AutoFS - An Automounting File System for FreeBSD 6.x

Adam David Alan Martin, and Erez Zadok
Stony Brook University

Appears in the Proceedings of the 2007 BSDCan Technical BSD Conference

Abstract

File systems store, organize, and retrieve data for users and often these files are stored on remote machines, or removable media. The UNIX system requires that these file systems must be mounted before files can be accessed. In network environments, mounted file systems can result in extra traffic, even when the file system is mounted, but no files are used. This extra traffic is undesirable, and adversely affects the available network bandwidth, and mounted file systems require more in-kernel memory and datastructures to maintain them as ”active.” Nobody wants to keep remote file systems mounted, when they're not in use. AutoFS works with AMD, a daemon which auto-mounts file systems, to provide an on-demand mounting facility. The purpose of AutoFS is to limit the load upon AMD, and to provide a layer of kernel control over mounting. This control is used to minimise the number of calls to the Automounting daemon (AMD) thereby providing better performance as a user navigates the ”unified” file system tree. This paper describes the implementation details of AutoFS for FreeBSD 6.x

1 Introduction

The file system is one of the most fundamental constructs of modern computer usage to almost any class of user. Anyone from the casual Web surfer to the dedicated developer relies upon this simple, subtle, yet crucial technology to save, file, and later review his work. One's installed software is saved in various files in the file system, as are one's own valuable work. Because this technology is so ubiquitous and relied upon, improvements in performance or capability to this simple mechanism will reap great rewards for system vendors, and users alike. System vendors can market their product's improvements over the status quo as value added, and improve their competitive edge in the software systems market. Users benefit, because all great software products will have these improvements to remain competitive. However, economics aside, improvements to the file system and it's access methods are arguably the easiest way to improve any software system.

File systems store, organize, and retrieve data for users and often these files are stored on remote machines, or removable media. The UNIX system requires that these file systems must be mounted before files can be accessed. In network environments, mounted file systems can result in extra traffic, even when the file system is mounted, but no files are used. This extra traffic is undesirable, and adversely affects the available network bandwidth, and mounted file systems require more in-kernel memory and datastructures to maintain them as ”active.” Nobody wants to keep remote file systems mounted, when they're not in use. AutoFS works with AMD, a daemon which auto-mounts file systems, to provide an on-demand mounting facility. The purpose of AutoFS is to limit the load upon AMD, and to provide a layer of kernel control over mounting. This control is used to minimise the number of calls to the Automounting daemon (AMD) thereby providing better performance as a user navigates the ”unified” file system tree.

Removable media can also be serviced by the automounter. Users often expect their removable media to be available on-demand, without requiring a user-issued command to attach the file system to the unified tree. AutoFS can be used to intercept calls, for AMD, and determine whether AMD must mount a file system. AutoFS maintains an in-kernel state, to ease the burden upon AMD, and to provide for the ability to service directory lookups which do not need AMD to mount file systems. AutoFS can be used to ease the user-burden for removable media – user access patterns for removable media revolves around using a set of files from a removable media file system, then removing the media, and perhaps inserting another removable volume, to access another set of files. For example, the user will insert a CDROM, access a set of documents and images, then remove the disc, and insert another, and copy a set of binaries and music to another directory.

AutoFS must be written to have an internal timer, removing stale file systems from the ”mounted” status, should the file system be inactive. The current AutoFS protocol is also very obsolete, and has many shortcomings. I have been writing an AutoFS implementation for FreeBSD, implementing a new protocol which will be a model for other future implementations on other operating systems. This AutoFS will have a restartable, transaction based protocol. AMD, should it crash, must be able to interact with AutoFS and find out the current system state for the set of paths it monitors. This AutoFS will allow ”in-place” mounting wherein the file system can be mounted directly over a path that AutoFS monitors, without forcing AutoFS to stop the AMD process. The AMD mounter and AutoFS will communicate using an ”asynchronous” protocol, whereby AutoFS can request several mounts of AMD simultaneously, and AMD will report successes in any particular order. This will allow AMD to be rewritten or modified in a multithreaded fashion, and allow for more responsive file system activity on operations which will work with several paths at once. AutoFS will also track an internal listing of mountable and nonmountable paths, which nonmountable paths AMD should never receive notification about access, but mountable paths AMD must be notified if the path in question is not mounted.

Hopefully this will allow FreeBSD to move to the forefront of the field regarding automount features, and make the system a reference platform for the creation of future AutoFS implementations on other systems. In other words, developers will look to FreeBSD as an example for implementation of this feature, opposed to the normal situation, where FreeBSD must strive to work like other systems. It is also possible that AutoFS could be easily ported to other UNIX systems in the BSD family, with great ease.

