Hierarchical File Systems are Dead

Margo Seltzer, Nicholas Murphy
Harvard School of Engineering and Applied Sciences

Abstract

For over forty years, we have assumed hierarchical file system namespaces. These namespaces were a rudimentary attempt at simple organization. As users have begun to interact with increasing amounts of data and are increasingly demanding search capability, such a simple hierarchical model has outlasted its usefulness. For this reason, we should design file systems whose organizations map to the ways we access and manipulate data now. We present a new file system architecture in which we replace the hierarchical namespace with a tagged, search-based one.

1 Introduction

Interaction with stable storage in the modern world is generally mediated by systems that fall roughly into one of two categories: a filesystem or a database. Databases assume as much as they can about the structure of the data they store. The type of any given piece of data is known (e.g., an integer, an identifier, text, etc.), and the relationships between data are well defined. The database is the all-knowing and exclusive arbiter of access to data.

Unfortunately, if the user of the data wants more direct control over the data, a database is ill-suited. At the same time, it is unwieldy to interact directly with stable storage, so something light-weight in between a database and raw storage is needed. Filesystems have traditionally played this role. They present a simple container abstraction for data (a file) that is opaque to the system, and they allow a simple organizational structure for those containers (a hierarchical directory structure). This weak set of assumptions about the structure of data has nonetheless allowed construction of a variety of general-purpose tools to be created (e.g., ls, tar, etc.) that could operate on application data without knowing about its internals. These tools, in turn, facilitated easy construction and management of a range of applications employing stable storage, often enabling features that the applications do not include on their own (e.g., archiving in the case of the tar utility).

The situation, however, has evolved. In 1992, a "typical" disk was approximately 300 MB. In 2009, a typical disk is closer to 300 GB, representing a three order of magnitude increase. While typical file sizes have also increased, they have not increased by the same margin. As a result, users may have many gigabytes worth of photo, video, and audio libraries on a single pc. This situation represents a management nightmare, and mere hierarchical naming is ill-suited to the task. One might want to access a picture, for instance, based on who is in it, when it was taken, where it was taken, etc. Applications interacting with such libraries have evolved external tagging mechanisms to deal with this problem.

At the same time, use of the web is now ubiquitous, and "Google" is a verb. With the advent of search engines, users have learned to find data by describing what they want (e.g., various characteristics of a photo) instead of where it lives (i.e., the full pathname of the photo in the filesystem). This can be seen in the popularity of search as a modern desktop paradigm in such products as Windows Desktop Search (WDS) [26]; MacOS X Spotlight [21], which fully integrates search with the Macintosh journaled HFS+ file system [7]; and the various desktop search engines for Linux [4, 27]. Indeed, Mac OS X in particular goes one step further and exports APIs to developers allowing applications to directly access the meta-data store and content index.

It should be noted that while search is a database-like model for information retrieval, databases themselves tend to be too heavy-weight a solution. They may not be ideally optimized for a given application, and they prevent independent evolution of the data. In addition, most databases are painful to both install and manage.

Of course, along the way there have been sporadic efforts to merge the features of filesystems and databases.

The file systems community began to adopt components of database technology: journaling (logging) [1, 2, 6], transactions [5, 12, 15, 17, 18], and btrees [5, 12, 25]. Periodically, researchers suggested that we should begin thinking of our file systems more like databases. First, in 1991, Gifford et. al proposed the semantic file system, which allowed users to create virtual directories organized by attributes rather than by hierarchical names [19]. In 1993, Olson proposed implementing the file system on top of a database, providing query-based access to file system data [8]. But the hierarchical directory structure has always remained as the central filesystem abstraction. We believe it is time to finally get rid of the hierarchical namespace it in favor of a more search-friendly storage system. The hierarchical directory model is an increasingly irrelevant historical relic, and its burial is overdue.

2 The Hierarchical Namespace Albatross

But what does our aging filesystem infrastructure cost? Is it such a burden? We argue that a hierarchical namespace suffers from the following problems:

2.1 Irrelevance

Users no longer know where there files are stored. Moreover, they tend to access their data in a more search-based fashion. We encourage the skeptical reader to ask non-technical friends where their email is physically located. Can even you, the technically savvy user, produce a pathname to your personal email? Your Quicken files? The last Word document you edited? The last program you ran? For that matter, how many files are on your desktop right now? How many of them are related to each other?

