Implementation of an Active File System

Steven Hart, James Nugent, and Arman Rouf

May 12, 2000

Abstract

The Active File System (ACFS) is a concept of file system design that allows an existing file system to be extended easily and in a secure and efficient manner. This paper describes the design and implementation of an Active File System. We first describe our concept of an Active File System and our goals for such a system. We include a detailed discussion of security and efficiency. We then describe our implementation of an active file system and how it relates to our goals. We conclude with some measurements of our implementation's performance and how it could be extended to a production system.

1 Introduction

A general file system, such as the UNIX file system, provides the basic operations for managing files considered necessary by the common user. However, there may be times when a user wishes to extend the functionality of the basic file system to tailor it to their own needs. In these situations, an active file system would be useful.

An active file system is an addition or extension to a pre-existing file system, which allows its users to customize the operation of the file system to suit their needs. This is possible by intercepting system calls and allowing module code to be run either before or after the actual system call.

While execution of user code on system calls is the primary goal of an active file system, there are a number of secondary goals which need to be met to make the system practical for everyday use. These goals are performance and security.

If the implementation of the active file system becomes a burden to the system and causes it to noticeably slow down, even in cases when module code is not being called, most users will not find the system useful-especially since the file system is something most users rely heavily on. To prevent any dramatic drop-off in performace, ideally the active file system should not impose any overhead in cases where a call to module code is not being made. In cases where module code is being run, efficiency is more dependent on the code itself and what actions it takes. Still, the overhead in calling the user code and to start executing it should be minimal. These are some of the performance issues that must be accounted for in any implementation of an active file system, if it is to be practical to use.

The other secondary goal that needs to be met by the system is security. Without a certain minimum level of security, the system cannot be safely used in a multi-user environment. Generally, every user of a system is not trusted, and so there must be some way of distinguishing between these untrusted users and trusted users. This would allow only certain authorized users to add or modify user code in the system. Also, the system needs to be protected from malicious or incorrectly running user code. Someone should not be able to bring the entire active file system down by creating and executing bad code. At the very least, a practical implementation of an active file system should meet these minimum security requirements and not impede greater security measures from being added.

The paper is organized as follows. In the next section, we will dicuss in more depth the goals of an active file system. In section 3, we will present our implementation of an active file system. In section 4, we will provide some numbers to demonstrate the performance of our implementation. In section 5, we will disuss how we attempted to reach our stated goals, and whether we were able to meet them. Finally, in section 6, we will present areas for future work that could improve the usefulness of an active file system.

2 Purpose and Goals

2.1 General Description

The general purpose of the active file system (ACFS) is to allow code to be interposed on file system events. The ACFS module code, or modules may modify any value passed to a system call. It may also modify the values passed back from the system call to a user. Thus a ACFS module may operate either before or after the system call. This requires a way to do three tasks which are intercept or dispatch a system call, check for module code associated with a <file,event> pair in some form of database , and run the code. Also, some form of communication is required to transfer data from the dispatch code to the module and back. This is our concept of an active file system and the capabilities it has and does not have. Later we will describe how we implemented one solution to this problem.

The goal of the system is to meet our definition of an active file system in a way that is secure and efficient. Note that we do not try to make this process correct. While we can make the system code correct, we cannot guarantee the correctness of module code.

Efficiency has two goals. When there is no module for a particular <file,event> pair then performance should indistinguishable from a normal system call. When there is a module for a <file,event> pair, then performance should be close to system call overhead + communication overhead + module execution. That is to say that the process of searching for an association and starting the module should be not much more than the time to run the module and communicate the necessary information to the module and back to the client.

The dispatcher has access to the following information, the file name(needed to lookup in the database for a module association) the type of call, the arguments to the call. Assuming the dispatch mechanism finds a module for that <file,event> the following information is sent to the module: the call arguments, the file name, and the uid and gid of the user who invoked the module.

Security should be similar to the standard Unix system. That is to say a module cannot be used to gain super-user access or to gain access to/corrupt other users data. We assume that if a user accesses a file with a module, there is no guarantee about the effect of the module on the file. I.e. we do not prevent modules from changing files they are associated with, but we try to prevent adverse effects beyond that. While this obviously does not provide a very strong guarantee, it is sufficient from an implementation perspective, since the system does not restrict a modules effects within its realm of functionality. (A module can affect any of the system call arguments or return values in an arbitrary way.)

