Active File System

Deb Deppeler and Evan Greenland
{deppeler, evan}@cs.wisc.edu

CS736 Advanced Topics in Operating Systems
Spring 2000
Professor: Remzi Arpaci-Dusseau

May 12, 2000

Abstract

The Active File System (*active*), an extension of the minix file system for Linux, provides automatic logging of individual file operations. In addition, we present a prototype of a file access auditing system as a configurable option available both internally within the *active* module and as an external user level program which communicates with the file system module and describe how the prototype can be extended to support the execution of arbitrary user commands in response to file operations. Timing tests are provided that prove the viability of these different approaches.

1 Introduction

Generally speaking, an active file system is one that expands the functionality of a conventional file system to include the execution of user code in response to file operations. Whereas a conventional file system performs the basic file operations without modification, an active file system might support user intervention that could not only add functionality in addition to the standard file operation but would be able to even affect the results of the standard file operations.

In this discussion that follows, we present a prototype for a Linux file system that works in concert with a user level daemon to recognize file operations on specific files and then execute user assigned commands within the context of the user level daemon. In addition we describe the modifications required to expand the presented functionality to include the ability to affect the results of the standard file operations.

In the course of our work, we addressed several areas of concern such as: where should the user code be stored, when should it be executed in relationship to the file operation it is assigned to, and what types of executable code can and should be allowed. Our project implements a limited set of answers to these questions. We chose answers that we found most interesting (flexible) or feasible given our limited time frame. Our goal was to implement this functionality in some model rather than search exhaustively for every possible model. Intermediate targets were chosen and named Active I, Active II and Active III.

The remainder of this paper is divided into the following sections: section 2 describes design options and decisions, section 3 describes the implementation of Active I through Active III, section 4 discusses the results of performance tests on each version, section 5 describes future work and potential extensions for the active file system, section 6 discusses some related work, and section 7 lists our conclusions.

2 Design options and decisions

We explored four implementation models for our active file system. These models explore the range of allowed functionality, i.e. arbitrary user commands vs. a limited set of system supplied utilities, and the level at which the file operation should be recognized, i.e. within the Active File System itself or at a higher level.

The first approach was to simply allow the storage of user supplied code within the inode of a file. Some of the implementation concerns with this approach included the need to store multiple, distinct code blocks of unknown size for each of the supported file operations and the danger to the entire operating system of allowing arbitrary user code to be executed from within the context of the kernel file system. The type of user code was also an issue in this model. It the implementation would likely depend on whether the code to be executed was system commands, compiled C programs, Java interpreted byte-code or other.

The second, a more conservative approach, was to allow only a limited set of system defined utilities to be available for execution within the kernel in response to specific file operations on a file. The user would be allowed to specify which actions should be associated with a particular file, but would only be allowed to choose from the limited set of supplied activities. This model was safer since only the *trusted* set of functions could be executed, but did not meet our goal of executing arbitrary user code.

The third possibility added a front end to the kernel file system that intercepted system calls for the supported file operations and compared the file operation and file name against a user supplied list of <file operation, filename, command> tuples. If a match was found, the associated command was executed from within the file system front end. This approach was quite intrusive since it required intercepting all system calls to the supported file operations, even those meant for other file systems. We had the option to simply ignore system calls to file systems other than the Active File System and continue with the normal execution of other file system calls, but this approach seemed to impose too large a penalty on the entire system in order to meet our goals.

The fourth approach, and the one developed in this project, supports the execution of arbitrary user assigned commands in response to file operations on particular files. The user again supplies a list of <file operation, file name, activity> tuples and when a file operation-file name is recognized within the Active File System, the activity is executed. System security is maintained as the assigned activity is executed within the context of a user level daemon.

3 Implementation

The design of our active file system called for the implementation of a system that would support the execution of arbitrary user-assigned commands in response to specific file operations on specific files. The kernel file system would monitor file operations and when it recognized a supported file operation on a referenced file, it would execute the command program that was associated with the <file operation, filename> pair.

We chose to modify the Linux operating system because of its freely available kernel source code and used its minix file system as our starting point because its design is straightforward, similar to UNIX [8], and has better than average available documentation [19].

