is a file system with a high level of security built over NFS. SFS has three goals: security, a global namespace, and decentralized control. It runs on a wide variety of UNIX platforms but previously was unavailable on Windows.

An SFS client for Windows is desirable because of the widespread use of the Microsoft Windows platform. With an SFS client a Windows machine could obtain files securely and remotely from UNIX SFS servers.

One of the key features of its design is that SFS runs in user space and does not depend much on the kernel. In this thesis we show how this user-level implementation can be ported to Windows.

One of the most important goals of this project is to keep the code as similar to the original as possible. SFSNT[9], a previous implementation of SFS for Windows became unusable because it was difficult to maintain and after a while did not work with the newest versions of SFS. By staying close to the UNIX code any updates to the UNIX code can be folded back into the Windows code easily.

1.1 SFS Architecture

SFS consists of a number of programs on the client and server side. This thesis only concerns with porting the SFS client and its supporting libraries. While the SFS

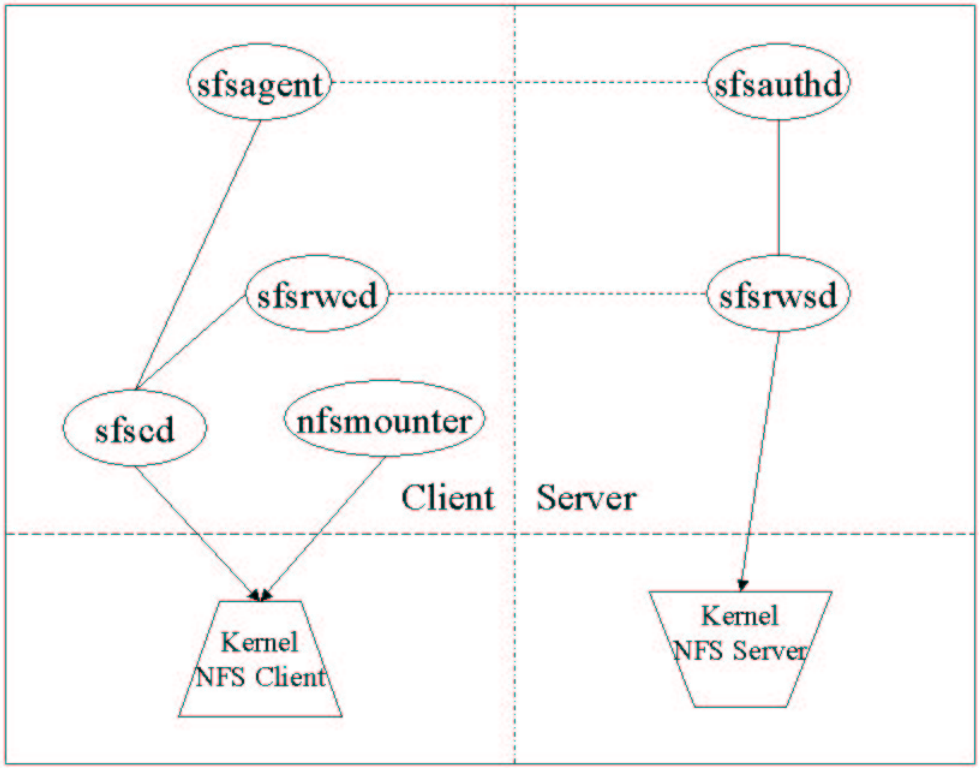


Figure 1-1: SFS System Overview

server compiled, it was not tested and is left as future work.

SFS is written in C++ with a strong use of templates to allow for asynchronous calls and simple function currying. SFS communicates between modules using SUN RPC. SFS is implemented as a loopback NFS server. This means that the SFS client runs on the local machine and pretends to be an NFS server. It responds to NFS RPC calls by converting these calls into SFS RPC calls and sending them out over the network. This process takes the file system development out of the kernel since the loopback NFS server can run in user space.

The client is implemented as a suite of four programs (See Figure 1-1). The first is an NFS Mounter, "nfsmounter." The nfsmounter handles the creation of NFS mount points for loopback servers in directories of the local file system. It runs as a separate process to help clean up the mess that loopback NFS servers may leave behind should they break. In an event of system failure the nfsmounter responds to all existing messages with errors and unmounts the file systems. This allows developers to not have to reboot their machines after a file system breaks.

The second part is the automounter, "sfscd". The automounter mounts SFS file systems as they are referenced. This means that the user can simply change to a directory and they will be accessing files on a different server. The automounter runs as a loopback NFS server. NFS automounters are challenging to develop because if a LOOKUP RPC is unanswered NFS clients will lock any access to the rest of the directory. To correct this situation, the automounter never delays a LOOKUP RPC but instead redirects you to a symbolic link where it will delay a response to a READLINK RPC until the file system has been mounted.

The third part is the actual SFS client, "sfsrwcd". The client is also implemented as a NFS loopback server. To avoid the pitfalls of NFS loopback servers, this server is not allowed to block and runs asynchronously using an asynchronous library written specifically for SFS.

The fourth part is a user agent that holds credentials needed by the client, "sfs-agent". The agents runs as a separate process to allow users to write their own agents if they want a different authentication scheme or do not trust a particular agent. In this way the authentication scheme is separate from the actual file access. When the SFS client needs to add authorization to a message it sends the message to the agent and the agent will sign the message. In this way the SFS client never has access to the user's private keys.

In UNIX one process can pass another process a file descriptor across UNIX Domain Sockets using the sendmsg system call. This allows one process to open a connection and then hand it off to another process to finish it. This is especially useful with servers; The main process receives a new connection and hands it off to a worker process. The main process can then keep listening for new connections. SFS uses this feature in many places. The nfsmounter keeps a backup copy of the connection to the client in case the client dies, and the client gets a file descriptor from the agent so that they can communicate. An SFS program, *suidconnect*, obtains a file descriptor securely and sends it back to you using this process.

The sendmsg system call, used to pass file descriptors, works as follows. A sender process calls sendmsg passing it some data, and a file descriptor. The receiver calls

recvmsg and gets the data and the file descriptor. Passing file descriptors only works over UNIX Domain Sockets.

1.2 Contributions

This thesis contributes a Windows port of SFS. The porting process was simplified by using Cygwin. The approach taken to port SFS can be used to port other software projects to Windows.

The Windows SFS client is fully interoperable with UNIX SFS servers. It supports encryption and authentication of clients and servers just like its UNIX counterpart. It supports symbolic links to files and directories stored on either the same server as the link or on a different server.

We contribute a system to automount SFS servers in Cygwin through a mountd daemon that allows a simple shell script to perform the mounting and unmounting of loopback NFS servers. We also contribute a PCNFSD server for loopback NFS servers that uses Cygwin UIDs. The translation of symlinks to Windows Shortcuts can also prove useful in other UNIX software packages.