An additional goal of security is to prevent malicious or malfunctioning code. This is done by preventing modules from having access to resources they don't need. We also try to make the parts of the system robust, so they can survive corruption or they can be restarted easily. A related goal is to prevent access to system resource so rest of the system cannot be corrupted. Some ways to do this would be to check and see if module threads were still running occasionally and take action if they were not. It is difficult to decide what to do in this case, as there is no reasons to believe that module operations are idempotent. A greater problem is the fact that a module may run forever. This could be solved by having a module thread or process run for a certain amount of time, and kill it beyond that. There are two problems with this approach, one is that half done work could cause data corruption. Another is that this value is both system and call dependant, so it would be difficult to set. Despite these problems, this is still probably the best approach. From this point of view, and active file system would work best with a network file system, where there is a server backup in case of problems.

2.2 Security

While security was not the focus of this project, it merits some discussion. We presume two things. One is that modules are placed into the system by the super-user. This means the modules have known(benign) behavior. Two, we assume any user can add any of these pre-installed modules two any event of any of their files. Generally, only the owner of a file or the super-user can assign a module to an event for a file.

One question that comes to mind is, who do the modules run as? There are three answers. One is root. This is the one we used, and as long as modules are installed by root and have know behavior, this is safe. Two is the invoker of the code. That is to say the person who is using the file that has a ACFS module associated with an event for it. The third is the owner of the file that has an ACFS module associated with it's event.

Running as the invoker limits the functionality of modules, because they can only keep state in the invokers space. Obviously this is a security hole, so modules that do this must be approved by root. This is a hole in that a module can do something to the user who invokes it, who may not even know it is there. One use of this case is to translate a file on open, this translated file could then stay in the users space, and would not need to be translated every time. This option is good for static data that is unique to every user of the file.

Another option is to run as the one who installed the module(owner of the file that has it). This means the module cannot effect the person who invoked it. Technically they can't even communicate directly. This give the module the some useful abilities. For instance it could create a log file of accesses, that you might not want the invokers to read. This option is good for keeping global data that relates to every user of the file.

Running as root has the advantage that it can perform both functions and do anything it wants, however there must be some way to ensure that modules are not having unknown side effects.

2.3 Advanced Capabilities

There are several advanced features that we did not implement, but are straightforward(although not simple to implement) extensions of the concept. One is the ability for modules to run as various users as discussed in the security section. This specification would be part of the module set up, and not changeable when users add modules to their files. Modules that run as either root, or the invoker would need to be added to the database of available modules by root, so they could be verified. However modules that run as the one who installed it could be written and used by anyone. This limits the capability of these modules to a degree, but the ability to have user written modules or at least non-professionally written/verified modules is an advantage.

There should be permissions associated with modules, i.e. users cannot link modules to files they do not have ownership of. Also, they cannot affect the modules of files they use without permission. This last point depends on what the ACFS is designed for. Unless there is a need to implement security type functions with the system, a system that allows users to selectively ignore modules may be better for performance reasons.

3 Implementation

Our active file system implementation involves three major components: the stub library (the stub), the server, and libraries that contain code that can be executed on the arguments of the system calls (module code).

This section begins with an overview of our implementation of the active file system. Following the overview, each of the three ACFS components is described in greater detail.

3.1 Overview

The stub library intercepts file system-related system calls (currently read, write, open, and close calls are intercepted). Each call can be intercepted at two points: prior to the execution of the system call and after the call executes. We use the terms pre and post to indicate where the call is being intercepted. For example, we term a call to the server prior to the execution of the open system call a *pre-open* call.

The absolute path of the file, an integer representing the type of call (a function code), and the arguments to the call are passed to the server. For any file, whether or not it has any active functions, a *pre-open* call is made to the server. The server assigns a connection identifier (CID) to a *pre-open* call whose file has at least one associated active function. The CID is not used by the server itself, but is used by module code to associate future calls with a particular file descriptor owned by a particular process. The CID is returned to the stub library. The stub associates the CID with the file descriptor returned by the system when the file is opened. The stub passes the CID with every call made on that file descriptor. The CID to file descriptor association is discarded only after the file has been closed.