2.2 Restrictiveness

There is no single, canonical categorization of a piece of data, and imposing one in the form of a full pathname conflates naming (how a piece of data is identified) with access (how to retrieve that piece of data). Much as a single piece of clothing may belong to multiple outfits, a single piece of data may belong to multiple collections. Note that we are not condemning hierarchy in general; indeed, hierarchy can be useful in a variety of situations. Rather, we are arguing against canonizing any particular hierarchy. A data item may have many names, all equally useful and even equally used.

Note, for instance, that the Unix Fast File System [13] attempts to group items logically in the same directory in the same (or a nearby) cylinder group. If those items are always accessed together, this arrangement works well, but what if the data are accessed in different ways, or access patterns evolve over time? MacCormack [11] noted, for instance, that while software architects frequently organize code into directories according to functionality, after a few years, the software's actual structure and interactions do not match the architect's initial vision. Even with appropriate clustering, Stein [22] notes that modern storage systems often violate assumptions made by existing filesystems anyway, and thus any performance gains by such clustering may be illusory to begin with. His work focused on the effect of systems like SANs, but upcoming solid-state drives, for which sequential access may no longer be fastest, will have the same effect.

2.3 Performance Limiting

In 1981, Stonebraker noted that implementing a database on top of a file system adds a superfluous level of indirection to data access [23]. This has not changed. Consider the path between a search term and a data block in most systems today. First, we look up the search term in an indexing system, which is built on top of files in the file system. Translating from search term to the file in which it is found requires traversing two indices: the search index and the physical index (i.e., file system direct/indirect blocks). That search yields a file name. We now navigate the hierarchical namespace, which is another type of index traversal, which could require multiple subsequent index traversals depending on the length of the path and the number of entries in each directory in the path. Finally, we have reached the meta-data for the file we want, and we have one last index traversal of the physical structure of that file. At a minimum, we encountered four index traversals; at a maximum, many more. Even if a system can capture all the indexes in memory, these multiple indexes place pressure on the processor caches.

Additionally, in today's multicore world, parallelism is becoming increasingly important, and a hierarchy imposes unnecessary concurrency bottlenecks. For instance, the directories `/home/nick` and `/home/margo` are functionally unrelated most of the time, yet accessing them requires synchronizing read access through a shared ancestor directory. A file system hierarchy is a simple indexing structure with obvious hotspots; better indexing structures with fewer hotspots exist, so we should take advantage of them.

We therefore posit the following basis of a modern filesystem:

- **Backwards compatibility** – With so much of the world currently built on top of hierarchical namespaces, a storage system is not useful without some support for backwards compatibility in interface if not in disk layout.

- **Separate naming from access** – The way one locates a piece of data should not be tied to how it is accessed, and multiple (potentially arbitrary) means of location should be supported.
- **Data agnostic** – Applications should be able to impose their own structure on the data items they store, and as such the filesystem should treat data items as opaque sets of bytes.
- **Direct access to data** – As the filesystem should be the thinnest possible veneer that is still useful over storage, it should allow direct access to data items. If an application knows exactly which data item it needs, it should be able to retrieve it directly.

3 hFAD: A New File System Architecture

We have developed a new file system architecture, hFAD (Hierarchical Filesystems are Dead). hFAD eschews a hierarchical namespace, instead using a tagged, search-based namespace. We present the system architecture and a brief discussion of our implementation approach.

At its lowest level, hFAD resembles an object-based storage device (OSD) [14]. Storage objects have a unique ID, and higher layers of the system access these objects by their ID. Unlike traditional OSDs, our objects are fully byte-accessible: not only can you read bytes from the object, but you can insert bytes into the middle of objects, remove bytes from the middle, etc.

Internally hFAD requires an indexing infrastructure that supports its novel, search-based API. The indexing structure contains an extensible collection of indices facilitating multiple naming modes and types of search. Figure 1 shows the hFAD architecture. In the following sections, we step through each of the major architectural components of the system and then conclude with a discussion of how we are implementing the components.

3.1 API

There are two main components to the native hFAD API. The naming interfaces map tagged search-terms to objects. The access interfaces manipulate an object, once it has been located.

3.1.1 Naming Interfaces

An object is named by one or more tag/value pairs. A tag tells hFAD how to interpret the value and in which of multiple indexes to search for the value. For example, we support POSIX naming as a thin layer atop the native API. A naming operation on POSIX path P translates into a lookup on the tag/value pair: POSIX/P. Note that a POSIX path is simply one name among many possible names. A full text search on search terms S_1, S_2, ... S_n translates into a naming operation on the vector of tag/value pairs of the form FULLTEXT/S_1, FULLTEXT/S_2, etc. As one would expect, the result of such an operation is the conjunction of the results of an index lookup for each element in the vector. Naming operations can return multiple items (which will be returned in an unspecified order). Moreover, no query need uniquely define a data item. Only the identifier for the data in the OSD layer must be unique.