The thesis also contributes to the Cygwin project. A system to pass file descriptors between processes, previously unavailable in Cygwin, was designed and implemented and behaves like its UNIX counterpart.

1.3 Organization

Since this research is from an already produced system the line between Design and Implementation is unclear. To that effect Chapter 2 describes several approaches to porting SFS to Windows and reasons why they were rejected. Chapter 3 describes the approach that was chosen, how the approach was implemented and what problems surfaced during the process. Chapter 4 evaluates the performance of the SFS client on Windows. Finally, future work is considered in Chapter 5.

Chapter 2

Design Alternatives

There are many choices to be made when porting an application from UNIX to Windows. In some situations, the best solution is obvious, in others they are just decisions that we must live with. Careful analysis will provide the most informed decision. The two important decisions to make are how to implement the file system, and what programming environment to select. All the approaches in this section were discarded because they required too many code changes or were incompatible with the current SFS system.

2.1 File Systems on Windows

Windows does not natively support NFS. This shortcoming is unfortunate because the SFS client runs as an NFS server that the NFS client connects to (See Figure 1.1). This means that some bridge will need to exist to link Windows with SFS.

2.1.1 CIFS Protocol

Microsoft Windows natively supports the Common Internet File System (CIFS)[4]. Unfortunately, CIFS is not really all that common as only Windows uses it. To make matters worse, Microsoft's only Internet Draft of the specification expired 3 years ago. The Samba project [13] has implemented CIFS in UNIX by reverse engineering

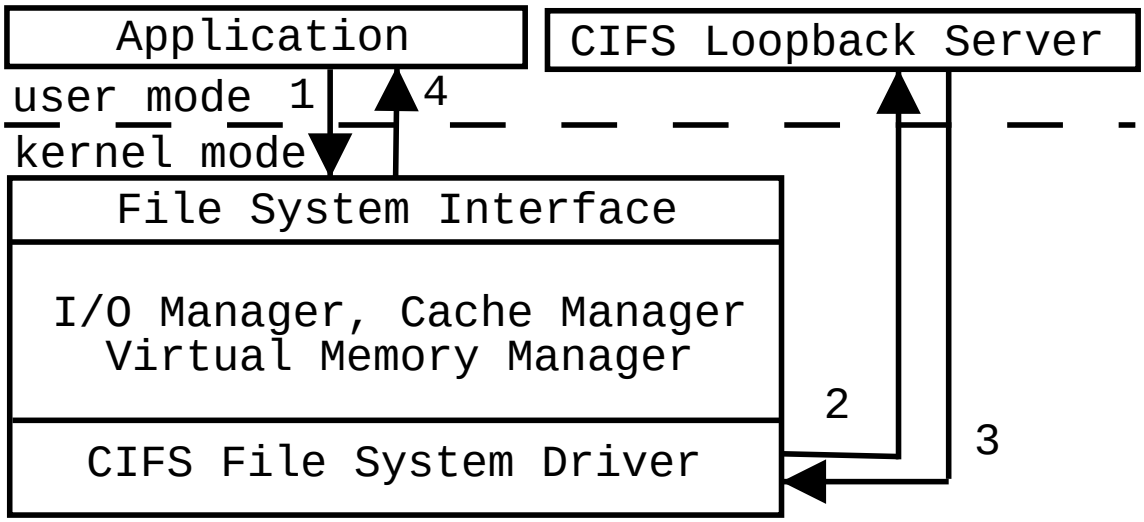


Figure 2-1: CIFS Loopback Server File System Architecture

the Windows version.

CIFS is a file system that preserves state; different from NFS, which is stateless. CIFS also depends on Netbios, a decentralized naming protocol, for its naming scheme. It does not rely on DNS.

Many different projects use loopback CIFS servers (Figure 2-1) to implement file systems. With the newest versions of CIFS (in Windows 2000 and later), Windows lost the ability to run multiple CIFS servers on one machine. Microsoft's solution is to add a setting to enable a CIFS Device (a one line change in the registry) or to run the CIFS server on a Loopback Adapter (a virtual device). Unfortunately, adding the setting to enable the CIFS Device causes Group Policy Object access to break and access to Windows Domains will fail.

To use a Loopback Adapter you must first install the Device Driver. The CIFS server then needs to be modified to bind itself to the correct adapter, and the hosts file must be modified so that the CIFS client can find the correct address.

One idea is to create a loopback CIFS server that translates CIFS calls into NFS calls. Unfortunately this solution would cause all file system calls to go through two programs, the CIFS loopback server and the NFS loopback server, requiring twice the number of context switches.

OpenAFS[12] is a "free" version of AFS that IBM released. It works as a Loopback

CIFS server and is used daily by many individuals around the world.

Work done at MIT by Information Systems [8] provided a solution to the Group Policy problem by using a Loopback Adapter. It detects if the adapter is installed, binds to the correct address, and changes the hosts file appropriately. This version has been used by MIT for a few years now.

We considered using the CIFS server code that AFS uses and write a file system to bridge to NFS. Unfortunately, not only did extracting the CIFS code prove to be difficult, since AFS RPCs are very different from NFS RPCs it made the conversion of these laborious.

Danilo Almeida implemented a Framework for Implementing File Systems for Windows NT[1]. This is a loopback CIFS server that runs on Windows machines. Its goal was to allow developers to create user mode File Systems on Windows, akin to [7]. It allows for stackable file systems, in which you add one file system on top of another. SFSNT was built using FIFS.

One could use FIFS to bridge the gap between CIFS and NFS. Unfortunately FIFS is encumbered by some Microsoft proprietary header files and has not been tested with the newest versions of Windows.

2.1.2 Kernel Mode Driver

Another way to implement a file system in Windows is to go down to the kernel. Microsoft sells an Installable File System Kit (IFS Kit) to develop file systems for Windows. For about $1,000, Microsoft will give you a header file, *ntifs.h*, and two sample drivers.