Module code is packaged as dynamically-loaded libraries. Functions in a library are constructed according to a convention that allows the server to resolve the function name given only the library name and the function code.

3.2 Stub Library

The stub library serves as the dispatch point, intercepting system calls and sending data to the server and sending results back to the application or system library.

This is the execution path of a *pre-open* call. The call comes into the library, the library sends the filename and arguments to the server. The server performs executes any module code and then sends the return values back to the stub. It also sends a connection id, and some info known as a mask. Then the information is sent to the glibc version of open. Then the stub library consults the mask. If there is a post open call for this file, it sends data to the server and sends the results back to the application. However if the mask indicates there is no call, the server is not contacted. This means there is almost no overhead in the case where there is no module code for a <file,event> pair. Essentially the only overhead is one extra function call from the stub to the actually library and two if-statements.(For both a pre-function and post-function call)

For a call other than open, the mask is consulted for both the pre and post events. If no call is indicated, the call completes normally. If a call is indicated the filename, call arguments and connection id are sent to the server. For calls that take a file descriptor, the filename comes from a reverse translation of the file descriptor. This is done by the filename that is sent to the glibc open call, because this is the filename the file descriptor refers to. This does not have to be the same name as was originally sent into the system call. The translation is the original filename given to the open from the file descriptor returned by the open call. This means the file descriptor and name do not have to go together, except in a logical sense. This is the most useful combination of information for the stub libraries.

3.3 ACFS Server

The active file system server (acfsd) runs is a user-level process that answers from the stub library and returns responses to those requests. It has four areas of responsibility:

1. Answering stub library requests
2. Maintaining file to library associations
3. Managing module code
4. Running module code

3.3.1 Stub Library Requests

The server listens to a single door on which the stub libraries make requests. When a request is received, it is handled in a new thread. The server determines the type of call that generated the request (i.e., open, close, read write) from the data sent to it from the stub. The request type and stub data are forwarded to a handling routine which invokes the appropriate functions for the request and returns the result and a status code. The status code and the results generated by the call are returned to the calling stub library.

3.3.2 Library Associations

The server maintains information about what functions should be called when a system call is invoked for a particular file. These associations are stored in a gdbm database, and only the server has access to this database.

The database contains the following tables: the library table, the association table, and the mask table.

Library Table The library table contains one record for each active module that has been registered with the server. Currently, the library table consists of the name of the library and a short description of the library's function.

Association Table The association table stores lists of libraries that are keyed by the a file name and an integer representing the function (e.g., read). For example, if the library *libgzip.so* (a library that decompresses a file on open and compresses it on close) were associated with a the file /tmp/foo, two entries would be made in the database: one for opening /tmp/foo and one for closing it. The value of each entry in the table would contain a list whose only element is a reference to the compression library, *libgzip.so*.

Mask Table The mask table is keyed by file name and contains a mask that indicates active associations for a particular file. The mask table entry for the file /tmp/foo (from the above example) would indicate that the open and close calls are both active. The mask table servers two purposes. It allows the server to determine which entries a file has in the association table. For example, if the file /tmp/foo was deleted, the server would know that an open and a close entry must be removed from the association table. Also, the mask is returned to the stub library on all *pre-open* calls. The mask tells the stub which functions are active for a particular file. The stub uses the mask to determine for which calls it should contact the server. This reduces requests to the server for which their are no ACFS calls.

3.3.3 Dynamic Library Management

For module code to be run by the server, it must be compiled as a dynamically loaded library, it must reside in the server's library search path, and there must be an entry for it in the server's database. Libraries are loaded by the server on demand.

When a library is loaded the library handle is cached, and a timestamp is associated with the handle. Each access to the handle (to resolve function symbol addresses) updates the timestamp. Some libraries may be used only occasionally, and infrequently used libraries may accumulate over the lifetime of the server. To address this problem, the server is capable of purging libraries that have not been used over some arbitrary interval. Purging unloads the library and removes its cached handle. A purged library will be reloaded when needed. This function could be used to update libraries to a more current version. Purging over a time interval of zero would cause all libraries to be unloaded and then reloaded when next referenced.

