joinFS

A Semantic Hierarchical File System

By Matthew Harlan

Senior Design 2011

B.S. Computer Science

George Washington University

School of Engineering and Applied Science

1 - Project Overview

File systems help us store and organize data on our computers. Most file systems make use of two kinds of data containers, folders and files. Files contain chunks of sequenced data, such as the series of characters that make up a text document or the images that make up a video. Folders contain files and provide a mechanism for grouping data together.

Thousands of different file systems exist with a large variety of interfaces for user interaction. The most basic file system that leverages these containers is the flat file system. A flat file system consists of a single folder that holds all of the files in the system. Each file has a unique name and users reference these names to access specific files. To accomplish data categorization in a flat file system, users can use long filenames with slashes separating each category.

Most modern file systems, such as NTFS in Windows or HFS+ in Mac OS X, extend the flat file system concept by allowing folders to also contain other folders. This interface has prevailed on the desktop PC due to the intuitive nature of the design. Humans naturally organize objects into categories and sub-categories. In hierarchical file systems, the folders represent these categories and sub-categories. Users can easily find files by navigating through their hierarchy of categories.

Hierarchical file systems make it easier for users to organize their data because users no longer have to use incredibly long filenames. Having shorter filenames also helps save storage space. However, users still have to perform all of the category organization themselves. When users get lazy, hierarchical file systems quickly turn into many loosely categorized flat file systems.

File tagging and file system search software have been added to desktop operating systems to solve the problems that lazy users create in hierarchical file systems. Spotlight, for Mac OS X, searches

through all the files in a system and provides users with a selection of categorized results. Users perform more advanced searches by refining their search terms with a set of predefined tags.

This system works reasonably well under most scenarios; however, users still have to perform these searches each time they want to find their files. Problematic scenarios also arise when users attempt to open a file they can't find from within a currently running program. In some cases, the user must perform the search, get the file's location, and then navigate to the file from within the open program. Spotlight partially alleviates this problem by allowing users to save search terms to a folder. Once again, this interface has its own limitations. Users must remember to perform another search in a different finder window, or else they will overwrite the saved search with the new one. Additionally, Spotlight restricts the user from putting search folders or permanent files within search folders.

Hierarchical File System

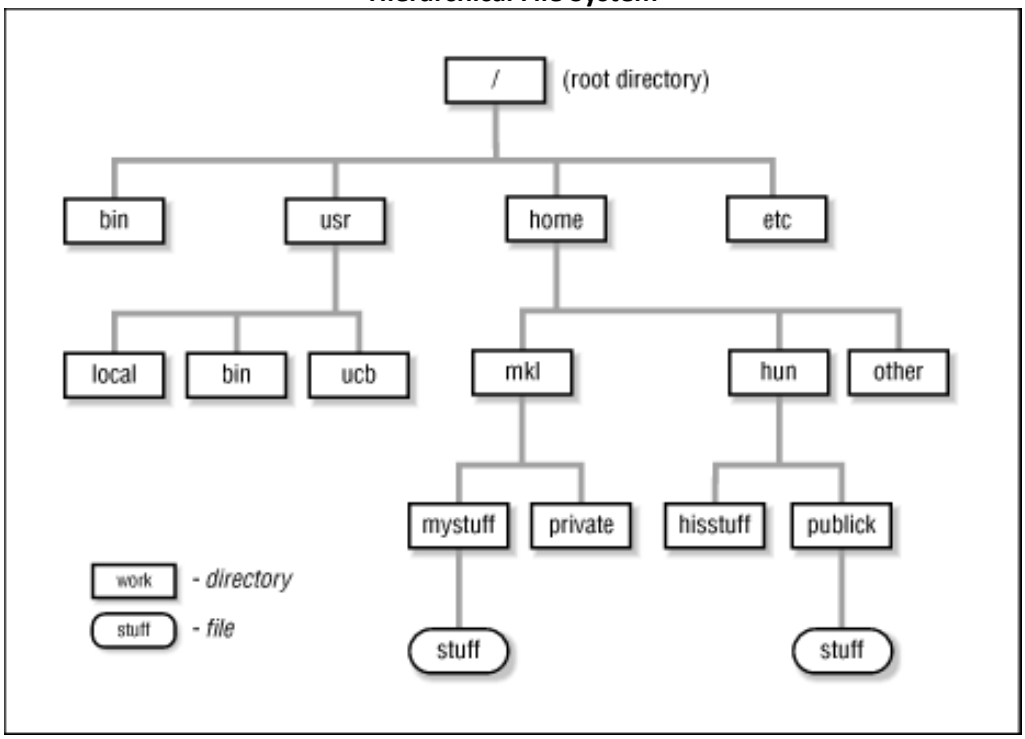


Figure 1.1

JoinFS extends these search concepts and fixes many of these issues by integrating a search system into the pre-existing file system interface. Users can tie searches to folders, which enables the contents of folders to be generated dynamically. Users can also nest searches to create more complex queries. Using these advanced folders, joinFS automatically manages your folder hierarchy and files can always be accessed under their expected category.