Kernel driver development in Windows is difficult because of various reasons. The driver cannot access traditional operating system services available to user mode programs and must comply with kernel paging restrictions. Additionally, the driver is difficult to debug because it runs in a single address space with other drivers and must be debugged using special debuggers. Also, while traditional drivers usually only interact with the I/O manager, a file system driver must also interact with the cache manager and the virtual memory manager (See figure 2-2)

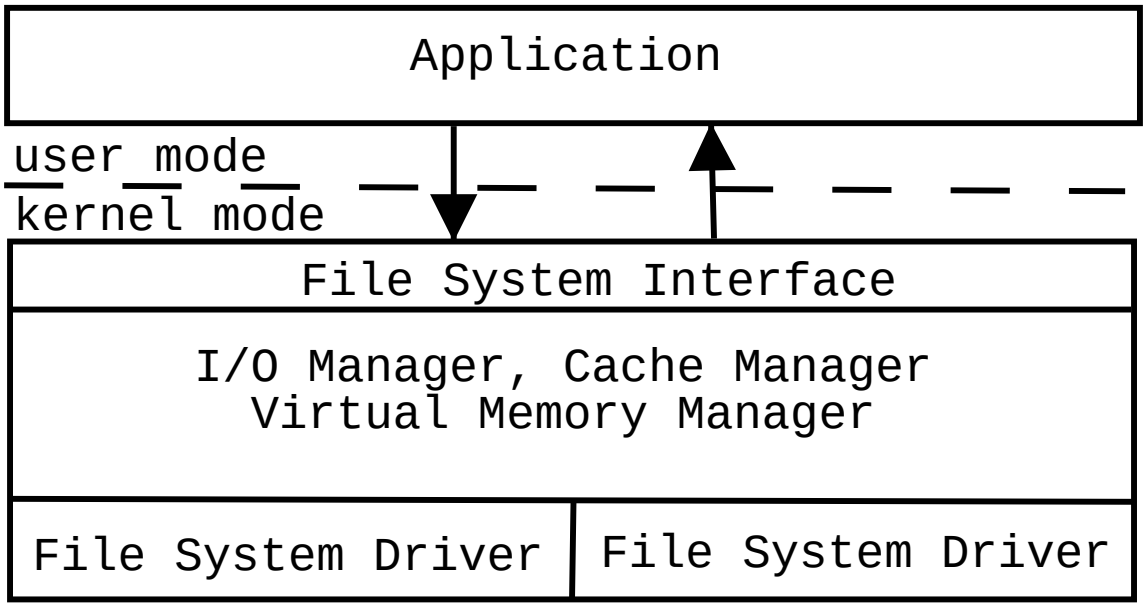


Figure 2-2: Microsoft IFS Architecture

Bo Branten has reverse engineered the IFS Kit [2] and has a GPL version of the *ntifs.h* header file on his website. Several people have taken the challenge to create file systems for Windows using this free software. Unfortunately all the file systems are for physical devices and no attempt has been made to make any network file systems.

Open System Resources (OSR) [14] created a file system Development Toolkit to make it easier for developers to create file systems. The Kit is a library that exports an interface very similar to the VFS interface on UNIX and comes with full documentation and support. The kit costs $95,000 for commercial use but it appears they have a more inexpensive version for research institutes. Coda [15], a file system developed at CMU, has a Windows port built using the FSDK. Unfortunately, we did not find out about the less expensive version until late into the project.

The Arla project is a project to create an Open Source implementation of AFS. Using the IFS Kit, they have ported their XFS file system to Windows NT [11]. We were disappointed to find out that currently this port of XFS only supports reading and does not support writing.

```
// primary template
template <class T, class A> class sample {
  // stuff
};

// partial specialization
template <class A> class sample<bool, A> {
  // stuff
};
```

Figure 2-3: Partial Specialization of Templates

2.2 Programming Environment

The question of which Programming Environment to use is very important. SFS uses automake and libtool for its build system; it would save a lot of time if this could be preserved. Also the compiler must be selected correctly because the asynchronous library uses many templates, which some compilers do not fully understand.

2.2.1 Microsoft Visual Studio

Microsoft Visual Studio is an Integrated Development Environment for Windows. It provides the developer a nice user interface and organizes development into projects and workspaces. Visual Studio also has the ability to process makefiles. Unfortunately Microsoft's makefiles are different from GNU's so they are not compatible.

Another big problem is that Template support in the Microsoft compiler is not up to date. Microsoft's compiler does not support Partial Specialization of Templates (See Figure 2-3). This is vital for the asynchronous library's pseudo function currying system.

2.2.2 Intel Compiler

Intel makes a C++ compiler for Windows that supports Partial Specialization of Templates and plugs into the Visual Studio framework. Unfortunately, this doesn't give us the support for automake and libtool.

Chapter 3

Design and Implementation

The final design can be seen in figure 3-1. The NFS client lives inside the kernel and communicates with the automounter and the SFS client using UDP sockets. The automounter communicates with the nfsmounter to mount file systems. The nfsmounter mounts file systems in the NFS client and in the Cygwin mount table.

3.1 Cygwin

Cygwin was chosen as the Programming Environment. Cygwin is a library and a set of tools that contain most of the system calls that UNIX supplies. Cygwin was created by Cygnus Solutions, is Open Source, and is now supported by RedHat. Cygwin allows the programmer to pretend he is developing on a UNIX system. Many

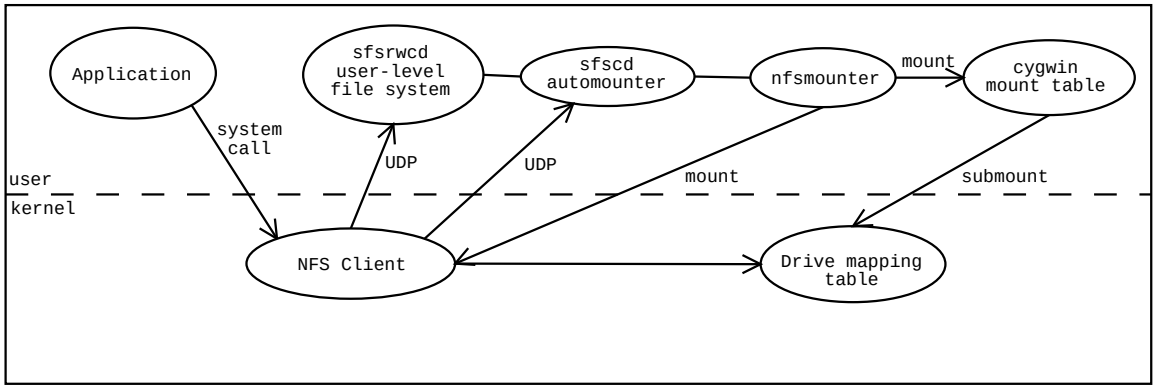


Figure 3-1: SFS for Windows Design

programs have been ported to Windows using Cygwin including Apache, OpenSSH and XFree86.

Cygwin is a good choice because it supports automake, autoconf, and libtool. The code can be kept the same because Cygwin supports many of the UNIX system calls that SFS uses. All changes made to the original SFS code are marked with `#ifdef __CYGWIN32__` so builds on UNIX will ignore them.

After placing ifdef's around missing functions and ignoring the nfsmounter, which is system dependent, SFS compiled and passed 18 of the 22 tests in *make check*. In fact Cygwin was so good at porting that all we had to do was find the problems and fix them. Most of the design was done as a reaction to what did not work in Cygwin. We followed the motto of "If it ain't broke, don't fix it."