3.3.4 Running Module Code

The server must be able to find the appropriate functions for a given <file,event> pair. The server requires that functions written for ACFS conform to a standard naming convention. In our current implementation, the a function name is the concatenation of the name of the library to which the function belongs and an integer representing the type of call. For example, the function written for the library *libgzip.so* to handle *pre-open* calls would be named *libgzip_16* (where 16 is the integer used to represent the *pre-open* function).

The server supports associating functions from more than one library with a <file,event> pair (we call this chaining). Chained functions behave the the same way as Unix commands behave when connected with a pipe. Each function's output becomes the input for the next function in the chain.

3.3.5 Handling a Request

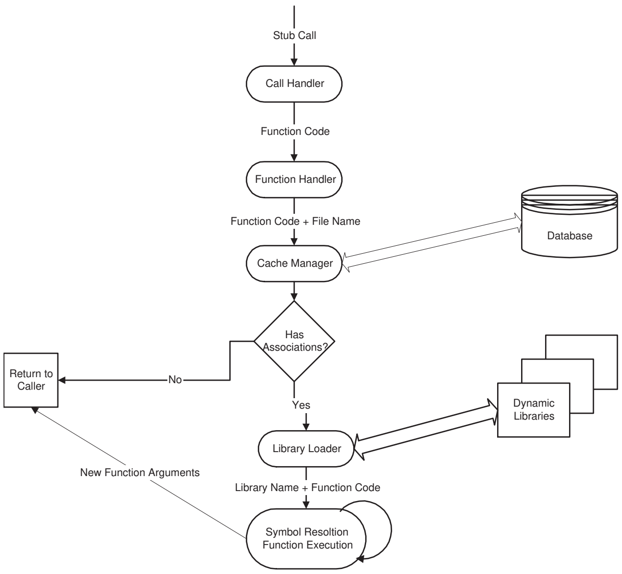

Figure 1: Server Control Flow Diagram

Figure 1 illustrates the flow of control through the server. Each of the major server components listed in the diagram will be discussed below.

Call Handler The call handler waits on the door used by the stub library routines for requesting service. When a call is received by the call handler, it extracts the function code (identifying the type of call) and sends the function code and client data to the function handler.

Function Handler The function handler dispatches requests to the routine which handles a particular function code. For example, *pre-open* and *post-open* calls are both dispatched to the open function handler. The function handler unpacks the data sent to it by the stub library. Included in this data is the name of the file on which the active function call is being invoked. The file name and the function code are forwarded to the cache manager.

Cache Manager The cache manager is a front end for the gdbm database discussed in the previous section. It is passed a function code and a filename and returns a list of libraries associated with a <file,event> pair (or an error code if there is no code associated with the <file,event> pair). Because building a list of libraries can be expensive, library lists are cached. However, in the current implementation a negative result (no associated code) is not cached.

If the cache manager returns a negative result, the original arguments are repackaged and control is returned to the client stub.

Library Loader If a <file,event> pair has at least one associated function, then the libraries in the list returned by the cache manager must be loaded. The library loader is responsible for library management. It is passed a library name and returns a handle to the loaded library. The details of library management have already been discussed above (Section 3.3.3).

Symbol Resolution For each library associated with a <file,event> pair, the library handle returned by the library loader is used to resolve the address of the appropriate function. The method used to determine the name of a function was described in Section 3.3.4.

After a function's name has been resolved, it is executed. The arguments passed to the server from the stub library are passed by reference to the function. If the name for a function cannot be resolved, execution is suspended and an error code is returned to the stub. Execution will stop even if the unresolved function is part of a chain of functions. The next function in a chain may depend on the processing done by the unresolved function, so continuing the chain of execution would just put the next module into an inconsistent state.

3.4 Module Code

A module simply contains user code that is compiled into a shared library object and executed when there is an associated <file,event> pair. This is really where the functionality of the basic file system can be extended by a user. There are few restrictions to what the user code can do as long as it adheres to some general conventions necessary to get the module to work with the rest of the active file system. This allows the user great freedom in creating modules for a variety of purposes.