JoinFS can also be used as a purely hierarchical file system by users while they transition to using search folders. Distinctions between search folders and basic folders remain transparent to users. OEMs and systems administrators can setup a collection of standard search folders and users will be completely unaware that they are even running a search. Additionally, joinFS supports existing file system search software.

2 – Technical Design

Overview

High Level Diagram

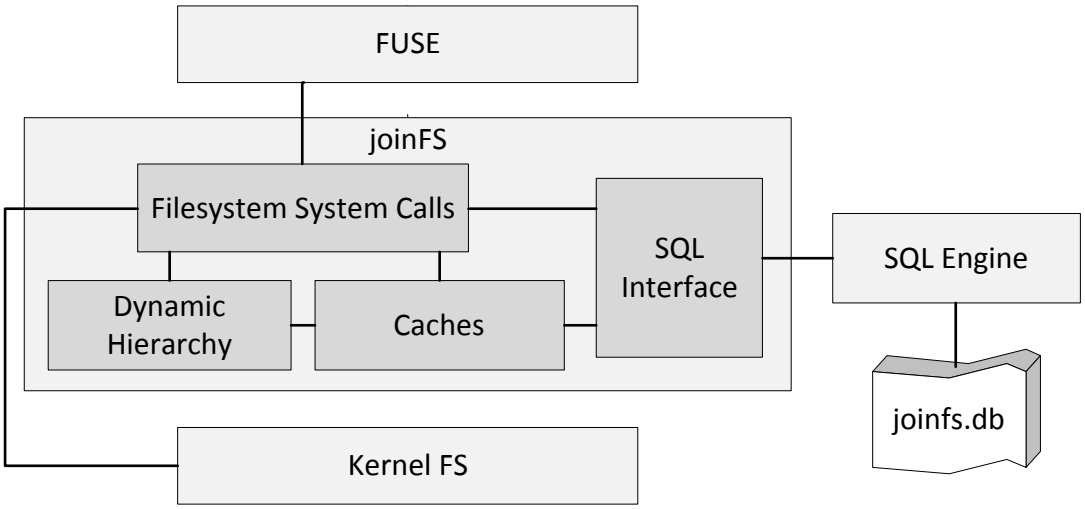


Figure 2.1

A set of core modules which interface with a set of pre-compiled libraries make up joinFS. To avoid reinventing the wheel, joinFS makes use of the SQLite library for SQL database functionality and FUSE for user-space file system functionality.

FUSE – File System in User-space

FUSE provides joinFS with a POSIX compliant user-space file system interface. FUSE mounts joinFS to a specified mount-point and intercepts all file system related system calls from the mount-point, executing joinFS system calls instead. FUSE also provides multithreading by default and enables joinFS to run as a background process. This facilitates concurrent file system access in joinFS.

In addition, FUSE provides a useful debugging interface via command line options. FUSE options can prevent process daemonization, control how much debugging information is printed, and whether or not FUSE should run as a single or multithreaded program.

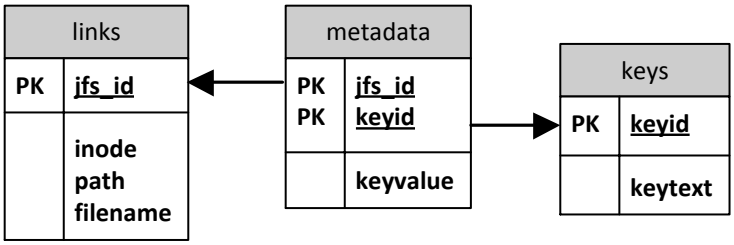
The Kernel File System

JoinFS does not handle any disk input/output. Instead, joinFS translates joinFS system calls into Linux system calls that interact with the underlying kernel file system. Any underlying kernel file system may be used to support joinFS. However, testing was performed exclusively with ext3. The kernel file system handles file security, real path lookups, link counting, basic file attributes, and all disk input/output. The kernel file system also provides caching for items that are accessed often.

SGLIB Data Structures

JoinFS makes use of the SGLIB C pre-processor based data structure library. All caches in joinFS use SGLIB hash tables. The joinFS dynamic hierarchy makes use of SGLIB lists to create a tree structure. SGLIB lists also manage database results.