3.1.1 Fork and Handle Problems

Windows does not have a function like UNIX's fork, which creates a new process that is an exact copy of the calling process. Windows only knows how to start a new process but not how to start a duplicate one. Cygwin works around this problem by using a complicated algorithm to start a new process and then copy all the data and duplicate all the handles. It was discovered in the development process for SFS that when a process forks it leaves multiple copies of the handles in the child process. This was problematic because processes would not get an EOF when the handle was closed because there were duplicates that remained opened. The solution is to store the values of the handles in another variable and force a close after the fork.

Fortunately, a solution to this problem was simple thanks to the asynchronous library. The most common use of fork in the library is before a new process is spawned. To spawn a new process the SFS library calls fork followed by exec to spawn the new process. The spawn function will also call a callback procedure before the exec call but after the fork. This callback procedure can close the extra handles.

We are currently working with the Cygwin developers to find the real cause of the problem in order to find an actual solution.

3.1.2 Namespace Problems

SFS uses the colon in a self-certifying pathname to separate the hostname from the hostid. It was unfortunate to discover that Windows does not allow a colon as a character in a filename because it is used to describe drive letters (E:) or special devices (com1:). In Windows the % character was used instead of the colon. The symlink fixes describe in Section 3.2.3 fix all the symlinks so that they use % instead of colons. The SFS team is currently discussing a new format for self-certifying pathnames.

3.1.3 Select Problems

The select call in Windows is incomplete. It selects only on sockets and not on any other type of handle. Cygwin implemented a select call for file descriptors by spawning a new thread for each different type of descriptor and having the main thread wait for the other threads to return. This solution is computationally expensive. While a call to select in Windows that should return immediately takes 16 milliseconds the same call to Cygwin select takes 940 milliseconds to run. As a result, SFS is unbearably slow. We solved this problem by calling the Windows select on socket parameters instead. This solution lowered the above-mentioned runtime to 20 milliseconds and made SFS twice as fast.

3.2 NFS Client

We chose to use a commercial NFS client to solve the NFS access problem. This choice means, unfortunately, that to run SFS the user will have to purchase an NFS client. A dependable NFS client is less likely to halt the machine and will also give us the fastest solution to the problem. We do hope that SFS will not be tied for long to the commercial NFS client and that someone will develop a free NFS client for Windows.

We chose Hummingbird's NFS Maestro Client [5] because it supports NFS Ver-

sion 3, supports mounting same servers using different ports, and MIT Information Systems has technical contacts with them.

3.2.1 Mounting NFS Filesystems

In UNIX, programs mount NFS file systems through the mount system call by passing this function the port number and IP address of the NFS server, along with the NFS file handle of the directory being mounted. Users generally mount NFS file systems using a command line program mount, which converts a hostname and directory into the info needed by the system and performs the mount system call. It obtains the port from a portmapper running on the NFS server and the file handle from a mountd daemon also running on the server. SFS calls the mount system call directly to mount loopback NFS servers. This direct mount interface is not available in Windows.

To mount directory 'D' on server 'S' to a drive letter, the Windows NFS client first contacts the mountd server running on S, asking to get the file handle for D. After the client gets the file handle it calls the portmapper running on S asking for the port for NFS. The NFS client now has all the hostname, the file handle, and the port, and can do the kernel magic necessary. Since we do not have the source code to the NFS client we don't know what this magic is so we must work around it.

The nfsmounter, part of the SFS client, needed to be modified to support mounting using a directory name, instead of a port and file handle. Since these two values are obtained from the portmapper and the mountd daemon respectively, these two daemons must be running in the system. The mountd daemon runs inside the nfsmounter so that it can easily obtain the file handles of the directories that the nfsmounter wants to mount. The portmapper however runs as a separate process and is not part of the SFS client.

The different SFS clients run on different ports and the portmapper must be updated to reflect that. Therefore before returning a file handle for a given directory the mountd daemon calls into the portmapper to change the NFS port to the correct value. (Steps 3-5 in Figure 3-2) When the NFS client calls into the portmapper it gets the correct value for the port. This process allows the NFS client to mount */sfs*

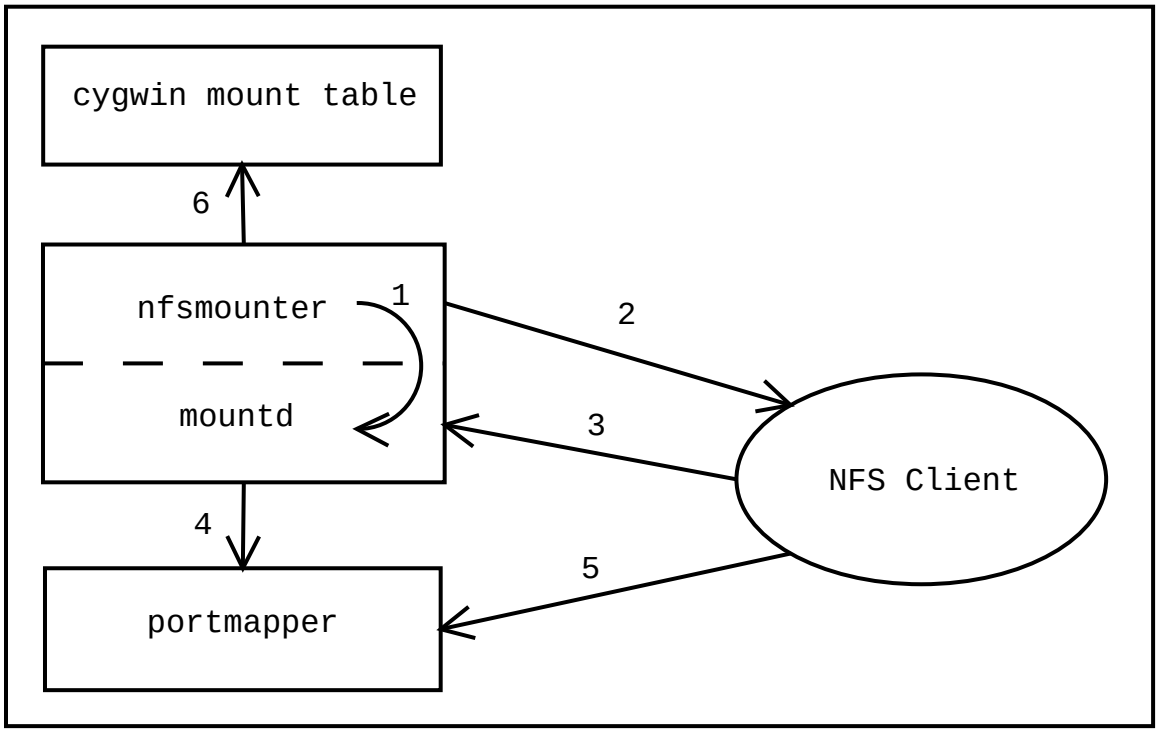


Figure 3-2: Mount Process

which is on one port, and mount */sfs/foo* which is on another port. This technique could also be useful on UNIX variants that do not have a mount system call.