Each function in a module must be appropriately named so that it can be located, and the signature for each function must match that of the associated call type. The naming convention for functions within a module was previously described in Section 3.3.4. The function given below follows this convention, and so the library the function belongs to is *libps2pdf.so* and the function is called on a *pre-open* event. The signature of the function given below matches that of a standard *pre-open* call and accepts the necessary arguments.

The arguments provided to a function vary based on the call type it is associated with. However, there are certain arguments always passed to a function. These include the path of the file, a connection id, and a structure containing various credentials. Values that the function is allowed to change are passed by reference, the rest are passed by value. The path of the file the event was called on is provided so that the file can be manipulated. A connection id is provided so that the module can keep state, and know which instance it is of all the modules. The credentials contain the real user id, real group id, effective user id, effective group id, and the pid of the client which made the original system call. These are given to the function so that the user can identify the caller and make appropriate decisions based on the information. In addition to these general arguments passed to each function, there are arguments passed specifically for the type of call. For example, in the function below, a *pre-open* call additionally requires mode and flags arguments.

```
void libps2pdf_16( char **path,int *flags,
   mode_t *mode, long cid,
   door_cred_t c )
{
char *newPath = (char*)malloc( 256 );
char sysString[1024] = "/s/std/bin/ps2pdf ";

genTmpName( newPath, cid );

strcat( sysString, *path );
strcat( sysString, " " );
strcat( sysString, newPath );

unlink( newPath );
system( sysString );

free( * path );

*path = newPath;

}
```

A function can either modify one or more of the variables passed to it before they are sent back to the client, or perform some side-effect of the system call that is allowed from within a function. The function above does the former, by converting the .ps file specified in the given path to a .pdf file, and returning the path to the .pdf file.

For the system to be practical and convenient to use, it should not be difficult for the user to create module code. This is true of the current implementation as the sample function above clearly demonstrates. It is only ten lines, and yet, with the use of an existing application, it provides the convenience of being able to dynamically convert a .ps file to a .pdf file on an open call.

4 Performance

The performance of ACFS is a significant concern. The file system will handle at least a subset of system calls made on behalf of files that do not have any active functionality. In our design, we sought to minimize the time required to contact the server. However, system call performance for files with active functions will vary based on the complexity of the function and the quality of the module code.

To evaluate the performance of ACFS, we sought to determine the overhead incurred by contacting the server. Further, we wished to determine the effect of ACFS on real applications. To evaluate "real world" performance, we used two ACFS modules: a module that performs postscript to PDF (hereafter *libps2pdf*) and a module that compresses unopened files (*libgzip*).

4.1 Call Overhead

Call overhead can be broken into two components:

Baseline Baseline overhead is that which can not be reduced by optimizing ACFS. Overhead here involves the time to perform a system call and the overhead incurred by IPC in contacting the server.

ACFS ACFS overhead comprises time spent in the stub library and the server. Naturally, the time spent in the server accounts for most of the overhead in this category.

In order to partition the overhead, we ran several micro benchmarks. All of the benchmarks were run on a 200 MHz Pentium workstation running Solaris 2.6.

Since we did not have privileged access to the machine, we did not run in single user mode. At the time we performed our benchmarking, the machine was running in only two xterms and the benchmark applications.

4.1.1 Baseline Overhead

Baseline overhead has two major contributors: the cost performing a system call and the overhead incurred by IPC to contact the server. In a production implementation, we would like ACFS's overhead to approach the sum of these two costs.

To measure the time required to perform a system call, we constructed a program that opened and closed a file some number of times provided by the user. The file was of zero-length and was located on the local disk of the test machine. We timed the program using Unix time(1) program over 100000 open/close iterations. We determined that the amount of time required to open and then close a file was 28 μs per open/close pair.

The overhead of Sun's doors IPC was determined in a similar manner. We constructed a client and a server. The client called the server and passed an integer. The server returned the integer in its response. For our benchmark, the client contacted the server 100000 times. The run time of the client was measured with time(1). From this benchmark, we determined that the time required to contact and receive a reply from the server was 58 μs.

The overall baseline overhead of the system is sum of system call time and IPC was 86 μs.

