32

A User Level Modular File-system Infrastructure

by

Yonah Schmeidler

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

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 1998 [illegible]

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

Certified by...
M. Frans Kaashoek
Thesis Supervisor

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

Eng.

A User Level Modular File-system Infrastructure

by

Yonah Schmeidler

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

Abstract

This thesis discusses the design and implementation of a framework for constructing user-level modular filesystems. This framework facilitates incremental extension of previous systems as well as the code reuse and sharing between systems. Unlike previous works, this work focuses primarily on file servers rather clients, although it can be used for both. The framework described here provides a convenient, flexible, and portable environment for filesystem design. A simple port of an existing file server to this framework performed only about 2% worse than the original server, with a minimal overhead cost for additional modules.

Thesis Supervisor: M. Frans Kaashoek

Acknowledgments

I would like to acknowledge Frans Kaashoek, my advisor, who guided me through writing a thesis, and my colleagues at Arepa, Inc., for providing input and encouragement throughout the design and implementation of this system.

Contents

1 Introduction **8**

2 Design Choices **10**

2.1 Kernel or User Space 11

2.2 One Process or Many 12

2.3 Mailboxes or Procedure Calls 13

2.4 A Single Protocol or Many 14

2.5 Dynamic or Static Linking 15

2.6 Module Organization 15

3 Implementation **17**

3.1 Overview of the Framework 18

3.2 Procedure Numbers 20

3.3 Framework Functions 21

4 Example **23**

4.1 Initialization 23

4.2 Dispatch 23

4.3 Cleanup 24

5 Performance **25**

5.1 Expectations 25

5.2 Results 26

6 Related Work **28**

6.1 Network Stacks 28

6.2 Stackable Filing 28

7 Future Work **30**

7.1 Generalized Module Organization 30

7.2 Static Linking 30

7.3 Moving from C to C++ 31

7.4 Client Modules 31

8 Conclusion **33**

A Framework Source Code **34**

B Framework Include File **42**

C Example Statistics Module Source Code **44**

List of Figures

1-1 Reusing transport and storage modules for multiple caching schemes . 9

2-1 Module reuse made possible by a common inter-module message protocol 15

List of Tables

3.1 Module entry points 18
3.2 Functions provided to modules by the framework 21

5.1 Seconds taken to build NFS modules 26
5.2 Seconds taken to copy a large file 26

Chapter 1

Introduction

With the rise in popularity of the World Wide Web, increasing numbers of people are using the Internet. Along with mainstream use comes an expectation of high availability and reliability, even in the face of greater demand. Many filesystems are showing performance problems under this load of increased usage and higher expectations. This, in turn, is leading to more work on improving the performance of existing filesystems, as well as designing new high performance and high reliability filesystems.

Most modern filesystem implementations consist of several integrated components. For example, a typical client or server includes components which interface to the network, interface to the underlying filesystem, provide caching services, and implement access restrictions and security. While it is always possible to separate these components conceptually, in many cases the implementations are too tightly bound for boundaries to be drawn between them.

This thesis discusses several possible designs for, and presents one implementation of, a filesystem infrastructure in which the various components are explicitly separated into modules, with limited and well defined intercommunication. Explicitly separating modules also forces the use of abstractions and requires the interfaces between modules to be planned out and documented. By separating storage (physical media), transport (network protocols), and intermediate file handling (caching, security, etc.) components, the various pieces can be independently created, updated,

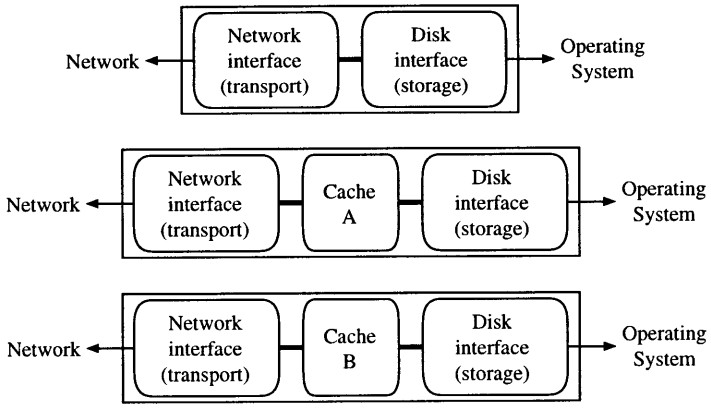

Figure 1-1: Evaluating multiple caching schemes for a filesystem by reusing transport and storage modules

and tested.

This is especially useful for filesystem development. The utility and performance of a new idea can be evaluated with respect to previous implementations; a new module can be added to an existing implementation without changing other modules. For example, Figure 1-1 shows how the same transport and storage modules can be used to evaluate two different caching schemes. Since the only part which changes is the caching module, differences in performances can be directly related to the differences in the caching.

Chapter 2

Design Choices

There are a number of issues that must be considered in filesystem development. One of the most important is speed. Much of the work being done today is aimed at improving, or improving on, the performance of existing systems. For filesystems, there are actually two performance metrics that must be considered, namely latency, or how long it takes to handle a single request, and throughput, or how many requests can be handled in a given amount of time.

Two other important issues in developing filesystems are the ease of debugging and the ability to do profiling. Being able to trace program execution and inspect data at various points while the program is running can speed up the debugging process considerably. Similarly, being able to use existing profiling tools to generate call graphs showing the amount of time spent in various routines aids in identifying and reducing performance bottlenecks.

A final issue to consider is portability. While it may be educational to implement a filesystem on a single platform, any system that wants wider acceptance must be able to run on several platforms. Porting a system between platforms is much easier if it is designed using standard operating system interfaces and widely available tools.

Given these design criteria, we will now examine some specific choices that must be made in designing a modular filesystem infrastructure.

2.1 Kernel or User Space

Today, almost all computer systems make a distinction between programs running as part of the system and user programs. The kernel and other privileged programs generally have complete access to the hardware, while user programs have more limited access in exchange for the convenience of standard, relatively portable, interfaces.