In order to focus our initial development efforts on the file system infrastructure and functionality and not be distracted the by implementation of support for execution of arbitrary commands on referenced files, we narrowed the scope of our first efforts to execution of a single action in response to supported file operations. To identify a single action that would be meaningful within the context of an active file system, we conducted an informal survey of software engineers who work with and build system tools and asked them what functionality they would like to see. An analysis of our survey results suggested that providing audit information would be interesting to this group. This audit information should describe file accesses; minimally reporting the file operation, the filename, and the user id.

In a further attempt to focus the scope of our initial implementation, rather than recognizing all file operations, we limited the set of supported file operations to create, delete, open, and close.

Our first order of business was to set up a development environment. At UW Madison, operating system course projects by graduate students are usually done in the crash-and-burn lab where resources for Linux projects comprise onlyl 4 machines which are naturally in great demand. To circumvent the logistical complications of sharing disk space, hours of access, and most importantly defined times within which we could freely crash and reboot the system, we decided to expand our development environment to also include Deb's laptop (Pentium) and a machine that Evan borrowed from Persoft. While all three machines ran Linux version 6.1, which gave us greater flexibility in terms of development time and ability to crash the system, the three had different point releases which necessitated merging our source code for each incremental release. Tools were developed to simplify installation and cleanup of our development environment on crash2 and to expedite the merging of source code between our different development machines. To insure a stable delivery platform, the UW machine crash2 running Linux 6.1 2.2.12-20 on an i586 was used as the target machine for this project.

3.1 Building a Linux File System

The development effort began with learning about the Linux installation and configuration environment, tools, and processes. We initially configured and built a custom Linux kernel with the intention of using this kernel as our development base but soon learned that the kernels that were in place on our machines were already configured and built with support for file systems as modules. We learned about modules and soon successfully built and loaded the minix file system module [16, 7, 5].

With minix available as an active module resource, we created a file to hold the file system, converted this file to a file system, and then mounted the file system. Details of these operations are included in the file README that resides in the top level of our supplied source code tree.

Once we were able to build and load a file system module, the next step was to clone a file system and define and load this new file system. We cloned the minix file system to create (our) Active File System. In addition to creating a new file system subdirectory (/usr/src/linux/fs/active), other changes included modifying the declaration of struct file_system_type in inode.c to rename this file system "active", changing the directory makefile to build active.o rather than minix.o, modifying /usr/src/linux/fs/Makefile to know about its new active subdirectory, and adding active.o to /lib/modules/linux/modules.dep so that modprobe would recognize the active module. To simplify our development efforts, we also modified /usr/src/linux/Makefile, copying and modifying the makefile target 'modules' to only build the file system modules, rather than building all the supported modules whenever a change was made.

With these changes in place, we were able to build and load the Active File System , essentially a renamed minix file system.

3.2 Kernel Level Auditing

The next step was the implementation of kernel level auditing. This process began by tracing file operations within the Active File System . Since a Linux file system supports the Virtual File System [2], we were able to identify the methods called for each of the supported file operations, creating stub methods if they were not already implemented within minix.

We created a function, minix_write_log(), to act as the dispatcher for the active functionality. In each of the supported file operation methods we added a call to minix_write_log() that passed the name of the file operation and the fully qualified path name of the affected file. In create and open, minix_write_log() is called after the file operation is executed, while in delete and close, the dispatcher is called before the file operation is executed. Other possibilities for the placement and action of minix_write_log() within a file operation method are discussed in Section 5 below.

In addition to receiving the name of the file operation and the affected filename, minix_write_log() retrieved the current user id and a time stamp and used these values to build an audit log entry of the form: line identifier, file operation, filename, user id, and time stamp. The line identifier is a counter modulo a large number that was added to make it easier to identify individual lines in the log file. These log entries were then written to an external, human readable log file, /usr/src/linux/fs/active/active.log.

After identifying the file operation methods, some of the challenges faced in this phase were retrieving the fully qualified name of the file being operated on and how to open, write, and close an external file from withina file system. Frustrations arose as we discovered that some of thedocumented interfaces [2, 19] were not supported in the versions of Linux we were using. The lesson we came away with is the popular solution to all Linux questions, *read the source code*. We found /proc/ksyms to be an especially useful tool in this regard.

3.3 User Level Auditing

Once the file system infrastructure in place, our next refinement was to create a user level listener daemon, a named pipe for communication between the kernel file system and the listener daemon, and to modify the kernel to use the named pipe to pass its audit log entry to the listener who then writes it to the log file.