3.2.2 Cygwin Submounts

Windows uses drive letters to refer to file systems. While UNIX has one root directory, '/', Windows has multiple drive letters. To solve this problem Cygwin supports the idea of submounts. The Cygwin shell allows the user to create submounts that convert drive letters into directories (i.e., map drive E: to */cdrom*). Cygwin also allows submounts on subdirectories. Therefore we can mount E: to */cdrom* and F: to */cdrom/drive2*, and when we access */cdrom/drive2* it will send us to F: and not to the directory drive2 on E:. Unfortunately Cygwin submounts exist only in the Cygwin world and outside of Cygwin, for example in Windows Explorer, going into E:*/drive2* may take you to an empty directory or give you an error if E:*/drive2* does not exist.

We modified the nfsmounter to create submounts in Cygwin to the drive letters

that correspond to the correct file system. In this way when the Cygwin user accesses */sfs* Cygwin sends him to the correct drive letter. Unfortunately outside of Cygwin it will not work this way, and the user will have to deal with drive letters.

Our modified mount process is shown in Figure 3-2. When the nfsmounter receives the `NFSMOUNTER_MOUNT` call a long process begins. First it adds the file handle and the port to a list of existing mounts (Step 1), then it tells the NFS client to map the directory to a drive letter using the "nfs link" command (Step 2), finally it calls Cygwin's mount command to map the POSIX path (i.e. */sfs*) to that drive letter (Step 6).

Much to our chagrin, it was discovered that changing the port in the portmapper was not enough to convince the NFS client to connect to another port as was proposed. Fortunately adding a parameter for the port to the "nfs link" command caused the NFS client to mount the file system to the correct port.

A simple shell script does the mounting. The script calls the NFS client mount command "nfs link", and then calls Cygwin's mount command with the result of the above. To unmount it does the opposite. It calls umount then "nfs unlink." The shell scripts can be easily modified to support other NFS clients for Windows.

3.2.3 Symbolic links

Because self-certifying pathnames can be very long and hard to remember SFS uses symbolic links. For example, when a user authenticates to a remote server using an agent the agent places a symbolic link in the user's */sfs* directory that points to the correct self-certifying pathname. This approach allows the user to follow the symlink to get to the right location and not have to remember a long self-certifying pathname. Also an SFS server can have symlinks to other SFS servers so that the users do not have to remember a long name.

Because Windows does not understand the concept of symlinks, Hummingbird's NFS client follows all symlinks when it first looks up a directory. This approach has a consequence that when you change into a directory the client gets the directory listing and follows all the symlinks until the end. Any symlinks that do not have an

end, either because they loop or their destination is not found, are removed from the listing.

Any symlinks to */sfs/foo* are completely undefined since */sfs/foo* is a Cygwin submount pointing to another drive letter, the NFS client runs outside of Cygwin and Cygwin submounts are not visible outside of Cygwin. In order to fix this problem the NFS client needs to defer the symlink resolution to Cygwin.

In Cygwin, symlinks are implemented as Windows Shortcuts. Cygwin has special handlers that detect shortcuts and make sure that they look like symlinks to the Cygwin user. Shortcuts are special Read-Only files that have a *.lnk* extension and special data naming the destination. When you try to access a file in Cygwin, Cygwin will first try to lookup the file and if that fails it will lookup the file with the *.lnk* extension. If it finds the file and its format is correct, it will do a lookup on the destination of the link. Currently Cygwin will only iterate through 10 links so that recursive links won't freeze your system.

We modified SFS to replace symlinks with shortcuts in the automounter. The */sfs* virtual directory is created using objects called afsnodes. One of these nodes is called a *delaypt* because it delays the `READLINK` call. This node was replaced by a different node that serves a binary file with special data that says where the destination is.

The automounter was changed as follows. When the automounter receives a `LOOKUP` RPC in the */sfs* directory that corresponds to a self-certifying pathname, *foo*, it creates a shortcut with *foo* plus *.lnk* that points to *foo*. The NFS client will then do a lookup on *foo* plus *.lnk* because Cygwin requests it. Once Cygwin sees that a file exists it reads the contents of the file and tries to follow it. Following it causes a `LOOKUP` RPC in the */sfs* directory pointing to *foo* and the process repeats itself. While Cygwin and the NFS client loop, the automounter actually mounts the remote file system under */sfs*. Cygwin always checks its mount table before doing a lookup and once the automounter mounts the file system and modifies the mount table, Cygwin sees the submount for *foo* and redirects the request to the correct drive letter.

Unfortunately using this method does not convey any error message to the user.

The only error message is "Too many symbolic links" which happens because the above loop was repeated more than ten times and Cygwin assumes that it's a recursive symbolic link. In UNIX you may get "Permission denied" or "Connection timed out," which is much more constructive.

The SFS client was also modified to make sure that all files that were symlinks received the *.lnk* extension and the result of `READ` RPC's on them pointed to the correct directory. This change was easy to implement because SFS has a notion of stackable NFS manipulators. For example, there is a class *nfsserv_fixup* that works around many bugs in kernel NFS clients to avoid panicking the kernel. Several NFS client implementations crash if the results of a `LOOKUP` RPC do not include the file attributes and this manipulator makes sure that no `LOOKUP` RPCs return without attributes.

We implemented a shortcut generator, as a stackable NFS manipulator, that converts all symlinks into shortcuts. In this way, Cygwin can both trigger the automounting that may be necessary, and send us to the correct drive letter. The rules for turning symlinks into shortcuts are as follows:

1. Any `LOOKUP` for a file that would return a symlink returns file not found.
2. Any `LOOKUP` for a filename that fails but ends in *.lnk* returns the file handle for the filename without the *.lnk* but with the attribute changed to be a regular file.
3. Any `READ` on a file handle that fails because the file is really a symlink returns the data for the symlink.
4. All `READDIR` calls are changed into `READDIR_PLUS` to obtain the attributes quickly.
5. `READDIR_PLUS` adds a *.lnk* extension to filenames if they are symlinks.
6. `ACCESS` and `GETATTR` return modified data for symlinks.

3.2.4 Access Rights

NFS uses user id's (UIDs) to figure out your permissions. SFS uses these UIDs to figure out who you are and what files to show you. It also uses your UID to determine the correct agent to use.

The Windows NFS client unfortunately has no access to these UIDs. To supply Windows NFS clients with UIDs an RPC daemon, PCNFSD, maps Windows usernames to UID's. A person that wants to give access to Windows users runs the PCNFSD daemon on their NFS Server. Another possible solution is to run a PCNFSD daemon locally and have it obtain your UID through other channels, like using Kerberos to do a hesiod lookup [3]. In our case both solutions are the same since our NFS server is running locally; the challenge is to obtain the correct UID.