2 Design

We designed the AutoFS Protocol with the following goals in mind:

Performance: AutoFS replaces the much slower ”NFS-server” emulation mechanism that AMD uses by default on systems which lack AutoFS. Our new AutoFS protocol allows for asynchronous ”transactions” between AMD and the kernel.

Asynchronicity: AutoFS’s protocol to userspace is explicitly asynchronous. All ”transactions” are tagged with an ID number, and are to be handled in parallel. This allows multiple file system access requests to separate directories to occur simultaneously.

Extensibility: The AutoFS protocol leaves a very large space of undefined transaction commands for future command creation. Each command structure has the size of the command packet embedded in the header. This will allow the AutoFS kernel module or user module to remove commands from the queue which either does not understand. The AutoFS protocol declares at startup time which version it will enact.

Portability: The AutoFS protocol is not tied to any system implementation or version, although our implementation is FreeBSD based. The AutoFS protocol is also independant of the communication mechanism used to transmit the protocol.

Simplicity: The AutoFS protocol was made intentionally simple. It has basic commands to support mounting, unmount notification, protocol initialization, and simple error detection. Advanced concepts, like hierarchical mounts, are implemented as userspace constructs and actions upon the simple framework provided.

AutoFS’s protocol is designed to be easy to implement from both ends, with minimal data transfer overhead. It is ”fault tolerant” in the sense that it is restartable, and all state information is stored on the ”server” or kernel side. The protocol provides commands for exchanging state information at startup, or restart. The userland program is responsible for all mount and unmount actions, and can perform these actions in any way it chooses – including, but not limited to mounting from a file system, populating a directory with a script, copying files, executing version control activities, and much more.

AutoFS implements the following commands:

Acknowledge: This command can be sent by either party (AutoFS, or an Automounter.) Its purpose is to report that one party or the other has received a message, and acted accordingly. Some command transaction sequences require an ”ACK” to complete the transaction.

Mount Request: This command is sent by the AutoFS to the Automounter to notify that a path has been accessed, and must now be made available. The path that was accessed is passed along with this command token.

Unmount Request: This command is a variant of the AutoFS mount request command. (Both use the same primary command number.) Unlike mount request, unmount request notifies the Automounter that the path argument is to be deactivated.

Mount Done: This command is sent in response to a mount request or an unmount request command, and only sent by Automounter. It notifies the AutoFS that the action on the path in question has been completed.

Hello: This command is sent by the Automounter, to AutoFS, to initialize a new session. The AutoFS communication channel should allow for only one Automounter at a time, per AutoFS instance.

Greeting: This command is sent by the AutoFS in response to the ”Hello” command. AutoFS sends some information to the Automounter in this command, including the protocol version it wishes to communicate using, and the current state of the

AutoFS's "mountpoints."

Greeting Response: This command is sent by the Automounter in response to the "Greeting" command. The command can include a list of "mountpoint" entries to add, modify, or delete. AutoFS must send back an acknowledgement for the "mountpoint" modifications requested by this command, if any.

Modify Mounts: This command will request "mountpoint" tracking modification of AutoFS. AutoFS must send back an acknowledgement for the "mountpoint" modifications requested by this command. A null list will cause a mount list report of the current state, without any changes.

Modify Mounts Acknowledge: This command sends a list of currently tracked "mountpoints" (post-modification) back to the Automounter, from the AutoFS. Any "mountpoint" modification which failed, can be implied by the lack of changes to the current state for that "mountpoint"

3 Implementation

We have implemented a prototype version of AutoFS on FreeBSD 6.x. The AutoFS is a virtual file system which can be compiled as a loadable kernel module (kld) for FreeBSD 6.x. We have tested this code on FreeBSD 6.1 and 6.2.

Unlike traditional file systems, virtual file systems have no real physical backing store. This means that, for virtual file systems, the concepts of an inode, or a superblock are quite different. Implmenting a virtual file system from scratch, or even using pre-existing stackable file system code is a more difficult task than it should be. At first, we started to implement AutoFS, based upon NullFS. After continually reaching difficulties related to allocation of memory and tracking file system state, we concluded that this common task should not need to be re-invented every time one wants to implement a virtual file system.