The listener daemon runs in the background in user space. Its primary job is to block on the named pipe waiting for input from the kernel file system. The input comes in the form of the same audit log entries that were manufactured in the previous phase, with an explicit EOL added at the end of each line of input. When the listener receives a line of input, it writes the log entry to the external log file, and then returns to block and wait for additional input.

The named pipe was created using the Linux 'mkfifo' command [9] and is used for one-way communication from the kernel file system to the listener daemon. (Again, please refer to the README file in the top level of our source code tree for details of the named pipe naming conventions and activation and use of the listener program.)

The final step in this phase was to modify the kernel active file system method, write_minix_log(), so that after it built audit log entry string, rather than writing directly to the external audit file, it did non-blocking writes to the listener daemon via the named pipe and then continued on without waiting to hear that the listener wrote the audit log entry to the audit log. This phase thus implemented asynchronous activity between the kernel level file system and the listener daemon.

3.4 Measurements vs. Better Engineering

With the kernel level file system noticing file operations and notifying the user level listener of these events, and the listener successfully writing the audit log, the next steps towards implementing a truly active file system would have been the addition of support for specification of commands to be executed when file operations occurred and then the execution of these commands at the appropriate times. Beyond the implementation itself, there remained the question of the expedience of this approach: what overhead does this active functionality infrastructure impose? In recognition of limited time and resources, we were advised to direct our attentions to these performance measurements rather than continuing to implement the active file system functionality. Please refer to Section 5 for a description of the implementation of the full active functionality.

3.5 Round Trip Communication

The current model employs an asynchronous paradigm in which after notifying the listener daemon of the file activity, the kernel file system continues on its way without waiting for a response from the listener. Another interesting possibility for active file system functionality is synchronous communication whereby the kernel waits until it hears that the listener daemon has completed its activity before continuing to handle the next file operation.

Support for synchronous communication was implemented with the addition of a second named pipe. This pipe was used by the listener daemon to notify the kernel that its activity has been completed. The listener writes in a non-blocking mode to this pipe, while the kernel, after having sent the audit information to the listener, waits to read from the second named pipe in blocking mode until it receives an acknowledgement from the listener. The synchronous communication model introduced another level of functionality and, as is common in such cases, was expected to introduce another level of overhead.

4 Results

To determine the amount of overhead introduced by the different models of active file systems we conducted a series of tests that compared the times required for each of these systems to perform the same file operations. The baseline system was determined to be the newly cloned active file system, Active, without any external auditing capabilities. The first level of auditing functionality, Active I, supports writing to the external audit log from within the kernel file system. The second, Active II, introduces asynchronous use of a named pipe, with the listener daemon writing to the audit log. Active III supports a second named pipe with synchronized activity between the kernel file system and the listener daemon.

In order to isolate the effects of the communication mechanisms from other file activity, in all four models and when applicable, the creation, opening, and closing of the external audit log and/or the named pipe(s) were done at module and/or daemon initiation and closing time and so did not contribute to the measured times.

To focus the tests on the effect of the communication mechanisms for file auditing capabilities, the tests consisted of repeated opening and closing of a test file via a Perl script, CmdTester.pl. The numbers displayed in the following charts reflect the average values for multiple (5) runs of each test. Each run consisted of opening and closing the test file one hundred thousand (100,000) times resulting in 200,000 file operations. The command used to run the tests was "time perl CmdTester.pl 100000"

Figure 1 shows the average amount of user level time each run used. Each model used approximately 4 seconds of user time more than its predecessor so that the overhead in the worst case scenario, Active III, is only 57ns per file operation more than in the baseline case.

Figure 2 shows the average amount of system time used for each run. Whereas the baseline model averaged only 52.5ns of system time per file operation (10.518 sec / 200000 file operations), we did not see a significant difference between the three models that implemented some level of auditing, with an overall average time of 145ns per file operation.

Figure 3 shows the total elapsed time for each run. Here we see an increasingly large penalty with each layer of implementation. The simplest implementation, Active I, costs 120ns per file operation more than the baseline case. Active II, with its use of a user level daemon, adds an additional 262ns per file operation, while Active III, with its synchronous activity, costs 571ns more than the corresponding baseline operation. Careful observation of the data returned by the "time" command showed that the percent of the CPU available to the test program varied markedly between the first two and the final two test runs. While CmdTester.pl had virtually the entire CPU to itself when activity was confined to the kernel, once the user level listener program was introduced, CmdTester's share was cut proportionally to about 58$additionalcostincurredwhenrunningActiveIII.Theadditionalcostofalmost$8named pipe with blocking.