When you first install Cygwin it creates a */etc/passwd* file from your computer's accounts, and assigns all accounts UIDs. These are the UIDs that SFS uses to give access. We developed a PCNFSD daemon that runs inside the nfsmounter and assigns you UIDs from the password file. This solution allows the NFS client to use your Cygwin UIDs.

We implemented the PCNFSD daemon and placed it inside the nfsmounter. This allows the NFS client to obtain a user's Cygwin UID. Unfortunately, the NFS client caches these UIDs and once */sfs* is mounted it has a UID associated with it that cannot change. This property causes many problems with SFS.

SFS selects the correct agent based on the UID of the NFS RPC call. It also displays different views of the */sfs* directory depending on your UID. This scheme allows SFS to put symbolic links in the directory depending on the agents that a user is running. In Windows, since the UID doesn't change, if a second user logs in to Windows he will get the same view as the first user,and since SFS can not tell that this another user, it will give him the same access permission. We have contacted Hummingbird about this problem but haven't received any response. For now, the only solution is to not allow multiple users on the machine at the same time.

3.3 File Descriptor Passing

Cygwin did not support passing file descriptors. The Cygwin code had to be modified to support this useful protocol. The challenge was to design a system that worked similarly to the UNIX method of passing file descriptors. We also wanted to add the support to Cygwin and not keep it inside SFS.

3.3.1 File Descriptors in Cygwin

Windows uses opaque handles to identify Pipes, Sockets, Files, Consoles, etc. The designers of Cygwin wanted to keep the UNIX notion of file descriptors, contiguous integers that represent Pipes, Sockets, etc. So in Cygwin when you create a file descriptor it allocates the correct type of the handle, stores it in an object, and adds the object into an array so that the file descriptor is really an index into that array. Cygwin also stores the type of handle, and other info like permissions into the object. This design allows Cygwin to use Pipe system calls on Pipes, and Socket system calls on Sockets, etc. Handles are per process values and only have meaning in a particular process; therefore, passing around the value of a handle has no productive result. To transfer handles between processes the handles must be duplicated, which we explain next.

3.3.2 Duplicating a Handle

A handle can be duplicated in Windows by using the DuplicateHandle call. This call takes a source process, a source handle, a destination process, and returns a destination handle. This destination handle is a duplicate of the source handle that is valid in the destination process. This call allows a sender, a receiver or even a third party to duplicate a handle assuming that that process has enough permission to do so.

3.3.3 Passing Handles

The Cygwin developers implemented the cygserver as a way to share memory between processes. It enables calls such as shmget and shmctl. Basically, the Cygserver is an RPC daemon that holds memory and handles for other processes. That way if the other processes exit the memory is retained. The cygserver is an optional tool in cygwin and must be run separately.

The cygserver is an RPC daemon and we can define functions and call them. We define two new functions in the cygserver for the purpose of passing file descriptors, `HOLDHANDLE` and `RELEASEHANDLE`. `HOLDHANDLE` takes a Process ID (pIn), a handle (hIn) valid in process pIn, and returns a DuplicateHandle of hIn valid in the cygserver process. `RELEASEHANDLE` takes a ProcessID (pOut), a handle valid in the cygserver process (hCyg), and returns a DuplicateHandle of hCyg valid in process pOut.

Using the cygserver we can easily duplicate the handles between two processes. The sending process first calls the cygserver `HOLDHANDLE` routine and gets a handle valid in the cygserver process. Then when it sends the message to the second process, using normal UNIX Domain Socket I/O, the first process adds a header that contains the file descriptor information and the value of the handle before the actual data. The receiving process detects the header and calls `RELEASEHANDLE` and then converts the Handle into a file descriptor. The entire process works as follows.

1. P1 calls sendmsg passing it some data and FD1.

2. Inside sendmsg, Cygwin extracts handle H1 from FD1 and calls `HOLDHANDLE` sending it H1 and P1; it receives HCyg.

3. Sendmsg sends the handle, plus the handle's info, in a header and the data to P2.

4. P2 calls recvmsg.

5. In recvmsg Cygwin receives the data, sees the extra header, and calls `RELEASEHANDLE` sending it HC and P2; it gets back H2.

6. Recvmsg then allocates a file descriptor (FD2) and binds the handle info and the Handle to that file descriptor

Since we were adding headers to messages we needed to come up with a scheme to not confuse headers with real messages. We did not want to add headers to all messages because that would slow things down. We needed a way to preserve message boundaries so that we could easily detect a header and could send multiple messages with headers without getting them confused with real data. UNIX sockets in Cygwin are really TCP sockets, which do not preserve message boundaries.

We considered using out of band data to pass the headers. Out of band data is used as a second transmission channel or as urgent data. Out of band data can also be sent inline in which case the programmer can detect the presence of it using the `SIOCATMARK` ioctl. Only one byte can be sent in out of band data so we could not send entire headers this way.

Another option was to send a byte of out of band data inline as a tag stating that a header was coming. Unfortunately the `SIOCATMARK` ioctl in Windows is only able to tell you if there is out of band data pending. It will not tell you if you are going to read it, and, if you send two messages out of band you cannot reliably detect the first one.

It was discovered that if you send normal out of band data the system somehow preserves the message boundaries. Therefore, before we send a packet containing a header we send an out of band byte and all our boundaries are preserved. Because we had no access to Windows source code we could not investigate why this behavior occurs.

This patch was submitted to Cygwin and is being looked at closely for admission.

Chapter 4

Evaluation

We evaluated two main aspects of this research: The number of changes that had to be made and the performance of the system.

4.1 Changes

We modified 25 files of the original SFS code and created 8 new files. Of the 25 files 4 are changes to Makefiles to include libraries not normally included by Cygwin like *resolv* and *rpclib*. The grand total of lines changed and created is 1328. We think that maintaining this small number of code changes will be relatively simple. The list of modified files and new files can be found in Figures 4.1 and 4.2 respectively.

4.2 Performance

We compared the performance of SFS on Windows to accessing files using Hummingbird's NFS client, Microsoft's CIFS client, and IBM's AFS client. Since SFS runs on top of Hummingbird's client we will see what the overhead of the loopback NFS is. We ran the LFS large and small file micro-benchmarks [10] to see the performance of all these network file systems.