The Database

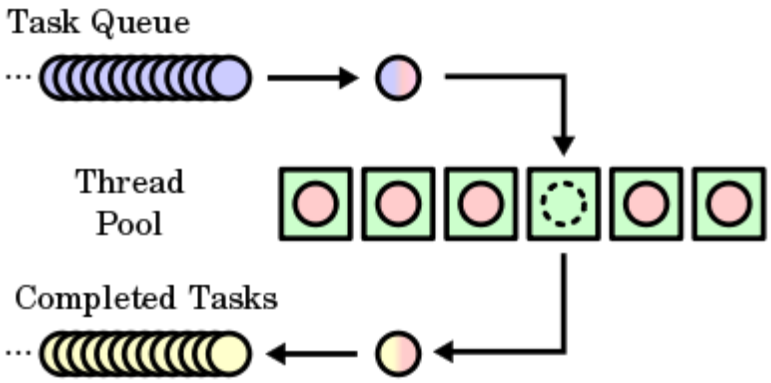


JoinFS stores file system information in a very simple database. The links table contains all file system items, the metadata table stores file system extended attributes values, and the keys table keeps track of all extended attribute names in the system.

The SQLite Interface

JoinFS maintains an index on file system items and metadata by using a SQL database. JoinFS performs all database operations using the *jfs_db_op* structure. This structure is manipulated using joinFS SQL interface functions. This API provides functions for creating queries, synchronizing processes with the multithreaded SQLite query engine that actually executes the database operations, and cleans up database operations once they finish. The API essentially acts as a layer between joinFS and the database, enabling the migration away from SQLite in the future if desired.

The SQLite Engine



In order to enable fast concurrent database reads, joinFS makes use of a thread pool. This thread pool initially starts with the same number of threads as FUSE, but can expand and shrink as necessary. The SQL interface adds all database read operations to a job queue. The read pool grabs jobs from this queue, executes them, and returns the results. The thread pool then wakes up the thread that added the job so that they can continue processing. JoinFS prevents write operations from taking place by restricting all database connections to read-only for all reader threads.

JoinFS handles writes separately because SQLite does not support concurrent write operations. JoinFS instead uses a single writer thread with its own job queue to perform all database writes. JoinFS also supports executing multiple writes at once as a transaction. This enables more complex inserts and updates without having to repeatedly attain an exclusive lock on the database file.

Database connection caching is another benefit of this design. Without the thread pool, a database connection would have to be made for each database operation. Eliminating this enables joinFS to perform queries very quickly. JoinFS also provides further SQLite optimizations by configuring SQLite via the PRAGMA command for each connection. Configuring SQLite to keep indexes in memory and to not synchronize writes drastically improved performance.

JoinFS Path Resolution and Dynamic Hierarchy

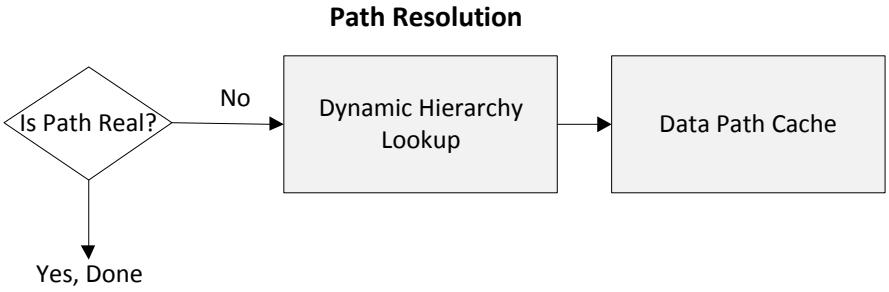


Figure 2.2

Many paths in joinFS do not actually exist in the underlying file system. To facilitate path translation, joinFS provides a function to retrieve the data or kernel file system path from a joinFS path. JoinFS also provides a mechanism for translating new paths into actual locations in the kernel file system.

To facilitate this, joinFS uses a dynamic hierarchy. The dynamic hierarchy makes use of a tree structure, each "/" separated path item corresponding to a tree node, with each folder node containing the root of a sub tree. When queries result in semantic paths joinFS adds them to the dynamic hierarchy. JoinFS also keeps the dynamic hierarchy up-to-date so that paths that are no longer viable do not remain valid.