On most systems, filesystem clients must at least hook into the kernel in order to be seen by other processes through the standard filesystem interfaces. For example, under Windows 95, filesystem drivers must be implemented completely within the kernel. However, some operating systems, such as Linux, do provide for user-level filesystems. In addition, it is possible to create "loop-back" filesystems, where a user-level filesystem is used as a server by a traditional in-kernel client. However, these loop-back filesystems are often limited by the functions and performance available through the loop-back protocol being used. In addition, servers written as part of the kernel may take advantage of low overhead and low-level access to the underlying filesystems that they are serving. Because of this it is usually easier to implement clients completely within the kernel, although user-level helper applications are sometimes used to provide parallelism [12].

Despite the speed and efficiency of filesystems implemented in the kernel, there are important benefits seen by user-level implementations. One of the most important is portability. While there are many competing implementations of filesystem and network interfaces at the kernel level, almost all systems today provide the user-level interface specified in the POSIX standard or something very similar [14]. This means that well-written user-level programs can be ported to many systems with little additional effort. For filesystem clients, only the lowest level interface need really be part of, and be ported between, various kernels.

In addition, user-level programs are generally easier to debug and profile than their kernel level counterparts. Even if they are available, kernel debuggers are generally less convenient and cannot be used without affecting the operation of the entire system. Similarly, it is easier to modify and restart a user-level process after it crashes

than to do the same for the entire kernel. In addition, there are tools for profiling the execution of user-level programs that are available for almost all platforms, but not generally available for kernels [1].

While kernel-level filesystem implementations win on performance, a user-level approach was chosen for this infrastructure because of its greater utility for filesystem research and development.

2.2 One Process or Many

One obvious way to approach the design of a user-level modular system is to have each module be a separate process, with data being passed as messages between modules. This structure has the advantage that each module would be independent, with its own storage space, and could block while waiting for incoming messages.

However, there are many drawbacks to this design. Most important, from a performance viewpoint, is the high latency associated with context switching between modules. Once a module has processed a message and sent it on to the next module, the system must perform a context switch to the next module before processing of the message can continue. While work is being done to improve this, even a short message typically requires 100 us to be transfered on a modern processor [7]. If there are many requests for a module to handle each time it runs, this cost might be amortized over many messages, leading to reasonable throughput. Unfortunately, most clients send few requests, if any, in parallel, so the full latency is likely to be seen, especially in small scale applications requiring high performance.

Another problem with this design is that data must either be copied from one process' address space to another's or stored in shared memory. While using shared memory eliminates unnecessary copying, it means that structures containing pointers cannot be sent from one module to another. Although some systems map shared memory to the same virtual address in each process, this cannot be guaranteed on all systems. Different pointers to a block shared memory may be returned when it is mapped into different processes' address spaces.

Building the framework such that all of the modules run as a single process allows for lower latency and easier data passing. However, it also reduces the separation between modules, so more care must be taken to ensure that they do not interfere with each other.

2.3 Mailboxes or Procedure Calls

The next question to address is how messages are passed. This can either be done as a set of mailboxes (first-in first-out queues) or through procedure calls.

Using mailboxes makes it easier for modules to handle requests asynchronously. A module need not wait for the next module to finish handling a given message in order to start processing another message. In addition, individual modules can choose whether or not to use multiple threads to handle blocking requests in parallel. Mailboxes, unlike procedure calls, may also be used if modules are split into separate processes.

In practice, however, most modules are concerned with associating messages being passed up the stack to be processed with their responses coming down the stack. While this is possible with mailboxes, a procedure call interface is simpler and easier to use. When the filesystem framework was originally developed, mailboxes were used. However, most modules implemented a blocking procedure call interface on top of the mailboxes, so the implementation was changed.

A procedure call interface provides functionality that is both more convenient and more useful for filesystem modules; it corresponds closely to the remote procedure call (RPC) interface used for many network protocols such as NFS [10, 11]. In exchange for these features, requests may only be handled in parallel through the use of multiple threads. This, in turn, means that modules must take care to provide thread-safe reentrant interfaces.

2.4 A Single Protocol or Many

Another issue is the format of the messages to be passed. The two extremes are for all modules to support a single common set of messages, or for each pair of modules to have their own set of messages. In practice, the former is often used but is overly restrictive; it leads to numerous revisions of the protocol and extra work to provide for backward and future compatibility [9]. On the other hand, the other end of the spectrum is less than ideal because it is overly general, making it difficult to create modules that can be transparently inserted between others.

The filesystem framework presented here requires the modularity provided by the use of a single message protocol; the use of standard protocols facilitates insertion, removal, and reordering of modules. However, in order to provide for extension and easy implementation of various filesystems, the framework does not define the protocol to be used. A set of modules implementing a filesystem may use whatever protocol is most convenient. In this way, a module that is designed for one filesystem can be used without any modifications either in another part of the same filesystem or in a different filesystem entirely that uses the same protocol. Most protocols for use between modules are expected to be based on standard protocols used on one end or the other of the module stack, such as the on-the-wire protocol or the system disk interface [13]. In addition, modules are expected to pass unrecognized messages through, so that intermediate modules can be used even if they handle only an older subset of the current protocol.

Figure 2-1 shows how network interface and cache modules written for a modular client can be reused in a server implementation, and then modules from both reused to create a proxy server. All of the modules in this scenario can communicate using a protocol based on what is sent over the network, since that includes all of the data of interest to the modules.