As described, these tests were conducted with file access auditing as the only supported activity executed when a file operation occurred. It is expected that with the introduction

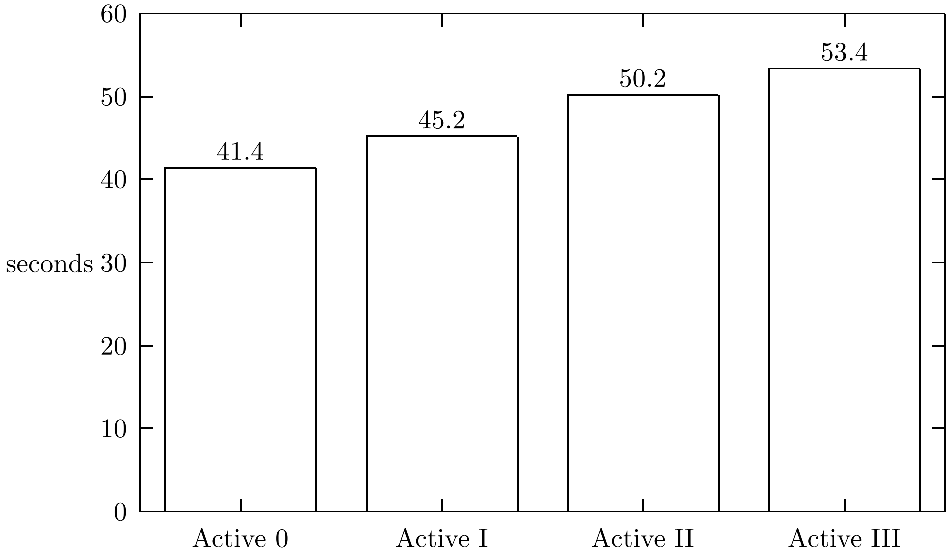

Figure 1: Time spent in user level for 100,000 open/close cycles

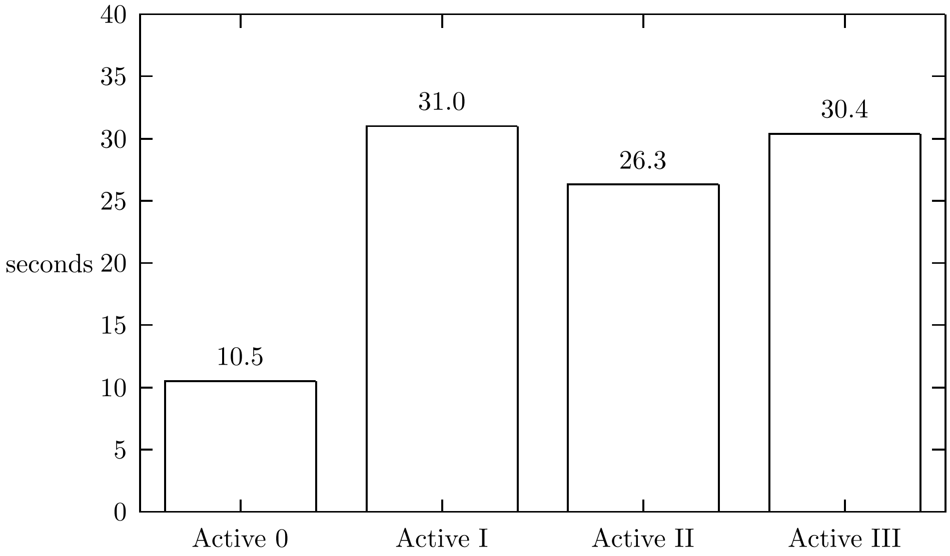

Figure 2: Time spent in system level for 100,000 open/close cycles

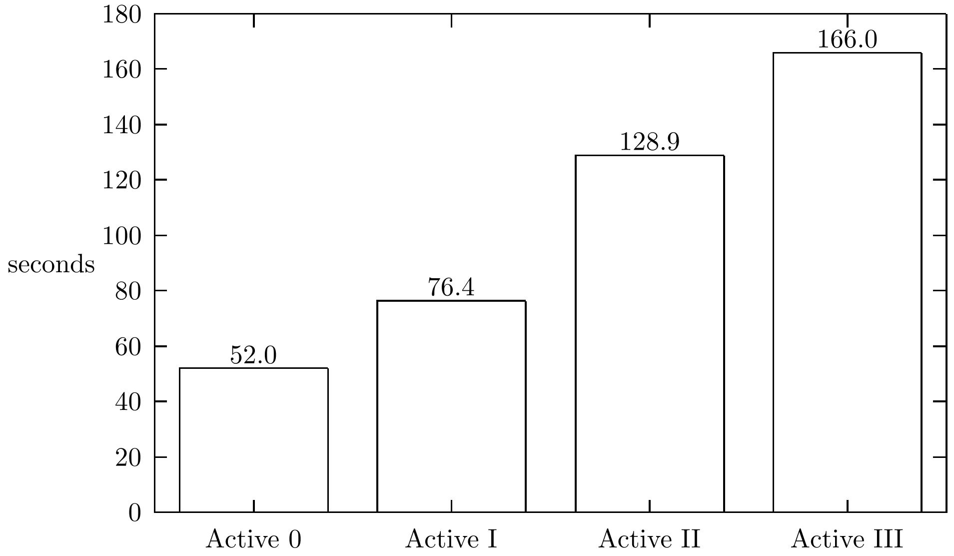

Figure 3: Elapsed time for 100,000 open/close cycles

of other more interesting and sophisticated user-assigned commands, the cost of command execution will increase, thus increasing the overall cost of supporting execution of arbitrary user commands vs. a baseline, non-active, file system. This will be especially noticeable if the user desires synchronous execution and use of return values from the listener daemon.

Comparisons of test results for Active I vs. Active II or Active III show that implementation of active functionality entirely within the kernel is clearly faster than execution at the user level. Tradeoffs for this speed include the increased difficulty of coding at the kernel level vs. at the user level and the need to introduce greater levels of security and permissions checking to guard against malicious or careless user coding.

Comparison of test results for Active II vs. Active III show an increased cost with the use of synchronous communication. Use of round trip communication should only be done when the application warrants it. Section 5 explores some of those possibilities.

In general, we see that with the addition of each level of functionality, the overhead of the implementation increases. This argues for the introduction of support for user control within the active file system to indicate which level of activity or communication to enable, and whether or not to enable any degree of file system intelligence at any given time.

5 Potential extensions of the Active File System

5.1 Addition of a configuration file

Currently, the Active File System supports only the automatic logging of specific file level operations. The timing of this write is hard coded so that the log is written after create and open operations and before close and delete. To change between Active I through Active III versions of the Active File System , the user must modify active.h and recompile the active module. While this was acceptable for the benchmarks we tested, our intention is to descripe a user-edited configuration file that allows for much broader variety of code to be executed.