File System Items – Files, Hard Links, Symbolic Links, and Folders

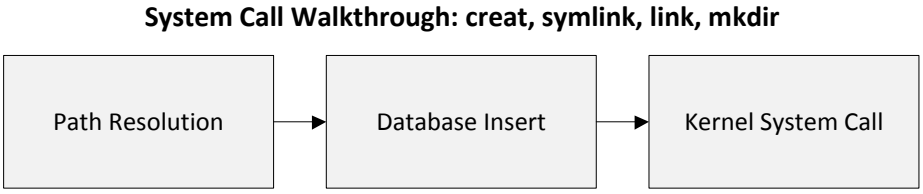


Figure 2.3

The joinFS database treats files, hard links, symbolic links, and folders the same. JoinFS adds all of them to the same table, called links. Each file system item is given its own unique joinFS ID and all database information related to a file system item can be found using this ID.

Metadata

JoinFS implements a POSIX compatible extended attribute system for file system items. Each hard link, file, symlink, or folder can have unique extended attribute name-value pairs attached to them.

JoinFS stores and indexes all extended attribute information using the SQLite database. JoinFS relates extended attributes to file system items through their joinFS ID.

Dynamic and Hybrid Folders

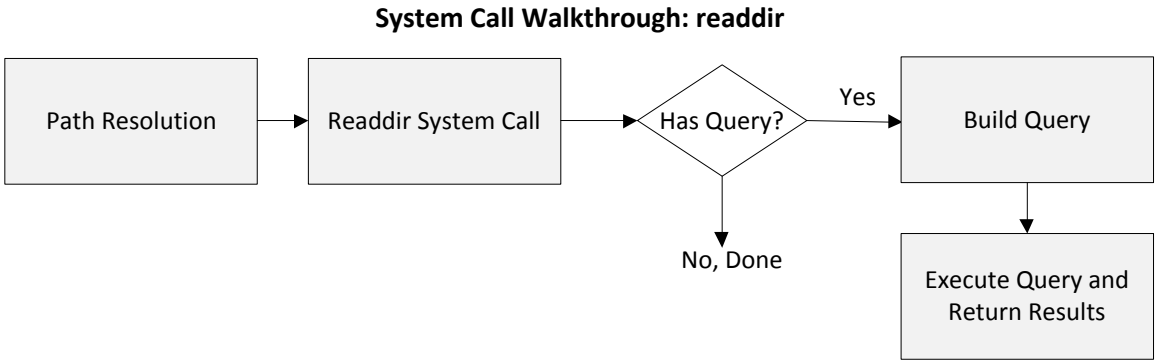


Figure 2.4

Folders in joinFS can function as semantic searches. This allows the contents of folders to dynamically change based on the current state of content in the file system. JoinFS accomplishes this by reserving a set of extended attribute names for dynamic search settings. Search results can be extended attribute values represented as folders, or file system items.

Multiple folder searches can be nested together to create complex semantic hierarchies with file searches acting as endpoints. JoinFS makes this possible by encoding search results in system paths so that the results of previous searches can be used as input to the next search. Searches enforce the search terms of parent folders to avoid search value replication at each level of the hierarchy. JoinFS also supports hybrid folders, which mix semantic and real file system items.

Data Caching

To avoid constantly hitting the database for commonly accessed items, joinFS takes advantage of a simple caching scheme. JoinFS caches extended attribute names, extended attribute values, and data paths. JoinFS only queries the database for searches or to resolve cache misses.

3 – Project Chronology

Early Development

The boot camp phase was the first phase of project development. During this phase I became familiar with the technology that I would be using for my project. I compiled and modified a file system

that logged all file system interactions to a file to become familiar with FUSE. I wrote a simple SQLite query program to get used to the SQLite3 C API. I then combined the two to create a file system that logged to a SQLite database instead of a file. I did this to ensure that I could combine the two APIs successfully and to ensure that I would have no problems building my source code in the future.

Now that I knew how to use the basics of these libraries, I began formal development of joinFS. I first built the SQLite interface. I then spent a large amount of time writing the thread pool. Thread pool development was challenging. Multi-thread code is complex and very hard to debug. There was a significant learning curve before I could effectively use the pthreads threading API. Thread pool code was modified from Sun example code to streamline development and ensure that there would be no scheduling or synchronization issues in the thread pool. To test the thread pool, I wrote a C program to spawn a variable quantity of threads and have them perform thousands of read and write database queries. I put the SQLite code through rigorous testing to make sure that all SQLite related code was finished and easy to expand or modify very early. I made this decision so that future development could focus on the more difficult file system related tasks.