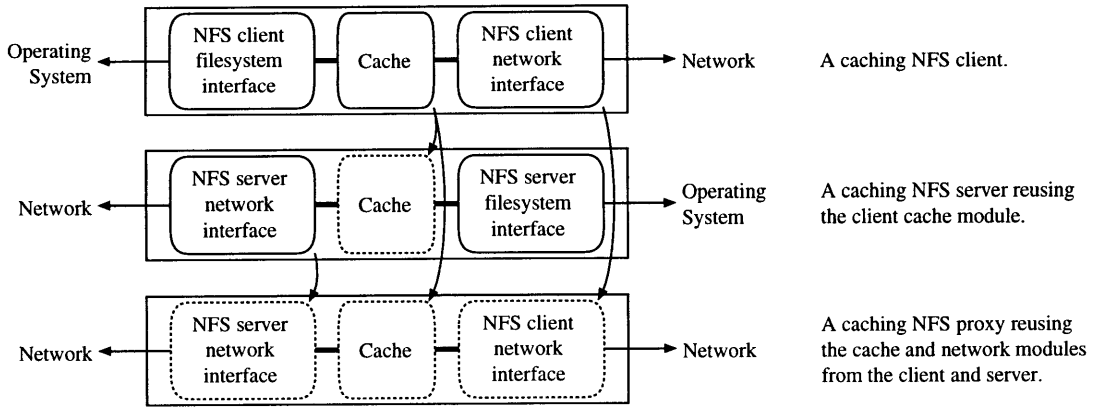

Figure 2-1: Module reuse made possible by a common inter-module message protocol

2.5 Dynamic or Static Linking

One approach to loading modules into the framework is to compile all of the modules to be used and then link them together with the framework to create a single executable binary. This approach has the drawbacks that all global symbols, both data and functions, share a single symbol space, creating a greater potential for collisions; that it is harder to change which modules should be used between invocations of the framework; and that it is harder to distribute modules individually other than as source code.

An alternative approach is to make each module dynamically loadable, and then to tell the framework which modules to load at runtime. While this does not have the problems listed above, it reduces the ease of profiling and debugging of modules. On most systems, the profiling and debugging tools do not handle dynamically loaded modules gracefully.

2.6 Module Organization

A final issue is the topology of module connections. The simplest option is only to provide for a single stack or chain of modules, each connected to at most one above and one below. This option is easy to implement, and the connections can be described by a simple ordered list of modules.

However, there are many applications for which this organization is not sufficient. For example, consider a server that allows requests to be made using either of two network protocols. The option described above would require this to be implemented using either a single network module that understood both protocols, or a module for each protocol, with one module passing the messages from the other through without processing.

A better option would be to have a module for each protocol, each of which talked to a single module that combined the requests, perhaps converting them into a single protocol, and sent them along to the next module for processing. However, this would require a more general description for the connections between modules, and a more general connection mechanism than single up and down pointers.

While the current implementation of the framework only supports the use of a single stack of modules, a provision for more general organizations is being designed.

Chapter 3

Implementation

There are two parts to the implementation of this system: the specifications and guidelines for creating modules, and the executable framework which binds and executes them. Both of these parts have been designed to work on Solaris, Linux, and Windows 95. However, modules do need to be recompiled between platforms, and not all modules will compile and run on all platforms because of differences in the operating systems.

The program responsible for managing the modules is referred to as the *framework*. The C source code for this program is listed in Appendix A. This program can be built to use either POSIX or WIN32 application interfaces depending on whether `unix` or `_WIN32` is defined. Encapsulating these differences in the framework, rather than in each module, provides greater portability.

When it is run, the framework takes a list of modules, loads them, and finds the entry points for each one. This is done using dynamically loaded libraries (DLL's) on Windows 95 and shared object files on Solaris and Linux. The list of modules may be changed with each invocation of the framework, but does not change dynamically (while the framework is running).

This chapter presents an overview of the execution of the framework and presents general information for implementing modules. It begins with a description of the initialization of modules, followed by general execution guidelines for modules, and then a summary of how the framework may exit. The chapter concludes with a

`module_init(int m, struct framework *f, dispatch_t up, dispatch_t down)`
: receives from the framework the module identifier (`m`), dispatch table for services provided by the framework (`f`; see Section 3.3), and pointers to the dispatch functions of the adjacent modules in the stack (`up` and `down`)

`module_dispatch(unsigned proc, unsigned length, void *data)`
: the dispatch function called by adjacent modules

`module_exit(void)`
: called immediately before the framework exits

Table 3.1: Module entry points

discussion of procedure numbers for messages between modules and a description of the functions provided to modules by the framework.

3.1 Overview of the Framework

Each module consists of a single dynamically loadable object that must export a small number of entry points, shown in Table 3.1. These entry points may call other functions within the object, in the dispatch table provided to `module_init` by the framework, or in other libraries with which it was linked.

Once the modules are loaded, the framework calls the `module_init` procedure in each module with the appropriate arguments. Any module may "fail" its initialization by returning a non-zero value, in which case the framework calls the `module_exit` procedure in each module which initialized successfully and then exits.

In order to prevent messages from being sent to modules before they have been initialized, no module may send a message until it has received one, with the first message being sent by the framework only after all of the modules have been successfully initialized. This "run" message gets passed through the stack as would any other message. However, modules must not rely on this being the first message they receive. For example, the bottom module might start a thread that accepts network connections when it receives the run message. This thread might then send a message up to say that it had received a connection before the original thread has had

a chance to continue the progress of the run message. Since a module may begin to send messages once it receives any message, it may continue processing this one, including passing it up the stack, even though it has not yet received the run message.

Once the run message has been sent, it is up to the modules to initiate further activity. This is done by some module or modules starting threads, either during the initialization phase or in response to the run message, which then initiate messages in response to outside events, such as client requests and server callbacks.

It is expected that many modules will be written so that they use multiple threads in order to process requests in parallel. Because of this, modules should be prepared to deal with concurrency control issues between messages in separate threads, unless all the modules being used are designed to work with only one message in progress at a time, which reduces both complexity and performance.

There are also guidelines on the allocation of data to be passed in messages. The module which starts a message is considered responsible for allocating and freeing the storage space for data associated with that message. The module which processes the message and generates a reply is responsible for allocating space for the data in the reply. However, since the reply only goes back down the stack of modules, the module which allocated the storage does not know when it's done being used. Therefore, the module which started the message is also responsible for freeing any space used by the reply.

The modules' execution ends when the framework exits, which may happen in three ways. The least desirable, as with any other program, is for some error to occur which halts the program immediately. For example, the computer may crash or a module may attempt to perform an illegal operation. A module calling the standard exit function is also included in this category, as it bypasses the preferred exit procedure described below. In any of these cases, the framework does not attempt to do any further processing before it exits.