3.1.2 Access Interfaces

The access interfaces support reading and writing as standard filesystems do, but due to our implementation (see 4) we can easily also support insertion and removal operations, enabling inserts into the middle of objects and truncates from anywhere in the file. To support this feature, we add an `insert` and a `truncate` call.

The `read` and `write` calls are compatible with POSIX for ease of supporting legacy applications. The `insert` call takes arguments identical to the `write` call, but instead of overwriting bytes in the middle of a file, it inserts those bytes into the appropriate position, growing the file by the number of bytes being inserted. Both `insert` and `write` can be used to append data to a file.

Truncation is similarly extended. While the POSIX truncate takes a single `off_t` with the number of bytes to truncate from the end of the file, hFAD takes two

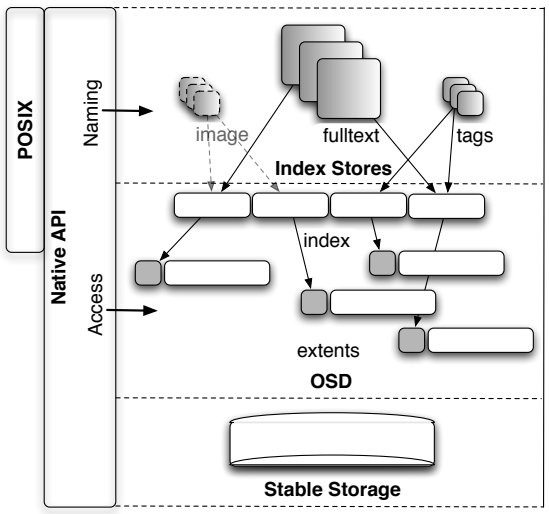

Figure 1: **hFAD Architecture** Index stores combined with arbitrary-length extents provide the primary means of accessing stable storage. A POSIX interface can easily be implemented on top of these services.

Use	Tag	Value
POSIX	POSIX	pathname
Search	FULLTEXT	term
Manual	USER	logname
	UDEF	annotations
Applications	APP	application name
	USER	logname
FastPath	ID	object identifier

Table 1: **Type/Value pairs for different API uses.** Callers into the Native API use different tags to identify different kind of values. For example, a regular keyword search specifies the `FULLTEXT` tag for all search terms. Applications tag items with the application name and the user who ran the application. Users can also add tags manually using the `UDEF` tag.

`off_t`'s, an offset and length, indicating exactly which bytes to remove from the file.

3.2 Index Stores

We specify an extensible index store to facilitate efficient search on rich data types. Given one or more type/value specifications, the collection of index stores must return a list of objects IDs matching the search terms. Just as the database community is moving away from a one-size-fits-all approach to data management [24], we believe that efficient access requires multiple indexing approaches. For example, a key/value store suffices for simple attributes, but not for full-text, and neither a full-text index nor a key/value store is likely to be suitable for image indexing.

In earlier work, we discovered that tagging indexed data provides valuable information to help in query processing [3]. Building on this experience, we have identified several types of tags we believe will expedite file search as well. Table 1 shows how different use cases will use different types of tags in their requests.

The index store layer is designed to support multiple indices. While keywords and pathnames might be easily stored in the same index, we want to leave open the possibility of extending hFAD with arbitrary index types, such as indices on images, sound, etc. A special tag, `ID`, indicates that the value is actually a unique object ID, supporting object reference caching inside applications.

3.3 OSD Layer

The storage layer is responsible for presenting the abstraction of a uniquely identified container of bytes. Each such container (object) has associated meta-data identifying the object's security attributes, its last access and modified times, and its size. The OSD layer is comparable to the ZFS Data Management Unit (DMU)[20], although the DMU provides collections of objects (objsets) as an abstraction and we do not. In ZFS, the DMU is a transactional object store; in hFAD, the OSD may be transactional, but this is an implementation decision, not a requirement.

3.4 Implementation

While we presented our architecture top-down, the implementation is easiest to understand bottom up.

We use Linux/FUSE to implement our hFAB prototype. The lowest layer of the OSD is a buddy storage allocator [9]. We've ported both Berkeley DB [16] and Lucene [10] to sit atop the raw device and the storage allocator. We allocate objects into variable sized extents. We represent objects in the OSD as Berkeley DB (BDB) btree databases whose keys are file offsets where extents begin and whose data items are the disk addresses and lengths corresponding to those offsets. As noted above, the use of btrees gives us the capability to insert and truncate with little implementation effort.