Core API Development

For the 30% demo I integrated the SQLite interface into FUSE and designed and implemented the file module. My original design abstracted files out into a data file and a hard link. The data file had a unique name that was generated by a library called libuuid and all data files were stored in the same folder. The data files were linked to a hard link in the visible file system using the database. In retrospect, this was a convoluted solution and it was eventually streamlined. I also started building caches early. I knew that performance would be an issue because file systems are heavily used. By building a caching infrastructure early, it was very easy to add caches for other things throughout development.

Development became more complex for the 70% milestone. Static folders were added as well as an extended attribute interface for metadata. Security functionality was also added. A crude dynamic folder interface was designed to test the waters for the expanded capabilities that would be added in future development phases.

Queries were tied to folders outside of joinFS because there was not currently an internal interface for adding this data. Only two levels of searches, a search and a sub-search, could be performed. This was later generalized. There were many difficulties associated with building dynamic folders. It was difficult to resolve paths to dynamic file system items because they did not correspond to

their actual locations. Linking these paths was eventually accomplished at this stage of development using a hash table. This solution was not ultimately used in the final product.

For the 80% demo I implemented a dynamic folder interface that was internal to joinFS. Instead of using a separate database table, dynamic folder data was stored as metadata. Folder queries were written in SQL and inserted as the value for a reserved extended attribute name. One of the advantages of this system was that extended attributes were already being cached. This improved performance. I also spent a lot of time fixing bugs to make joinFS a more polished system.

The Final Push

The push for 90% was larger than I initially planned. I made a lot of significant changes to the dynamic folder architecture of joinFS to improve the interface. Query generation code was added to prevent directly exposing the SQL interface in joinFS and make it easier to add queries to folders. Search hierarchies were also expanded to support an unlimited depth instead of just a search and a sub-search. I also finally managed to get editing dynamic files to work. This was tricky because I had to make sure I did not lose the metadata attached to the file or else it would disappear from the dynamic location. Symbolic link and hard link code was expanded to support more functionality, but it was not bug free.

A tree-based dynamic path hierarchy was added and I improved all path resolution code as a result. The dynamic hierarchy replaced the initial dynamic path hash table solution and provides a much more memory efficient way of performing dynamic path lookups. It also helped simplify debugging.

The most intense phase of development occurred at the end. JoinFS went under a significant redesign to provide enhanced reliability and improved performance. It was my goal to be able to build the Linux kernel within joinFS by the 100% demo. Unfortunately, I did not meet this goal, but I came very close.

The unique data file indirection was removed during this development phase. The database structure was also simplified to consist of only three tables instead of four. Full hard link and symbolic link support was then added. Query generation code was updated to correctly generate SQL search code. The previous approach was found to not work correctly for larger data sets. Query generation code was re-written to produce SQL search code that performed the explicit intersection of the results of each key-value pair that a user searches for.

To improve reliability, transaction based batch writes were added to reduce how many exclusive write locks needed to be attained during each file system write operation. This was helpful because deletes would ripple through multiple tables. Large performance gains were also made by optimizing

SQLite to keep table indices in memory and not wait to synchronize each insert to disk before releasing a write lock on the database. This increased performance tremendously. JoinFS went from transferring 40,000 files in 40 minutes to performing the same transfer in 30 seconds. This makes joinFS a truly usable file system.

Retrospective

Overall, development for joinFS was hectic. A lot of changes were made towards the end and a lot more development took place between 80% and 100% than I had hoped. Though it was a lot of work, I am very satisfied with the end product. I think that it is an interesting way to think about a file system and how we access our data. JoinFS also proves that a semantic hierarchical file system is a viable file system design and performance is acceptable for everyday use. I plan to publicly release a version of joinFS soon. However, the source code for joinFS is currently publicly available online .

4 – The Future

Development of joinFS will continue. A major new feature that future versions will support is automatic file tagging. This feature will make joinFS easier to use because indexing file metadata automatically increases the quantity of available search parameters in the system. This makes joinFS a more desirable file system solution. This will be accomplished using the Linux inotify system combined with scripts to parse files for metadata. In addition, files added to dynamic folders will automatically be updated to include the extended attributes needed to satisfy the dynamic folder search parameters.

Future versions of joinFS will also support dynamic files. Dynamic files perform searches on file metadata and present the results as a text file. This feature provides advanced system profiling capabilities. This feature could also be used to generate play lists for MP3 players.

Continued development will also fine tune the joinFS code base resulting in improved reliability and performance gains in future versions. JoinFS has been released as a completely open source project under the GPLv3 copyright license. Any developer can fork and contribute updates upstream to joinFS. This encourages community development and the expansion of the core development team. It will be interesting to see what happens in the future.

5 – Sources and Figures

1.1 - http://docstore.mik.ua/orelly/unix/upt/ch01_19.htm