Luckily, FreeBSD provides a framework and prototype system for handling virtual file systems. FreeBSD ships with a number of virtual file systems already installed, including ProcFS, and LinProcFS (Linux ProcFS emulation.) Both of these file systems are implemented on top of PseudoFS, a toolkit for making virtual file systems. PseudoFS provides the programmer with a set of functions to create virtual files, destroy virtual files, and hooks to handle "events" on these files – open, close, read, write, and more. PseudoFS was not quite adequate for our needs, but solved many of the original problems inherent in building a virtual file system.

We forked PseudoFS, maintaining our own changes to it. Eventually the changeset became so vast that we decided to rename the modified version "TemplateFS." This was to avoid confusion between stock PseudoFS features, and the extensions we created. TemplateFS offers these features above and beyond PseudoFS:

- Multiple instances of any file system type
- Ability to create instances of TemplateFS clients at runtime
- Ability to destroy instances of TemplateFS clients at runtime
- Per-inode hook for vfs_lookup() function
- vfs_init() can call client code for its file system type
- Removed some unnecessary PseudoFS code for supporting ProcFS like systems.

AutoFS required the ability to hook the vfs_lookup() call. Instead of embedding our code directly into TemplateFS, we created AutoFS as a client file system of the TemplateFS code, following the PseudoFS model. This was partly because AutoFS was developed originally against PseudoFS, and partly because we felt that others might benefit from the TemplateFS work itself.

From the kernel perspective, the AutoFS code for FreeBSD has four major components. Of these, only one is really system independent.

- Device Driver: The code component which drives the virtual device for AutoFS communications
- Protocol Handler: The code component which parses the AutoFS protocol, and invokes other portions to perform the specified actions
- File system Client: The TemplateFS client code which implements the AutoFS file system semantics
- Expiry Thread: A kernel thread which runs and tracks current "mountpoints", and triggers the userland Automounter to invoke "unmount" actions.

The Automounter requires a protocol handler for this AutoFS protocol. Erez Zadok's AMDutils package provides a straightforward way of implementing this handler.

3.1 Sample Transactions

AutoFS and Automounter initialization transaction:

- The Automounter announces its presence and requests to speak a protocol version.
- The AutoFS responds with the highest version number that it can speak. It also passes a "mount"-state list to the Automounter, such that the Automounter can track "mounts" or make adjustments to this state list.
- The Automounter returns a list of modifications it wishes for this session, if anything. Otherwise it just returns an acknowledgement.
- The AutoFS returns a "Modify Mount Acknowledge" command, with a list of modified "mounts",

if any.

AutoFS requests a ”mount” of Automounter

4 Related Work

Apple’s volfs, which runs on Darwin and MacOS X, is an automounting file system of a different nature. Unlike the traditional automounter file systems, volfs automatically creates mountpoints for file systems based upon available network share resources. In this respect, it is much like an automounting system in reverse. The mounts are still requested explicitly by the user, but the creation of the mount hierarchy is automated. Apple has automounting tools for CD-ROM and other removable media formats, but these function very differently.

AM-UTILS, the automounting utilty suite, is a suite of tools including an automounter. This automounter can simulate an NFS fileserver, and allow userland emulation of an automounting file system facility. This capability, while universal, is inherently slow, due to the overhead of NFS’s RPC mechanism.

SIGMA, the Simulation Infrastructure to Guide Memory Analysis [3], represents another approach to memory profiling. This tool is used in three phases: *instrumentation*, in which SIGMA instruments all data-access instructions in the executable image of the application under test; *trace generation*, in which the instrumented code generates a compressed trace of all memory accesses; and *simulation*, in which the compressed trace is replayed against a simulator of the computer’s cache hierarchy and memory manager. This allows collection of low-level cache statistics (TLB misses, cache misses for all levels of cache, page faults, etc.) for the application running on a variety of memory systems.

The `mprof` tool concentrates on determining the cause of heap memory allocations [7]. This consists of a library which is linked into an executable in order to interpose on `malloc` and `free`. Although `mprof` requires the developer to link with a custom library, we presume that the linking could be accomplished using dynamic loader interposing as we do. The library’s implementations of the memory allocator functions record the size of the area allocated and the top five addresses from the call chain that led to this allocation, obtained by inspecting the return addresses stored in the stack. This data is post-processed and correlated with symbol information from the binary image of the application under test to produce an *allocation call graph*, in which each function is credited with its callees’ allocations.