The addition of a configuration file that stores active association records would allow the user to control the activity for that particular operation. Each association record would be composed of a set of keys representing file operations, a filename, a command to execute, a flag indicating whether to execute the command before or after the file operation, and a flag that indicates asynchronous (faster) or synchronous (permits error checking and more control) execution of the command and the file operation.

Once the configuration file has been implemented, a wide range of whole file actions become feasible. Possibilities for text files and other ASCII data files include the aforementioned audit log, as well as, automatic compression / decompression, whole file encryption / decryption, sorting, and other text translations, substitutions and possibility even some types of text analysis. There are likely many additional possibilities that have not occurred to us during this work, especially for other file types such as multimedia and database files. Our motto from the movie "Field of Dreams" is, "If we build it, they will come."

The active associations should be stored in the address space of the listener daemon. The modification time of the configuration file can be compared to the last read time to determine if the configuration as been changed and should be re-read. This allows the user to add, edit and remove associations thereby dynamically controlling the activity of the active file system.

5.2 Support for block level operations

While the activities described above are designed to act on the file as a whole, other desired activities or implementations might require a finer level of file granularity. For example, whole file encryption / decryption requires the decryption of the whole file on open, possibly resulting the undesired stored of the decrypted file on disk (albeit temporarily). Alternatively, if the active file operations were modified to act on individual file blocks, it would then be possible to encrypt / decrypt the file on an as-needed block basis so that decrypted blocks would only be in memory when desired.