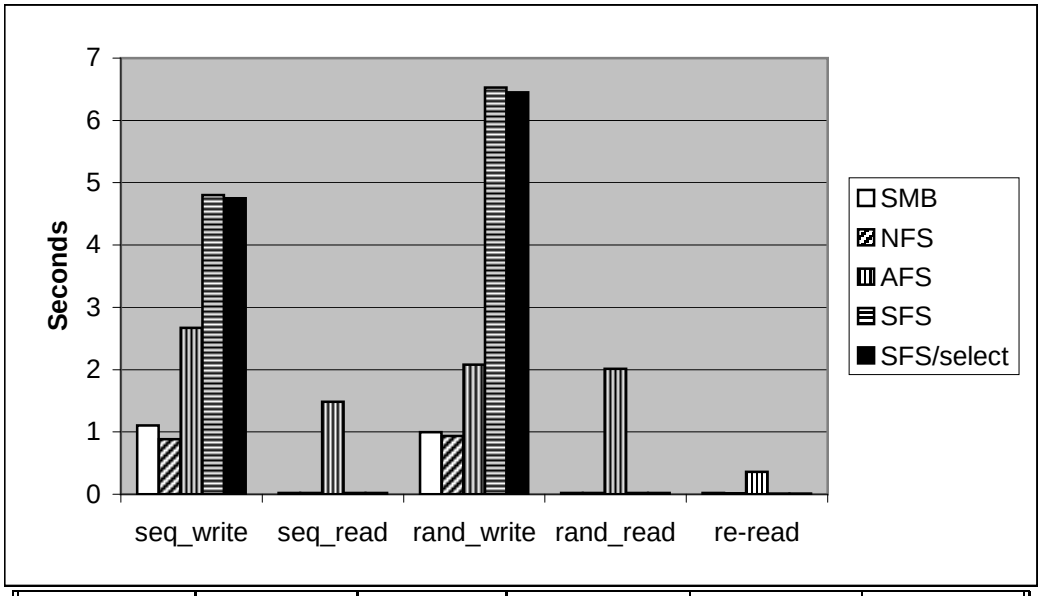
The SFS client ran on an 866MHz Pentium3 with 256K Cache, 512Mb of RAM, with a 10Mb 3Com Ethernet Controller. The SFS server, the NFS server and the

Filename	Number of Lines Changed
agent/Makefile.am	14
arpc/Makefile.am	3
arpc/axprt_unix.C	18
async/Makefile.am	7
async/aio.C	2
async/async.h	3
async/core.C	76
async/sigio.C	2
crypt/getkbdnoise.C	3
crypt/rndseed.C	4
nfsmounter/Makefile.am	6
nfsmounter/mpfsnode.C	11
nfsmounter/nfsmnt.h	9
nfsmounter/nfsmounter.C	9
sfscd/afs.C	5
sfscd/afsroot.C	26
sfscd/afsroot.h	9
sfscd/sfscd.C	3
sfsmisc/afsdir.C	17
sfsmisc/afsnode.C	22
sfsmisc/afsnode.h	12
sfsmisc/nfsserv.C	274
sfsmisc/nfsserv.h	14
sfsmisc/sfspath.C	4
sfsrwcd/sfsrwcd.C	3
Total	556

Table 4.1: List of Modified Files

Filename	Number of Lines
nfsmounter/domount-cygwin.C	213
nfsmounter/mountsfs	6
nfsmounter/umountsfs	6
async/select-winsock.C	40
nfsmounter/pcnfsdsrv.C	235
portmap/Makefile	63
portmap/srv.C	209
Total	772

Table 4.2: List of New Files



	seq_write	seq_read	$rand\_write$	$rand\_read$	$re-read$
NFS	0.883	0.02	0.935	0.02	0.014
AFS	2.672	1.4842	2.081	2.011	0.357
CIFS	1.102	0.018	0.9972	0.018	0.02
SFS	4.8034	0.02	6.5298	0.018	0.012
SFS/select	4.7543	0.02	6.4258	0.018	0.012

Figure 4-1: Run time in seconds for Large File Microbenchmark (8Mb file using 8KB I/O Size

CIFS server all ran on a 233Mhz Pentium2 with 256K Cache and 128Mb of RAM. AFS was tested against MIT's AFS servers. The NFS client was configured to use NFS version 3 over UDP. All the servers are located in the same subnet to make transit time as equal as possible. Each test was run five times to make sure the data was accurate.

4.2.1 Latency

We first tested the latency of NFS versus SFS because we were having problems with speed. We tested how fast we could send a `SETATTR` RPC since this involved no disk activity. We observed that by adding the fix suggested in Section 3.1.3 the calls per second went from 11 to 21. This result is pretty low compared to 333 calls per second in NFS.

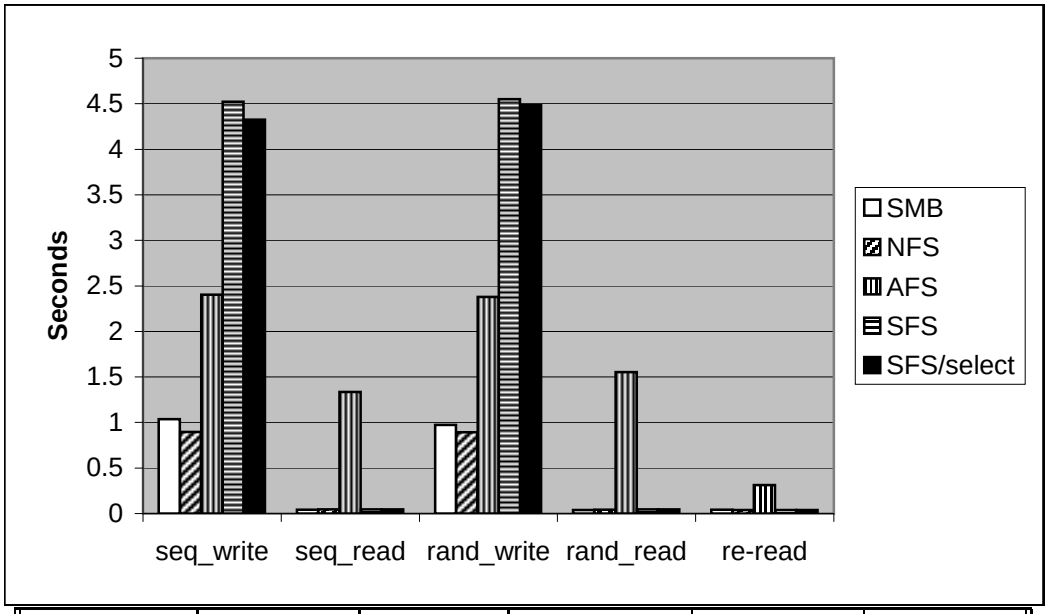
4.2.2 Large File Microbenchmark

The large file microbenchmark sequentially writes a large file, reads it sequentially, writes it randomly, reads it randomly, and then re-reads it sequentially. Data is flushed to the disk after each 8 Kb write. We used a 8MB file with I/O sizes of 8KB and 256Kb. The results are shown in Table 4-1 and 4-2.

As can be observed, with an 8K buffer size SFS is about 6 times as slow in writing as NFS and half as slow as AFS. The reading for both SFS and NFS are affected by the fact that the NFS client has cached the result in the NT Cache Manager, which AFS has not. The same results were found using a 256K buffer. Strangely, the select fix discussed in Section 3.1.3 did not make these number improve.