The framework may also be asked to exit by a module calling the exit function passed into the initialization procedure. In this case, the framework calls the exit procedure for each module before exiting. While it would be nice to have a restriction

that a module may not send messages after its exit function has been called, such a restriction would impose too great a burden on module implementations because other threads are not killed off before the program exits. Instead, modules are simply allowed to stop processing messages once their exit functions have been called or they have received the stop message described below.

However, it is often desirable to have the system shut down more gracefully. If a module calls the framework's stop function or the framework receives a user interrupt signal, it sends a stop message to the bottom module before carrying out the exit procedure described above. This message is similar to the run message after initialization; it is sent through the stack like other messages, and it is a signal to the modules that the framework is about to exit and that they need not process any further messages.

3.2 Procedure Numbers

A module passes a procedure number, a length, and a pointer to data when making a procedure call to another module. A 32-bit unsigned integer is use for the procedure number. This number is split into several pieces for convenience.

The top 16 bits of the procedure number are used as a protocol number. Protocol zero is reserved for use by the framework. If this framework were in widespread use, a registry of protocol numbers might be appropriate, as with SUN's RPC [10].

The lower 16 bits of the procedure number are used to designate the procedure number within the protocol. The highest of these bits is used to distinguish the direction of messages; procedures 8000 through 8FFF (hex) should be used for calls going down the stack. Aside from this, there is no need for assignment of these numbers to be consistent across protocols.

For protocol zero, procedure zero is defined to be the run message and procedure one is defined to be the stop message. In the future, other procedures within this protocol may be used for other messages of interest to all modules. For example, messages could be sent as notification of various signals from the system under

`log(int severity, char *format, ...)`	log a message
`stop()`	shut down and exit cleanly
`exit()`	shut down and exit quickly
`malloc(unsigned length)`	allocate memory
`realloc(void *buf, unsigned length)`	allocate memory
`free(void *buf)`	free allocated memory
`start_thread(void(*fn)(void*), void *data)`	start a new thread
`mutex_create(mutex_t *)`	create a mutex
`mutex_lock(mutex_t *)`	lock a mutex
`mutex_unlock(mutex_t *)`	unlock a mutex
`mutex_destroy(mutex_t *)`	destroy a mutex
`semaphore_create(semaphore_t *)`	create a semaphore
`semaphore_wait(semaphore_t *, mutex_t *)`	wait for a semaphore to be signaled, atomically releasing and later reacquiring a mutex
`semaphore_signal(semaphore_t *)`	signal a semaphore
`semaphore_destroy(semaphore_t *)`	destroy a semaphore

Table 3.2: Functions provided to modules by the framework; each function takes the module identifier as an additional argument

UNIX, such as the hangup signal, which is often used to ask programs to reread their configurations.

To allow for future expansion, modules should pass any unrecognized message through to the next module in the appropriate direction. This also allows non-adjacent modules in the stack to communicate through a second protocol in addition to the one being used by all of the modules.

3.3 Framework Functions

The framework provides a dispatch table containing a number of useful functions to each module's initialization routine. These functions are summarized in Table 3.2. The usage of the `stop` and `exit` is described above.

Another function provided to modules by the framework is a logging function. This function writes some text to a standard place, including the name of the module sending it and a severity level. By handling this through the framework, messages

from all modules go to a single location, such as the console or the syslog facility.

On Windows 95, memory allocated using the standard `malloc` functions in one dynamically loaded library (i.e., a module) can only be freed by a call to `free` from the same module. This is a poor situation given the memory allocation guidelines above, where different modules are responsible for allocating and deallocating memory for replies. To resolve this, the framework provides `malloc` and `free` functions to the modules which call the base C functions from within the framework rather than within a module, and can therefore be used across modules.

The remaining functions are provided in order to facilitate the processing of requests in parallel and are based on the POSIX threads interface [14]. Providing a common interface to these functions provides portability to platforms which do not support the POSIX interface as well as a single point from which all of them can be easily traced.

Chapter 4

Example

To demonstrate how modules work in this framework, a simple performance monitor was built to work with the Linux user-level NFS server, which was ported to use the framework. Every 250 calls to a given NFS procedure, this performance-monitoring module logs the average and peak time required to perform the procedure. The source code for this module is listed in Appendix C.

4.1 Initialization

The `module_init` procedure here is straightforward. First, it checks to make sure that it is somewhere in the middle of the stack of modules by making sure that it is given dispatch functions for modules on both sides. After that, it copies the arguments to global variables for future use and returns.

4.2 Dispatch

In this example, the module passes all calls through in the appropriate direction, timing and logging any NFS calls. The dispatch routine, `module_dispatch`, first checks to see if the procedure is a downcall, as described in Section 3.2. If it is, then the routine returns the result of the downcall. Otherwise, the routine checks to see whether the procedure is an NFS call. In this example, procedures numbered 10200

through 10211 hexadecimal are NFS calls. If it is an NFS call, the upcall is timed and the statistics for that procedure number are updated. Otherwise, the upcall is done without further processing.

The implementation of this module is single-threaded. No concurrency control or synchronization is needed because the modules being used to implement the NFS protocol are all specifically designed to be single-threaded for messages. Modules are allowed to do background processing in other threads, as long as there is only one message in progress at a time. While this may result in a less efficient server, the traditional server code on which this implementation was based was designed to be single-threaded, so keeping this limitation made porting it to the new framework much easier.

4.3 Cleanup

The `module_dispatch` procedure simply prints out remaining statistics. Since summaries are printed every 250 calls to each procedure, it is likely that there is some information which has not been shown because procedures were not called a multiple of 250 times.

Chapter 5

Performance

One of the design goals of the filesystem framework, as mentioned in Chapter 2, was performance. Two metrics for evaluating this are the time and space overhead seen by implementing a filesystem using this framework, as compared to a traditional server.