To support these sorts of file block actions, a number of extensions to the Active File System would be required. Implementation of the flags that indicate whether the action should be executed before or after the file operation is carried out becomes more important as these actions may either create initialization conditions for the file operations or, conversely, act on initialized blocks. The round trip mechanism between the kernel and the user level listener daemon would need to be expanded to allow transfer of file blocks in either direction, and finally, the set of file operations recognized would require a simple extension to include read and write, the file operations most likely to be utilized for file block activities.

6 Related Work

6.1 Overview

There are many research teams working on extending the functionality of file systems and even more that extend operating systems in general. Jones [10] presents a toolkit that makes it easier for user code to intercept the kernel's system calls. These are called *interposition agents* and are a high-level interface to the low-level mechanisms already being provided. This allows the transparent execution of user level code at the system interface. Unfortunately, the code then runs unprotected in kernel space.

Seltzer and Small [17] provide overview of several different extension technologies and their basic trust models. Hardware protection is simple and dependable, but has high cost. Software protection is more complex, but can offer the highest performance. In interpretation, the kernel runs an interpreter to ensure its safety, but performance suffers since interpreting is 10 to 100 times slower than compiled code [15]. Their work suggests *grafting* extensions onto micro kernels to add new functionality, while preserving the desired security level.

Alexandrov, et al. [1] use a user level *Catcher* program to intercept system calls and provide additional operating system functionality without the security risks of letting user code run in system space. They demonstrate their method with Ufo [1], a global file system that is implemented in two modules, a catcher (that intercepts system calls) and the Ufo module. Their Ufo is able to make remote files appear as local files to the user. The real benefit is that the system administrator is not required for any user to attach Ufo to their own process. Only that user's process is affected. This is similar to an early attempt of ours that also intercepted system calls to provide a place to execute our own code. However, our interception was more intrusive, since all system calls, including those for different file systems were intercepted by our module. Thus, it was dropped in favor of a user level daemon that received communication from only our file system.

Ghormly, Petrou and Rodriques describe SLIC [6], an extensibility system prototype for commodity operating systems. SLIC uses *interposition* as a suitable mechanism for making operating systems extensible. But, only trusted code is inserted without modification into the kernel. They describe interposition as follows:

> "*Interposition* is the process of capturing events crossing an interface boundary and forwarding those events to an interface extension. The extension performs some processing on the event and then either passes the event to its original destination or forces the event to return." [6, §2]

This idea corresponds closely with our listener daemon as the interface extension. The Active File System uses a user level listener, similar to the Alexandrov's Catcher [1] and this listener daemon is responsible for writing to the active.log. Eventually, the listener will fork new process executions as configured by the user.

In all the references to extending the capabilities of file systems, we did not find any that address the idea of allowing the user to associate which activity (code) should be performed for which <file operation, user file, code> tuple. What we have found are several examples of adding a specific functionality to the entire operating system and in fewer cases the file system. The rest of the related work presented focuses on the work that most closely relates to the extensions described in Section 5 above. The work cited fall into three main categories.

6.2 Automatic auditing of file accesses

Windows NT has both an event log [13] and an auditing option [12] that can be turned on globally and accessed by the system administrator. Almost all web servers provide a log file that records all pages requested. The World Wide Consortium even has a standard devoted to logging web traffic. [22] There is an entire industry devoted to logging data, like Campbell Scientific's [3] system for tracking various weather related data and Vemco [21], which specializes in underwater temperature and depth tracking. However, none of these suggest the ability of a user to request auditing of a set of file operations for a given file. This is the level of granularity that the Active File System aims to provide to users.

6.3 Automatic Compression/Decompression

Lasus [11] presents a brief introduction to the benefits of using file compression. Numerous compression utilities are available, but even those file systems with compression support built in, require the user to manually select each file that is compressed/decompressed. For example, Windows NT has file compression available at the file level [14]. But, this option is only available to drives formatted as NTFS and there is no automatic support for which files to compress/decompress.

Only three automatic file compression systems were found and two of these, operated on entire drives. The systems DCF/2 v1.4 for the OS/2 platform and Stacker for Warp for FAT-based systems like OS/2 and DOS operate on entire drives. Zipstream [23], which does allow the user to choose which files are to be automatically compressed, is only available for OS/2 drives.

The future Active File System will provide this compression capability and more as the user has the flexibility to determine what actions, in addition to or instead of, occur for each supported file operation.