4.1.2 ACFS Overhead

Next we must determine the overhead of single ACFS call. For this measurement, we chose the open call, as the open call is executed once for each file, whether or not the file has associated modules. To measure the overall overhead of ACFS, we used a procedure similar to that described for the open/close benchmark described in Section 4.1.1. We made the following modification.

1. We preloaded the stub library so that open calls were intercepted by ACFS.

2. The stub library was modified so that close calls were not intercepted by ACFS.

3. A library that defined an association for the *pre-open* call was constructed. The the function that was executed on a *pre-open* simply returned control to the server (we call this library *libnull*).

4. The server, acfsd, was started to answer calls from the stub library.

We ran an open/close sequence on two files. The first file had no associations. The second file's *pre-open* event was associated with *libnull*. Our test program ran the open/close sequence 100000 times on each of the files.

The test of the file with no associated module required 197 μs per call. Associating *libnull* with the *pre-open* event added 64 μs to the per-call cost, resulted in 262 μs per call. In order to determine where this time is spent, we constructed a test program that ran a modified version of the stub code without contacting the server or making system calls. This test indicated that the time to execute the stub code for open and close was 2.5 μs per call. This suggested that the remaining overhead was in the server.

We benchmarked the server's function handling routines and determined that the cost of handling a call without an associated function was 51 μs. In this 51 μs, the server classifies the request and dispatches it to the appropriate function handler. The function handler queries the database and receives a negative result. It retrieves the mask for the file and returns the data passed to it by the client.

The cost handling a request with one function call was 118 μs. The question becomes, what in the server accounts for the additional 67 μs to run a function? We benchmarked the call of a null function and determined that it takes about 1 μs per call. Our initial belief was the the cost of resolving the address of a function in a dynamic library was very high. However, we found that it takes 3 μs to resolve a function symbol. We believe that the remaining 63 μs is explained by the additional server code that is run when a function is associated with a call (which includes string manipulation to construct the function name and a call to gettimeofday to update the library time stamp).

There remains another 50 μs of time unaccounted for by our analysis. We believe that this could be accounted for by context switching. However, this belief remains unconfirmed.

Figure 4.1.2 shows each component of the ACFS benchmark without an associated function. Figure 4.1.2 show the same information for calls associated with *libnull*. The reader should note that IPC, acfsd, and other sources of time take up roughly equal proportions of a ACFS call that has no associations. However, in the *libnull*-associated benchmark, acfsd dominates all the other sources of run time. Clearly, acfsd would be the target of optimization in a second implementation of ACFS.

Source	Time	Percentage
IPC	58 μs	29%
other	55 μs	28%
acfsd	51 μs	26%
open/close	28 μs	14%
stub	5 μs	3%

Figure 2: Performance: No Association

Source	Time	Percentage
acfsd	118 μs	45%
IPC	58 μs	22%
other	53 μs	20%
open/close	28 μs	11%
stub	5 μs	2%

Figure 3: Performance: *libnull* Association

4.2 Postscript to PDF Conversion

Now that we have some understanding of the overhead associated with an ACFS call, we can move to the performance of ACFS in "real" applications. Here we investigate the performance of the *libps2pdf* module when used with Adobe Acrobat Reader.

The *libps2pdf* module provides active functions for the *pre-open* and *post-close* events. On a *pre-open* event, the module receives the path of the postscript file to be opened. It converts the postscript file and saves the resultant PDF file as a temporary file (using GhostScript). It returns the name of the temporary file to the user. The *post-close* event unlinks the temporary file.

To measure the performance of Acrobat Reader, we invoked the Acrobat Reader on the command line with the *-toPostScript* option. In this mode Acrobat Reader converts the PDF file to postscript and then exits. The application's GUI does not display, so it is possible to get accurate timings with this option.

The application was invoked under three conditions:

pdf ⇒ ps Under this condition, Acrobat Reader was used to convert a PDF file to a PostScript file without ACFS running.

pdf⇒ps (ACFS) This condition was similar to the above; however, ACFS was running. None of the files used by Acrobat Reader under this condition had any associated module code.

ps⇒pdf⇒ps (ACFS) Here Acrobat Reader was given a PostScript file with an association to the *libps2pdf* module. The PDF to PostScript converter was invoked on the PostScript file. When the converter opened the file the *libps2pdf* module converted the file to PDF format. Acrobat Reader then converted the PDF file back to PostScript.