5.1 Expectations

Given the design decisions discussed in Chapter 2, the overall performance of the system was expected to be close to the performance of a traditional user-level server. In terms of space, the only changes are minor overheads, such as the framework itself, the symbol tables in the modules, and per-module storage for framework information (pointers to adjacent dispatch procedures, etc.). These are all fairly small, so the net result is not significantly larger when using the filesystem framework. In addition, these are one time costs; they affect the total space needed, but not the amount of space required to process each message.

In terms of time, there is both a startup cost and an overhead on each message to be processed. The startup cost is simple: the framework must load the modules and find their entry points. This is similar to the cost of having linked against a small number of additional shared libraries, and is quite small. All symbols in modules are resolved when they are loaded, so there is no startup cost associated with the first time a procedure is called from a module.

	min	avg	max
traditional	25.1	25.4	25.8
framework	25.6	25.9	26.3

Table 5.1: Seconds taken to build NFS modules

server	time
traditional	227
framework	232
framework with null modules	235

Table 5.2: Seconds taken to copy a large file

The overhead for each message should also be small. One change is that messages must now be passed between the modules through pointers to dispatch routines, rather than through a single direct dispatch table. Another change is that some procedures, such as `malloc` and `free`, must now be called through the framework's service table. Once again, these differences are not expected to cause a significant change in the performance of servers using the framework.

5.2 Results

The Linux user-space NFS server was modified to use the framework, and the performance of the new version was compared to that of the original server. Two tests were used to examine the performance of the filesystem framework. Two machines running Linux were used for the tests, one as client and one as server, connected by a 10 megabit ethernet.

In the first test, a copy of the source code for the framework and the NFS modules was compiled on each server. This test was repeated 7 times for each configuration; a summary of the results of this test is shown in Table 5.1. Overall, the performance of the server using the framework was 2-3% worse than that of the traditional server.

For the second test, a 100 megabyte file was copied onto a local disk from each server. Again, as shown in Table 5.2, the performance of the framework was about

2% worse. This test was repeated 3 times for each configuration; all times were within 0.6% of the average time which is shown.

To show the impact of adding modules to a server, a "null" module was created. This module does nothing other than pass on messages in the correct direction. The second test was then run on a server running the NFS modules used above, with ten copies of the null module inserted between the network module and the rest of the server. The resulting server, also shown in Table 5.2, performed about 1% slower than the server without the null modules.

Overall, the performance of the server using the modular framework was comparable to the performance of the traditional server. The space overhead is constant and minimal when compared with the size of the entire server. The time overhead is slightly more of a problem. However, it should be kept in mind that the server used in these tests was originally written to run on its own; code optimized for the framework might be as fast, or faster than, the traditional server.

Chapter 6

Related Work

Code reuse and abstraction through the use of modules is not a new concept. However, most previous works along these lines have had different design goals and have made different decisions, either explicitly or implicitly, about the design issues presented earlier.

6.1 Network Stacks

The x-kernel presents a similar approach for modular network protocols [4]. It provides a framework for implementing and composing protocols from smaller building blocks. This allows many of the same advantages as the modular filesystem framework described here, but is implemented as its own kernel. Since it is designed for network protocol stacks, most operations are based on packet header manipulation. Similarly, because the focus of most messages is different in this context, the framework focuses on optimizing a different set of operations.

6.2 Stackable Filing

Previous works have concentrated on providing the ability to implement new filesystems in terms of old ones by stacking and the use of modules [2, 5, 9]. However, these works have focused primarily on the client side of filesystems.

Because of this focus, most of these systems have continued to work within the kernel. As discussed above, this allows a simpler framework and higher performance, as well as easier migration from previous systems. In exchange, the modules are harder to develop and port to other operating systems and applications, such as file servers.

Similarly, these systems use fixed interfaces for inter-module communications. While this helps guarantee that modules can be arbitrarily reordered, in many cases this too limiting. For example, very few of the standard abstractions have a mechanism for allowing messages to be passed from the server side of the stack to the client side.

On the other hand, there are some features provided by these systems which are not available with the framework described here. For example, the stackable filesystem work provides for more generalized module organization (see Section 2.6, above) and includes a cache coherence architecture for use between modules [2, 3]. Similarly, the Spring system provides a more advanced interface between modules, using strongly-typed objects and interface inheritance [5].

Chapter 7

Future Work

There are a number of changes and improvements to the system presented above that are currently being worked on.

7.1 Generalized Module Organization

As mentioned in Section 2.6, there are some situations in which simply connecting modules as a stack is inconvenient. A module should be able to receive requests from multiple downstream modules and pass those requests on to any of a number of upstream modules.

This generalizes to connecting the modules as a directed acyclic graph. Work is being done to implement this now, although there are several issues still to be resolved. For example, it is unclear what a module should do if it receives an unrecognized message; should it get passed on to all of its upstream neighbors, or only to one?

7.2 Static Linking

Another modification being considered is to statically link the modules to be used with the framework, rather than having them be dynamically loaded when the framework is run. This makes it harder to change which modules are being used, but in exchange only one executable needs to be kept track of to run the system. As mentioned in

Section 2.5, one benefit of this approach is that many existing debugging and profiling tools do not work well on dynamically loaded objects.

Another important benefit of static linking is that it resolves the problem mentioned earlier of allocating and freeing memory in different modules. Since the modules are linked into a single binary, the standard `malloc` and `free` functions may be used as usual on Windows 95, and need not be exported by the framework.

7.3 Moving from C to C++

One issue seen with the change from dynamic to static linking of modules is that some modules use the same global variable or function names. When modules are dynamically loaded, symbols are not resolved between modules; modules can only interact with each other through the dispatch function pointers provided by the framework. However, when modules are statically linked together, the linker, unaware of module distinctions, attempts to resolve references across module boundaries. Similarly, it is no longer possible for each module to use a function named `module_init` for its initialization procedure, as the framework must be able to call each one separately.

The solution currently being tried in conjunction with static linking is to encapsulate each module into a C++ class. This allows modules to keep "global" data as class members without concern for matching names in other modules, while allowing a module to be split across several source files. In addition, the framework can continue to access modules through a standard set of functions which each module inherits from a base module class.