6.4 Automatic Encryption/Decryption Systems

The Transparent Cryptographic File System [20] project most closely resembles our future goal for automatic encryption / decryption of files. Their work provides a user transparent encryption of data as a solution the to the problem of privacy for distributed file systems. This is similar to our encryption / decryption extension with the exception that our system doesn't attempt transparency, rather it would require the user to select encryption as a global option or on a per file association via the configuration file.

The Secure File System [18] is a set of programs which handles encryption at the volume level on both DOS and Windows. Again, this is not selected per file, but rather for an entire volume.

The makers of Zip Stream have also developed Crypt Stream [4], a product that automatically encrypts the specified OS/2 files.

7 Conclusions

We have demonstrated a plausible prototype for an active file system. Some observations stand out: It is easier and safer to add programmer designed extensions to a file system than it is to design the more general solution that gives the user the ability to assign arbitrary code to be executed. Although there is some overhead associated with extending a file system to be active, the overall cost of these extensions are not significant within the context of the activity that is desired.

Of the actions our Active File System could be extended to support, encryption / decryption requires the greatest amount of additional sophistication. In reviewing the availability of commercial [4] and non-commercial [20] encryption systems, we are inclined to believe that this use of the Active File System is not likely to have the flexibility, performance and robustness already available. The more specific solutions to encryption can be optimized in ways that the Active File System can not.

However, we believe that the real value of the Active File System is as a file system that can allow the user to experiment with many different types of active associations, even if this generality does not provide the highest performance for any specific activity. This illustrates the trade-off between generality of use and performance of specialized functionality.

References

[1] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheima. Extending the operating system at the user level: the ufo global file system. In *Proceedings of the 1997 USENIX Technical Conference*, January 1997.

[2] Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Iinternals.* Addison-Wesley, United States of America, 1998.

[3] Rugged dataloggers for environmental, scientific and engineering measurements. http://www.campbellsci.co.uk/products/loggers.htm.

[4] Cryptstream: The next generation data security software for os/2. http://www.carbonbased.com.au/zsinfo.htm.

[5] Arman Danesh. *Mastering Linux.* Sybex Inc., United States of America, 1999.

[6] Douglas P. Ghormley, David Petrou, and Steven H. Rodriques. Slic: An extensibility system for commodity operating systems. In *Proceedings of the 1998 USENIX Technical Conference*, June 1998.

[7] Duane Hellums. *Red Hat Linux Installation and Configuration Handbook.* Que Corporation, United States of America, 1999.

[8] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. In *ACM Transactions on Computer Systems*, volume 6, pages 51–81, February 1988.

[9] Michael K. Johnson and Erik W. Troan. *Linux Application Development*. Addison-Wesley, United States of America, 1998.

[10] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[11] Howard A. Lasus. File compression: A useful technique for saving time and resources. In *ACM SIGUCCS conference XVII on User Services*, October 1989.

[12] Douglas Ludens. Enabling file auditing. http://windows2000.about.com/compute/windows2000/library/weekly/aa111499.htm?iam=ask&terms=how+can+I+find+file+systems+with+auditing

[13] Douglas Ludens. The event logs. http://windows2000.about.com/compute/windows2000/library/weekly/aa040499.htm?iam=ask&terms=how+can+I+find+file+systems+with+auditing

[14] Douglas Ludens. Windows nt compression. http://windowsnt.about.com/compute/windowsnt/library/weekly/aa120599.htm?iam=ask&terms=file+compression/decompression.

[15] C May. Mimic: A fast system/370 simulator. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, in SIGPLAN Notices*, July 1987.

[16] David Pitts and Bill Ball. *Red Hat Linux 6 Unleashed*. Sams Publishing, United States of America, 1999.

[17] Margo Seltzer and Christoper Small. A comparison of os extension technologies. In *Proceedings of the 1996 USENIX Technical Conference*, January 1997.

[18] Secure file system. http://www.cs.auckland.ac.nz/ pgut001/sfs/index.html.

[19] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice-Hall, Sebastopol, CA, 95472, States of America, 1999.

[20] Transparent cryptographic file system. http://tcfs.dia.unisa.it/.

[21] Vemco limited: Specializing in electronic systems for data logging, underwater acoustic telemetry, tracking and positioning. http://www.vemco.com/.

[22] Logging control in w3c httpd. http://www.w3.org/Daemon/User/Config/Logging.html.

[23] Zip stream: The next generation disk compression software for os/2. http://www.carbonbased.com.au/zsinfo.htm.