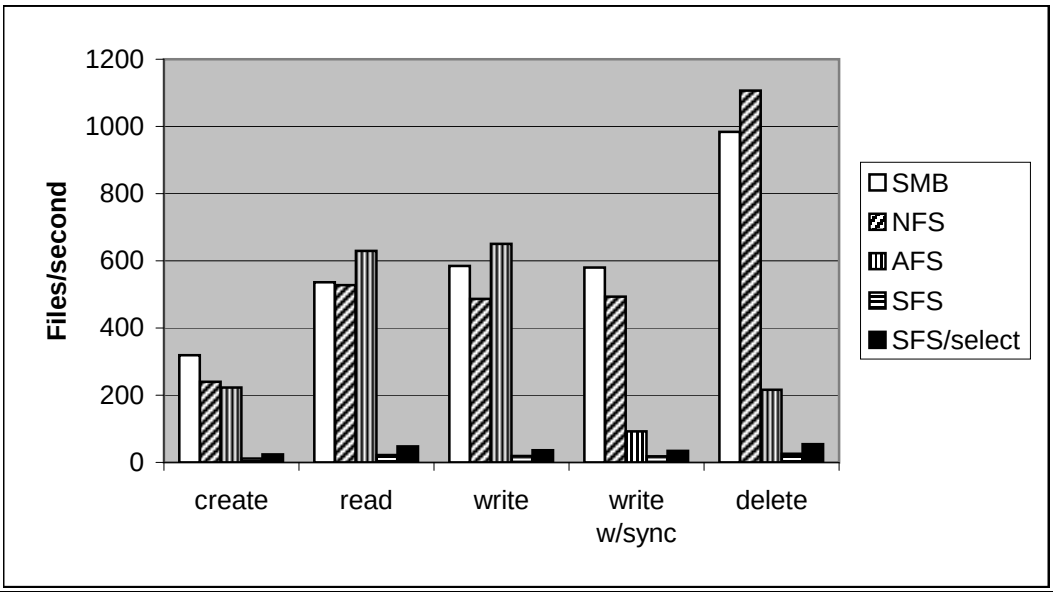
4.2.3 Small File Microbenchmark

The small file microbenchmark creates 1000 1Kb files across 10 directories. It then reads the files, re-writes them, re-writes them flushing changes to the disk (write w/sync), and deletes them. Because of the large amount of files, the time really represents the lookups on each file.



	seq_write	seq_read	$rand\_write$	$rand\_read$	$re-read$
NFS	0.897	0.044	0.893	0.042	0.04
AFS	2.402	1.336	2.3776	1.554	0.312
CIFS	1.038	0.042	0.9972	0.018	0.02
SFS	4.523	0.044	4.551	0.044	0.04
SFS/select	4.327	0.044	4.485	0.044	0.04

Figure 4-2: Run time in seconds for Large File Microbenchmark (8Mb file using 256KB I/O Size



	create	read	write	write w/sync	delete
NFS	239.81496	527.21222	487.00168	492.95294	1106.8717
AFS	222.58972	629.47972	650.79284	92.7624	215.9645
CIFS	318.83076	535.96386	584.31022	579.79568	984.0685
SFS	11.18796	21.66708	18.63374	18.17918	25.76406
SFS/select	23.6961	47.24716	35.71712	34.7592	54.42674

Figure 4-3: Performance in files per second for Small File Microbenchmark 1000 1KB files in 10 directories

Our modifications to select mentioned in Section 3.1.3 doubled the performance of SFS in the small file benchmark but it still remained ten times as slow as other systems. It is however interesting to see that the writes with sync and the delete tests in AFS are extremely slow compared to the others. This may be due to the file locking mechanism inside AFS or to bad interaction with the Cache Manager.

The performance of SFS is very bad compared to the other systems. We believe that this slowness may be caused by other slow Cygwin calls. We hope that future work on the project will turn up the source of these problems in SFS.

Chapter 5

Conclusion

5.1 Future Work

It is unfortunate that we used the commercial NFS client for NFS functionality. The commercial client costs money and we do not have access to the source code to correct errors and many work-arounds had to be devised. In the future it would be great to develop an NFS client for Windows. The OSR FSDK toolkit could be used for such an endeavor. CMU wrote Coda for Windows using it so perhaps it is a good choice.

The problem of passing file descriptors can also be solved without using the cygserver. In fact, the Cygwin developers would much prefer a solution that does not involve the cygserver because they consider the cygserver an optional tool.

Another issue with the current file descriptor passing system is that if the receiver dies before picking up a file descriptor from the cygserver the handle will be stored in the cygserver forever. Some method of garbage collection could be developed to overcome this problem.

5.2 Conclusion

While the performance of SFS left something to be desired we believe that this research will prove useful. Not only did we reach our goal of creating an SFS client for Windows but we also did it in a manner that will make maintenance and future work

simple. Also, the lessons learned in this research can be applied to other projects.

To our knowledge there have never been any automounters in Windows. Our system, while very NFS client dependent, may prove to be useful for such a project. Also our handling of symlinks could prove valuable in future projects. Our research into passing file descriptors in Cygwin sparked many discussions in the Cygwin mailing lists and this useful feature will finally exist in the Cygwin library so that other developers may use it.

We believe that the Windows port of SFS can be improved and that it can be made comparable in speed to NFS.

Bibliography

[1] Danilo Almeida. Framework for Implementing File Systems in Windows NT. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1998.

[2] Bo Branten. Project: ntifs.h. `http://www.acc.umu.se/~bosse/`.

[3] Thomas Bushnell and Karl Ramm. Anatomy of an Athena Workstation. In *System Administration Conference*, pages 175–180, December 1998.

[4] Paul Leach and Dilip Naik. A Common Internet File System (CIFS/1.0) Protocol. *Internet-Draft IETF*, December 1997.

[5] Hummingbird Ltd. Hummingbird NFS Maestro Client. `http://www.hummingbird.com/products/nc/nfs/index.html`.

[6] David Mazières. *Self-certifying file system.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 2000.

[7] David Mazières. A Toolkit for User-level File Systems. In *USENIX Technical Conference*, 2001.

[8] MIT. Project WinAthena. `http://web.mit.edu/pismere/`.

[9] Matthew Scott Rimer. The secure file system under Windows NT. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1999.

[10] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *13th ACM Symposium on Operating Systems Principles*, pages 1–15, October 1991.

[11] Magnus Ahltorp Royal, Love Hörnquist-Åstrand, and Assar Westerlund. Porting the Arla file system to Windows NT.

[12] Open Source. OpenAFS. `http://www.openafs.org/`.

[13] Open Source. Samba. `http://www.samba.org/`.

[14] Open System Resources Inc. File System Development Kit. `http://www.osr.com/fsdk/`.

[15] Carnegie Mellon University. Coda. `http://www.coda.cs.cmu.edu/`.