7.4 Client Modules

One application of this filesystem framework that has not been explored fully is as a client. The modules currently implemented only provide for filesystem servers and proxies. At one point, a client module was begun for Windows 95, but it turns out that filesystem drivers for Windows 95 must be implemented either completely within

the kernel or using a loopback mechanism as described in Section 2.1. It would be interesting to examine the performance of a user-level filesystem client on a system with support for it, such as Linux.

Chapter 8

Conclusion

This thesis presents a different approach to modular filesystem design than previous works. It attempts to provide greater access to standard development tools as well as increased portability by resembling a traditional user-level program in many respects. While this approach does not yet show the same performance as other works, the difference seems minimal and may be overcome in future tests. The framework presented here is the subject of ongoing development, such as the ideas presented in Chapter 7.

Appendix A

Framework Source Code

```
#include "fw.h"
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#if defined(unix)
#include <dlfcn.h>
#include <signal.h>
#elif defined(_WIN32)
#include <process.h>
#endif
#ifdef SYSLOG
#include <syslog.h>
#endif

static struct modinfo {
    char *name;
#if defined(unix)
    void *handle;
#elif defined(_WIN32)
    HINSTANCE handle;
#endif
    int (*init)(int, struct framework *, dispatch_t, dispatch_t);
    dispatch_t dispatch;
    void (*exit)(void);
} *info;

#if defined(unix)
static pthread_attr_t detach_attr;
static pthread_cond_t end_sem;
static pthread_mutex_t end_mut;
#elif defined(_WIN32)
```

```
static HANDLE end_sem, end_mut;
#endif

static int end_flag, log_debug, nummods;

void m_log (int module, int severity, char *format, ...)
{                                                                    40
#ifdef SYSLOG
    char buf[2048];
    static int sev_prio[4] = { LOG_DEBUG, LOG_NOTICE, LOG_WARNING, LOG_ERR };
#endif
    static char *sev_desc[4] = { "debug", "info", "warning", "error" };
    va_list ap;

    if (!log_debug && severity == M_LOG_DEBUG)
        return;
    va_start(ap, format);                                            50
#ifdef SYSLOG
    if (severity < 0 || severity > 3)
        sprintf(buf, "%s [%d]:  ", info[module].name, severity);
    else
        sprintf(buf, "%s [%s]:  ", info[module].name, sev_desc[severity]);
    vsprintf(buf+strlen(buf), format, ap);
    if (severity < 0 || severity > 3)
        syslog(LOG_INFO, "%s", buf);
    else
        syslog(sev_prio[severity], "%s", buf);                       60
#else
    if (severity < 0 || severity > 3)
        fprintf(stderr, "%s [%d]:  ", info[module].name, severity);
    else
        fprintf(stderr, "%s [%s]:  ", info[module].name, sev_desc[severity]);
    vfprintf(stderr, format, ap);
    fprintf(stderr, "\n");
#endif
}
                                                                     70
void m_stop (int module)
{
    int i;

#if defined(unix)
    pthread_mutex_lock(&end_mut);
#elif defined(_WIN32)
    WaitForSingleObject(end_mut, INFINITE);
#endif
    if (!end_flag)                                                   80
    {
        end_flag = 1;
        info[0].dispatch(1, 0, 0);
        for (i = 0; i < nummods; i++)
            info[i].exit();
        exit(0);
    }
```

```
    for (;;)
#if defined(unix)
        pthread_cond_wait(&end_sem, &end_mut);
#elif defined(_WIN32)
        WaitForSingleObject(end_sem, INFINITE);
#endif
}

void m_exit (int module)
{
    int i;

#if defined(unix)
    pthread_mutex_lock(&end_mut);
#elif defined(_WIN32)
    WaitForSingleObject(end_mut, INFINITE);
#endif
    if (!end_flag)
    {
        end_flag = 1;
        for (i = 0; i < nummods; i++)
            info[i].exit();
        exit(0);
    }
    for (;;)
#if defined(unix)
        pthread_cond_wait(&end_sem, &end_mut);
#elif defined(_WIN32)
        WaitForSingleObject(end_sem, INFINITE);
#endif
}

void *m_malloc (int module, unsigned len)
{
    return malloc(len);
}

void *m_realloc (int module, void *buf, unsigned len)
{
    return realloc(buf, len);
}

void m_free (int module, void *buf)
{
    free(buf);
}

int m_start_thread (int module, void (modthread *fn)(void *), void *data)
{
#if defined(unix)
    pthread_t thread;
    return pthread_create(&thread, &detach_attr, (void*(*)(void*))fn, data);
#elif defined(_WIN32)
    return _beginthread(fn, 0, data);
```

```
#endif
}

int m_mutex_create (int module, mutex_t *mutex)
{
#if defined(unix)
    return pthread_mutex_init(mutex, 0);
#elif defined(_WIN32)
    return ((*mutex = CreateMutex(0, 0, 0)) == 0);
#endif
}

int m_mutex_lock (int module, mutex_t *mutex)
{
#if defined(unix)
    return pthread_mutex_lock(mutex);
#elif defined(_WIN32)
    return (WaitForSingleObject(*mutex, INFINITE) != WAIT_OBJECT_0);
#endif
}

int m_mutex_unlock (int module, mutex_t *mutex)
{
#if defined(unix)
    return pthread_mutex_unlock(mutex);
#elif defined(_WIN32)
    return !ReleaseMutex(*mutex);
#endif
}

int m_mutex_destroy (int module, mutex_t *mutex)
{
#if defined(unix)
    return pthread_mutex_destroy(mutex);
#elif defined(_WIN32)
    return !CloseHandle(*mutex);
#endif
}

int m_semaphore_create (int module, semaphore_t *semaphore)
{
#if defined(unix)
    return pthread_cond_init(semaphore, 0);
#elif defined(_WIN32)
    return ((*semaphore = CreateSemaphore(0, 0, 256, 0)) == 0);
#endif
}

int m_semaphore_wait (int module, semaphore_t *semaphore, mutex_t *mutex)