The Valgrind suite of tools simulates program execution using dynamic binary translation [5]. It includes three memory profiling tools. Memcheck, the first of these tools, checks for most kinds of memory errors. It can pinpoint the location of buffer overruns and identify memory leaks. The second, Cachegrind, collects cache statistics, including the number of memory accesses and the number of cache misses, for each line of code. The third, Massif, profiles heap allocation; it outputs a graph that shows the amount of memory allocated over time by each line of code that makes allocations. These are by nature not real-time, but provide very high-resolution data.

Solaris’s `libumem` provides `malloc` and `free` implementations that collect data about allocated regions [1]. Developers can instrument programs with these functions when the executable is loaded, as Memcov does. The Solaris OS Modular Debugger (MDB) [6] can search for references to allocated areas in a memory dump of a running program. Those areas that are not referenced are flagged as memory leaks. The `libumem` allocation functions can also be configured to pad allocated areas with *red zones* so that MDB can detect bounds violations, and they can similarly fill recently freed areas to detect writes to those areas.

The profiler in IBM’s Rational Test RealTime instruments memory allocation and deallocation functions by directly transforming the source code immediately before compilation [2]. The instrumented functions profile allocations using techniques like those described above for `libumem`. Profiling is not real-time, but instrumented programs can be configured so that events, such as calls to a particular function, trigger the profiler to analyze the running program and report issues, including memory leaks.

Electric Fence debugs memory-access errors using memory-protection hardware [4]. However, it is not intended to profile allocations or memory accesses like Memcov and other profilers discussed here. Electric Fence’s implementation of `malloc` places a read- and write-protected page after each allocation. The protected pages trap the program as soon as it overruns or underruns a buffer, revealing exactly which instruction made the illegal read or write. Electric Fence `free` also protects pages with freed allocations in order to detect accesses to freed regions, and reports unfreed allocations when the program exits.

5 Conclusions

We have presented and demonstrated Memcov, an innovative toolkit for building memory profilers. It is *flexible*, providing insight not only into an application’s memory allocation behavior but also into its memory accesses, thus allowing the detection not only of leaks but of inefficient memory usage. It is *non-invasive*, performing large portions of its logic outside the context of the program being instrumented and requiring no additional instrumentation or analysis phases. Finally, it is *simple*, providing a straightforward and easy-to-understand interface for new profiling applications. We developed two profilers using Memcov, showing how they can be used to gain insight into how applications use memory. We also demonstrated that Memcov can be implemented with moderate overheads.

Future Work. We intend to design a leak detector based on Memcov, flagging areas that have not been accessed for a long time as potential leaks. We will reduce the penalty incurred by erroneously monitoring commonly-accessed areas by designing the detector to monitor allocations less frequently as more accesses to them are detected. This will leave accesses to those allocations uninstrumented for most of the program's execution. Memcov will use a decision process that tries to monitor rarely-accessed allocations. Profiling these allocations incurs less overhead, yet they are of greater interest, since they are more likely to be unused or leaked memory—i.e., they are more likely to represent a bug.

Additionally, we intend to explore graphical representations for allocation information that will allow programmers to quickly prioritize potential leaks and inefficient allocations by frequency of access, size, and how much of the allocation is accessed. These capabilities will complement Memcov's real-time data acquisition, making a complete tool for diagnosing inefficiencies in memory allocation and access.

6 Acknowledgments

Alfred Perlstein, Robert Watson, and Benno Rice provided valuable input at varioius stages of the design and implementation.

This work was made possible thanks to a Google Summer of Code award, in the Summer of 2006.

???? How do we change references?

References

[1] R. Benson. *Identifying Memory Management Bugs Within Applications Using the libumem Library*, June 2003. `http://access1.sun.com/techarticles/libumem.html`.

[2] J. Campbell. *Memory profiling for C/C++ with IBM Rational Test RealTime and IBM Rational PurifyPlus RealTime*, April 2004. `http://www-128.ibm.com/developerworks/rational/library/4560.html`.

[3] L. DeRose, K. Ekanadham, and J. K. Hollingsworth. SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In *Proceedings of the Supercomputing Conference 2002 (SC2002)*, Baltimore, MD, USA, November 2002.

[4] B. Perens. *efence(3)*, April 1993. `linux.die.net/man/3/efence`.

[5] J. Seward, N. Nethercote, and J. Fitzhardinge. Valgrind. `http://valgrind.kde.org`, August 2004.

[6] M. Shapiro. *Solaris Modular Debugger Guide (Solaris 8)*. Fatbrain, October 2000.

[7] B. Zorn and P. Hilfiger. A Memory Allocation Profiler for C and Lisp Programs. In *Proceedings of the Summer 1998 USENIX Conference*, Berkeley, CA, USA, June 1998.