We use a NULL key value in the Btree to store the meta-data associated with an object. Indeed, POSIX metadata can easily be stored in a similar fashion as a unique key (or set of unique keys) for a file's btree. Directories also potentially map nicely onto btrees as well.

We also use BDB Btrees to map unique object IDs (OID) to the meta-data for an object and for all string indexes other than the full text search. Finally, we use Lucene for full-text search indices, and we use background threads to perform lazy full-text indexing.

Finally, the APIs are implemented as a thin layer on top of the index stores.

4 Open Questions

Obviously, the description of hFAD above is incomplete, and there are a variety of open research questions associated with this work. Among them:

- Should hFAD support arbitrary types of indexing through, for example, a plug-in model? If so, how?
- Could/should we employ ideas from the semantic filesystem work to extend the notion of a "current directory" to be an iterative refinement of a search?
- How much should the index stores do? Should they support arbitrary boolean queries? Should they include full-fledged query optimizers? How much control should they expose to filesystem clients?

5 Conclusions

We have presented a file system architecture that eschews the hierarchical namespace in favor of a tagged, search-based one. As user expectations move increasingly away from organized data to search-based data, this architecture provides a more natural way for users to manage and access data. We encourage other researchers to design their own implementations, and we look forward to the comparisons of different implementations and how well these new system work relative to historical practice.

References

[1] Jfs. http://oss.software.ibm.com/jfs.

[2] Veritas sanpoint direct file access.

[3] Layering in provenance systems. In *Proceedings of the 2009 USENIX Annual Technical Conference* (San Diego, CA, June 2009), USENIX Association.

[4] The beagle project. http://beagle-project.org/Main_Page.

[5] CHISNALL, D. Zfs uncovered. http://www.informit.com/articles/article.aspx?p=712746, April 2007.

[6] GALLI, R. Journal file systems in linux. 50–56.

[7] Hfs plus volume format. http://developer.apple.com/technotes/tn/tn1150.html.

[8] The design and implementation of the inversion file system. In *Proceedings of the 1993 USENIX Winter Conference* (1993), pp. 205–218.

[9] KNUTH, D. *The Art of Computer Programming Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA.

[10] Lucene. http://lucene.apache.org/.

[11] MACCORMACK, A., RUSNAK, J., AND BALDWIN, C. Y. The impact of component modularization on design evolution: Evidence from the software industry.

[12] MACDONALD, J. Reiser4 transaction design document. http://lwn.net/2001/1108/a/reiser4-transaction.php3.

[13] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for unix. *ACM Transactions on Computer Systems 2* (1984), 181–197.

[14] NAGLE, D., FACTOR, M. E., IREN, S., NAOR, D., RIEDEL, E., RODEH, O., AND SATRAN, J. The ansi t10 object-based storage standard and current implementation. *IBM Journal of Research and Development 52*, 4 (July 2008), 401–411.

[15] OLSON, J. Enhance your apps with file system transactions. http://msdn.microsoft.com/en-us/magazine/cc163388.aspx.

[16] OLSON, M. A., BOSTIC, K., AND SELTZER, M. I. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track* (Monterey, CA, June 1999).

[17] SCHMUCK, F., AND WYLIE, J. Experience with transactions in quicksilver. *ACM SIGOPS Operating System Review 25*, 5 (October 1991), 239 – 253.

[18] SELTZER, M., AND STONEBRAKER, M. Transaction support in read optimized and write optimized file systems. In *Proceedings of the 16th International Conference on Very Large Data Bases* (1990), Morgan Kaufmann, pp. 174–185.

[19] Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 16–25.

[20] SOLARIS. Zfs source tour. http://opensolaris.org/os/community/zfs/source/#DMU.

[21] Working with spotlight. http://developer.apple.com/macosx/spotlight.html.

[22] STEIN, L. Stupid file systems are better. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), USENIX Association, pp. 5–5.

[23] STONEBRAKER, M. Operating system support for database management. *Commun. ACM 24*, 7 (1981), 412–418.

[24] STONEBRAKER, M., AND CETINTEMEL, U. "one size fits all": An idea whose time has come and gone". In *Proceedings of the 21st International Conference on Data Engineering* (Tokyo, Japan, April 2005), IEEE Computer Society Press.

[25] SWEENEY, A., SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the xfs file system. In *In Proceedings of the 1996 USENIX Annual Technical Conference* (1996), pp. 1–14.

[26] Windows search. http://www.microsoft.com/windows/products/winfamily/desktopsearch/default.mspx.

[27] The xapian project. http://www.xapian.org/.