{
#if defined(unix)
    return pthread_cond_wait(semaphore, mutex);
#elif defined(_WIN32)
#if WE_WERE_USING_NT
```

```
    return (SignalObjectAndWait(*mutex,*semaphore,INFINITE,0)!=WAIT_OBJECT_0);
#else
    ReleaseMutex(*mutex);
    return (WaitForSingleObject(*semaphore, INFINITE) != WAIT_OBJECT_0);
#endif                                                                          200
#endif
}

int m_semaphore_signal (int module, semaphore_t *semaphore)
{
#if defined(unix)
    return pthread_cond_signal(semaphore);
#elif defined(_WIN32)
    return !ReleaseSemaphore(*semaphore, 1, 0);
#endif                                                                          210
}

int m_semaphore_destroy (int module, semaphore_t *semaphore)
{
#if defined(unix)
    return pthread_cond_destroy(semaphore);
#elif defined(_WIN32)
    return !CloseHandle(*semaphore);
#endif
}                                                                               220

struct framework myframework = {
    m_log,
    m_stop,
    m_exit,
    m_malloc,
    m_realloc,
    m_free,
    m_start_thread,
    m_mutex_create,                                                             230
    m_mutex_lock,
    m_mutex_unlock,
    m_mutex_destroy,
    m_semaphore_create,
    m_semaphore_wait,
    m_semaphore_signal,
    m_semaphore_destroy
};

#ifdef _WIN32                                                                   240
static char *GetErrorText (const int err)
{
    static char buf[2048];
    if (!(FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, 0, err, 0, buf, 2048, 0)))
        sprintf(buf, "%d (FormatMessage error %d)", GetLastError());
    return buf;
}
#endif
```

```
#ifdef unix
void sighandler ()
{
    if (!end_flag)
    {
        end_flag = 2;
        pthread_cond_signal(&end_sem);
    }
}

void * modthread sigfn (void *p)
{
    sigset_t set;
    struct sigaction act;
    int signum;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    act.sa_handler = sighandler;
    act.sa_mask = set;
    act.sa_flags = SA_RESTART;
    sigaction(SIGINT, &act, 0);
    sigwait(&set, &signum);
    if (!end_flag)
    {
        end_flag = 2;
        pthread_cond_signal(&end_sem);
    }
    return 0;
}
#endif

int main (int argc, char **argv)
{
    pthread_t thread;
    int i;
    char *p, **modargs;

    nummods = argc - 1;
    if (argc > 1 && !strcmp(argv[1], "-d"))
    {
        log_debug = 1;
        modargs = argv + 2;
        nummods--;
    }
    else
        modargs = argv + 1;
    if (nummods < 2)
    {
        fprintf(stderr, "usage:  %s [-d] module [module...]  module\n", argv[0]);
        exit(1);
    }
    if (!(info = malloc(nummods * sizeof(*info))))
    {
```

```
        perror(argv[0]);
        exit(1);
    }
    for (i = 0; i < nummods; i++)
    {
        if (!(info[i].name = strrchr(modargs[i], '/')))
            info[i].name = modargs[i];
        else
            info[i].name++;
#if defined(unix)
        if (!(info[i].handle = dlopen(modargs[i], RTLD_NOW)) ||
            !(info[i].init = dlsym(info[i].handle, "module_init")) ||
            !(info[i].dispatch = dlsym(info[i].handle, "module_dispatch")) ||
            !(info[i].exit = dlsym(info[i].handle, "module_exit")))
        {
            fprintf(stderr, "%s:  %s:  %s\n", argv[0], info[i].name, dlerror());
            exit(1);
        }
#elif defined(_WIN32)
        if (!(info[i].handle = LoadLibrary(modargs[i])) ||
            !(info[i].init = (void*)GetProcAddress(info[i].handle, "module_init")) ||
            !(info[i].dispatch = (void*)GetProcAddress(info[i].handle, "module_dispatch")) ||
            !(info[i].exit = (void*)GetProcAddress(info[i].handle, "module_exit")))
        {
            fprintf(stderr, "%s:  %s:  %s\n", argv[0], info[i].name,
                GetErrorText(GetLastError()));
            exit(1);
        }
#endif
        if ((p = strrchr(info[i].name, '.')) != 0)
            *p = 0;
    }

#if defined(unix)
    pthread_cond_init(&end_sem, 0);
    pthread_mutex_init(&end_mut, 0);
    pthread_attr_init(&detach_attr);
    pthread_attr_setdetachstate(&detach_attr, PTHREAD_CREATE_DETACHED);
    end_flag = 0;
#elif defined(_WIN32)
    end_sem = CreateSemaphore(0, 0, 16, 0);
    end_mut = CreateMutex(0, 0, 0);
#endif

    if (info[0].init(0, &myframework, info[1].dispatch, 0))
    {
        fprintf(stderr, "%s:  %s:  initialization failed\n", argv[0], info[0].name);
        exit(1);
    }
    for (i = 1; i < nummods-1; i++)
    {
        if (info[i].init(i, &myframework, info[i+1].dispatch, info[i-1].dispatch))
        {
            fprintf(stderr, "%s:  %s:  initialization failed\n", argv[0], info[i].name);
```

```
            while (i)
            {
                i--;
                info[i].exit();
            }
            exit(1);
        }
    }
    if (info[i].init(i, &myframework, 0, info[i-1].dispatch))
    {
        fprintf(stderr, "%s:  %s:  initialization failed\n", argv[0], info[i].name);
        while (i)
        {
            i--;
            info[i].exit();
        }
        exit(1);
    }

#ifdef SYSLOG
    openlog("fw", 0, LOG_LOCAL2);
#endif

    info[0].dispatch(0,0,0);

#if defined(unix)
    pthread_create(&thread, &detach_attr, (void*(*)(void*))sigfn, 0);
    pthread_mutex_lock(&end_mut);
    while (end_flag != 2)
        pthread_cond_wait(&end_sem, &end_mut);
#elif defined(_WIN32)
    while (end_flag != 2)
        WaitForSingleObject(end_sem, INFINITE);
#endif
    for (i = 0; i < nummods; i++)
        info[i].exit();
    exit(0);
}
```