For each of the above conditions, we used the same 36 KB PDF file.

Activity	Time
pdf ⇒ ps	1.97 sec.
pdf⇒ps (w/ ACFS)	2.01
ps⇒pdf⇒ps	2.97

Figure 4: Acrobat PDF to Postscript Translation

The results are presented in Figure 4.2. From the figure, it is clear that ACFS contributed little overhead to Acrobat Reader's run time when the *libps2pdf* module was not being invoked. The 0.04 second difference between the first and second condition is the overhead incurred when ACFS is running, but no module code is executed. Logs from acfsd indicate that translating a pdf file to PostScript calls open(2) 26 times. A significant of this low overhead is the call mask, which allows to stub to refrain from calling the server after a *pre-open* if the call has no associated module code. In this study, there were 3486 calls to read, none of which reached the server due to the call mask. This reduces the runtime by 0.69 seconds, or about 30% of run time, on reads alone.

The third case runs about 50% slower than Acrobat Reader without ACFS. When examining the acfsd's logs, it was determined that Acrobat Reader opened the active file twice. As a result, two translations from PostScript to pdf were incurred.

4.3 Compression

Our third benchmark makes use of the *libgzip* module. This module provides active functions for the *pre-open* and *post-close* events. The library assumes that the file on which it operates is compressed using gzip prior to calling the active open(2) function. When a file is opened, the *libgzip* module decompresses the file to a temporary file. It returns the name of the temporary file to the caller. On close, the *libgzip* compresses the temporary onto the original file.

We evaluated the performance of ACFS using *libgzip* by compressing the C and C++ source files for acfsd (header files were not compressed), and associating each file with the *libgzip* module. This comprised 2200 lines of source code in 14 files. We then measured the amount of time spent by make(1) to build the project. All compilation was performed on the local disk of the test machine.

The we evaluated three conditions:

non-ACFS The C/C++ source files were uncompressed, and acfsd was not running.

ACFS non-assoc The C/C++ source file were uncompressed, but acfsd was running. Every call to open resulted in call to acfsd (on the *pre-open* event).

ACFS The C/C++ source files were compressed and associated with *libgzip*. Every call to open(1) for any of the source files caused that source file to be uncompressed to a temporary file.

We measured the time taken to do a parallel make of acfsd in addition to a make that compiles each source file sequentially. For the parallel make, we passed GNU make the command line parameter "-j3" which compiles three files simultaneously.

Activity	Make	Parallel Make
non-ACFS	43.57	42.23
ACFS non-assoc	44.68	42.94
ACFS *libgzip*	46.30	43.42

Figure 5: gzip compression

Figure 4.3 shows the results of the two types of make invocations under the three conditions described above. The difference between the *non-ACFS* case and the *ACFS non-assoc* case is 1.11 seconds for sequential make and 0.71 seconds for the parallel make. Given that compiling acfsd takes 40 seconds, the one second of ACFS overhead is not noticeable by the user. An examination of the acfsd's log file indicates that there were 2886 *pre-open* events serviced by acfsd. The call mask eliminated all read, write, and close calls to the server.

When *libgzip* is associated with the source files, the build time is 2.73 seconds slower than the case in which ACFS is not running (1.19 seconds slower for parallel makes). Associating *libgzip* with the source files adds an additional 1.62 seconds to the overhead of running ACFS (the overhead is 0.48 seconds in the parallel case).

5 Discussion

In this section we will discuss how we have either met or approached our previously stated goals for an active file system. The goals were to design and implement a system that met our definition of an active file system, for that system to achieve as close to ideal performance as possible, and for the system to provide a basic level of security while not impeding greater security measures from being added.

5.0.1 Definition

Our working definition of an active file system is a system which allows user code to be executed on specific file system events. We have met this definition by implementing a system which allows users to execute code they have written on <file,event> pairs.

5.0.2 Performance

In Section 1 we stated two goals for the performance of our implementation of ACFS.