Appendix B

Framework Include File

```
#ifdef unix
#include <pthread.h>
typedef pthread_mutex_t mutex_t;
typedef pthread_cond_t semaphore_t;
#define modfnspec
#define modthread
#endif
#ifdef _WIN32
#include <windows.h>
typedef HANDLE mutex_t;
typedef HANDLE semaphore_t;
#define modfnspec _declspec(dllexport)
#define modthread _cdecl
#endif

typedef int (*dispatch_t)(unsigned, unsigned, void *);

struct framework {
    void (*log) (int module, int severity, char *format, ...);
    void (*stop) (int module);
    void (*exit) (int module);
    void *(*malloc) (int module, unsigned len);
    void *(*realloc) (int module, void *buf, unsigned len);
    void (*free) (int module, void *buf);
    int (*start_thread) (int module, void(modthread *fn)(void*), void *data);
    int (*mutex_create) (int module, mutex_t *mutex);
    int (*mutex_lock) (int module, mutex_t *mutex);
    int (*mutex_unlock) (int module, mutex_t *mutex);
    int (*mutex_destroy) (int module, mutex_t *mutex);
    int (*semaphore_create) (int module, semaphore_t *semaphore);
    int (*semaphore_wait) (int module, semaphore_t *semaphore, mutex_t *mutex);
    int (*semaphore_signal) (int module, semaphore_t *semaphore);
    int (*semaphore_destroy) (int module, semaphore_t *semaphore);
```

```
};

#define M_LOG_DEBUG 0
#define M_LOG_INFO 1
#define M_LOG_WARNING 2
#define M_LOG_ERROR 3
                                                                    40
modfnspec int module_init (int, struct framework *, dispatch_t, dispatch_t);
modfnspec int module_dispatch (unsigned, unsigned, void *);
modfnspec void module_exit (void);
```

Appendix C

Example Statistics Module Source Code

```
#include "fw.h"

static int module;
static struct framework *fns;
static dispatch_t upfn, downfn;

#define HITS_PER_LOG 250

static struct {
    unsigned hits, total, peak;
} the_stats[18];

int module_init (int m, struct framework *f, dispatch_t up, dispatch_t down)
{
    if (!up || !down)
    {
        fns->log(module, M_LOG_ERROR, "must be middle of stack");
        return 1;
    }
    module = m;
    fns = f;
    upfn = up;
    downfn = down;
    return 0;
}

int module_dispatch (unsigned proc, unsigned len, void *buf)
{
    int i, ret;
    unsigned delta;
```

```
    struct timeval begin, end;
    struct nfs_query *nq;

    if (proc & 0x8000)
        return downfn(proc, len, buf);
    if ((proc & 0xffffff00) == 0x10200 && (i = proc & 0xff) < 18)
    {
        gettimeofday(&begin, 0);
        ret = upfn(proc, len, buf);
        gettimeofday(&end, 0);                                                    40
        delta = ((end.tv_sec - begin.tv_sec) * 1000000) + end.tv_usec - begin.tv_usec;
        the_stats[i].hits++;
        the_stats[i].total += delta;
        if (the_stats[i].peak < delta)
            the_stats[i].peak = delta;
        if (the_stats[i].hits == HITS_PER_LOG)
        {
            fns->log(module, M_LOG_INFO, "proc %d:  avg %u, peak %u us",
                    i, the_stats[i].total / HITS_PER_LOG, the_stats[i].peak);
            the_stats[i].hits = the_stats[i].total = the_stats[i].peak = 0;       50
        }
        return ret;
    }
    return upfn(proc, len, buf);
}

void module_exit (void)
{
    int i;
    for (i = 0; i < 18; i++)                                                      60
    {
        if (the_stats[i].hits)
            fns->log(module, M_LOG_INFO,
                    "proc %d:  avg %u, peak %u us for %u hits",
                    i, the_stats[i].total / the_stats[i].hits, the_stats[i].peak, the_stats[i].hits);
    }
}
```

Bibliography

[1] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A Call Graph Execution Profilers," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, SIGPLAN Notices, vol. 17, no. 6, p. 120-126, June 1982.

[2] John Heidemann, "Stackable Design of File Systems," Technical Report CSD-950032, University of California, Los Angeles, September, 1995.

[3] John Heidemann and Gerald Popek, "Performance of Cache Coherence in Stackable Filing," *Proceedings of the 15th ACM Symposium on Operating System Principles*, p. 127-143, Dec 1995.

[4] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, 17(1):64-76, Jan. 1991.

[5] Yousef Khalidi and Michael Nelson, "Extensible File Systems in Spring," *Proceedings of the 14th ACM Symposium on Operating System Principles*, p. 1-14, Dec 1993.

[6] Ted Kim and Gerald Popek, "Frigate: An Object-Oriented File System for Ordinary Users," *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, June 1997.

[7] Jochen Liedtke, "Improving IPC by Kernel Design," *Proceedings of the 14th ACM Symposium on Operating System Principles*, Dec 1993.

[8] Dennis M. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Tech. Journal*, vol. 63, no. 8, October 1984.

[9] David S. H. Rosenthal, "Evolving the Vnode Interface," *Proceedings of 1990 Summer USENIX Conference*, p. 102-117, June 1990.

[10] Sun Microsystems, "RPC: Remote Procedure Call Protocol Specification Version 2," RFC-1057, June 1988.

[11] Sun Microsystems, "NFS: Network File System Protocol Specification," RFC-1094, March 1989.

[12] Amin M. Vahdat, Paul C. Eastham, and Thomas E. Anderson, "WebFS: A Global Cache Coherent File System," Technical Draft, December 1996.

[13] Brent Welch, "A Comparison of three Distributed File System Architectures: Vnode, Sprite, and Plan 9," *USENIX Computing Systems Journal*, Summer 1994.

[14] ISO/IEC 9945-1:1996(E), ANSI/IEEE Std 1003.1, 1996 Edition: *Information technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language].*