1. When a file has no associated module code, there should be no ACFS overhead when running system calls.

2. When a file has module code associated with a system call, the overhead should be minimized.

We believe that our implementation has met the first goal. This is supported by the data presented in Section 4. The benchmarks of Acrobat Reader (Section 4.2) and GNU make (Section 4.3), suggest that it is possible to use ACFS with an application and see no perceptible difference in that application's performance.

Application performance under ACFS is substantially aided by the call mask optimization that we made to the system. If an application opens a file that has no associated modules, it needs to contact the server only once (during the pre-open event). Using the mask, the stub library can refrain from calling the server on events that have no associated module code. As we have suggested in Section 4.2 and Section 4.3, this optimization noticeably reduces the time to execute an application under ACFS.

Next, we refer the reader to Figure 4.1.2 which shows the relative contribution each source of overhead makes to the a call to ACFS. The ideal time for such a call is 86 μs, or the overhead incurred by system calls and the time required to communicate over IPC. We measure the overall time for a call to be 197 μs. Thus, we are 129% slower than our optimal performance goal. However, since a call with no association is made once over the lifetime of a file descriptor, the cost of this call is practically insignificant.

Our second goal states that the call overhead should be minimized when executing a system call with associated module code. Figure 4.1.2 shows that overhead from acfsd accounts for 45% of time required to make an active call. We believe that this time could be reduced through careful optimization of the server code. It should be noted, that our ACFS implementation was a reference implementation, and was not written or intended for production use. However, despite our unoptimized implementation, our macro benchmarks suggest that we achieve acceptable performance. Both of these benchmarks associate active code with open(2) and close(2). These calls are executed only once per file descriptor obtained from the system. If read(2) or write(2) were associated with active code, they could be executed much more frequently than open or close. In this situation, our performance overhead may be much more noticeable.

One aspect the ACFS's performance that has yet to be resolved this the the true source of the overhead we call *other* in figures 4.1.2 and 4.1.2. We have suggested that this overhead may be accounted for by such activities as scheduling acfsd. If this is the case, then raising the priority of acfsd would reduce the contribution of *other* overhead to the overall overhead.

5.0.3 Security

For the active file system to at least meet the minimum level of security necessary for wider usage, it must provide mechanisms to distinguish between authorized and unauthorized users of the system, and it should provide fault tolerance to prevent malicious or incorrectly running user code from bringing down the rest of the system. Also, the system architecture must not prevent additional security measures from being added.

Our current implementation provides the necessary facilities to distinguish between different users, and to specify which users can and cannot add, modify, or run code. One way of securing the system would be to use these facilities to prevent anyone but root from adding modules, and to only allow the owner of a file to associate it with a <file,event> pair. Assuming root is trusted, this provides the necessary level of security. However, this method has the disadvantage of only allowing users with root access to add modules to the system.

To restrict execution of module code to only authorized users, the creator of the module can insert a check to see if the caller's credentials are sufficient to run the code. If not, the call will return to the user without having run the function.

The other issue related to security is fault tolerance. One client should not be able to bring down the rest of the system. This is an area in which our implementation does not quite meet one of the stated goals. Currently, there is nothing to protect the rest of the system from an incorrectly running client. However, there exist a number of solutions to this problem. One possibility would be to run the user code in an external process. This way, the damage that one module could do would be limited. However, this sacrifices performance at the cost of safety. A compromise would be for the system to allow trusted code to run within its address space, and to force untrusted code to run in an external process. This provides a balance between performance and security in the system.

6 Future Work

Some areas exist for future work that could enhance the current implementation of the active file system by making it more convenient, flexible, and fault tolerant.

The ability to allow any user of the active file system to write modules and add them to the system would be of great benefit. This would give every user the opportunity to customize the system to his/her own needs.

Support for ordered chaining would improve the flexibility of the system by allowing users to perform multiple actions on their files. This would allow the user to perform more complex transformations on his/her data by simply chaining together each of the modules necessary for one of the transformations.

The issue of fault tolerance was touched upon in Section 5.0.3 in relation to security. It was mentioned that a possible solution to protect the rest of the system from an incorrectly running client would be to run the user code in an external process. However, this reduces the performance of the system. A better solution may be to implement some type of sandboxing for the threads running user code, so that the damage rogue code can do is limited, while the performance advantage offered by threads is retained.