Getting More Information into File Names

Michael McClennen and Stuart Sechrest

Software Systems Research Laboratory
the University of Michigan

Abstract: *Hierarchical naming, while deeply embedded in our conception of file systems, is a rather weak tool for storing information about files and their relationships. A consequence is that users of today's file systems frequently have trouble locating files. We describe a system in which a standard directory tree is extended by allowing names to contain auxiliary components representing descriptive attributes rather than directory names. This system allows files to be characterized more extensively, and lets users choose among multiple organizational structures for their stored information. A prototype has been implemented by means of a new vnode layer under SunOS 4.1.3.*

1. Introduction

"Where is that file? I know I saved a copy of that announcement, six months ago. What did I name that figure? Why did I put that link there? Which version is that?" Despite the powerful capabilities of modern workstations, frustrated users ask themselves such questions many times each week. As disk size and processor speed increase, people take advantage of them to store ever more extensive collections of information. A single logical file system can easily span hundreds of hosts, and hold gigabytes of information gleaned from a worldwide network. The contents are likely to be quite heterogeneous, ranging from source code to love letters, from executables to graphics, encoded in a wide variety of formats.

In the midst of this explosion of information, our tools for organizing the contents of file systems are woefully inadequate. Most Unix users depend on utilities that are nearly 20 years old, designed for a far simpler world where file systems were small and contained little more than source code and executables. Files, especially non-textual ones, have very little associated information that could aid in retrieving them.

In fact, the very foundation of our concept of file systems, the hierarchical directory tree, puts severe limitations on the ability of users to characterize their files. Encoding descriptive information about files in long path names can hinder retrieval as much as aid it. Although people have come up with clever hacks over the years to get around these limitations, there is much to be gained by relaxing our strict reliance on hierarchical structures.

A system that provides extended facilities for characterizing files would not only solve many of the frustrating problems with misplaced data, but would provide a base on which other information management tools could be implemented in an integrated way. Examples range from configuration-management systems to WWW servers to content-based indexers for text and images.

The standard directory-tree system of file naming can, in fact, be generalized to allow much greater flexibility, while preserving its innate advantages. We can allow the path names of files to contain additional components that represent descriptive attributes rather than directories. These auxiliary components can be optionally left out when a name is used, and can commute with one other. They are available for use when needed, to search for files or to select groups of files with similar properties.

This paper describes the design and implementation of such a system under SunOS 4.1.3. We have implemented a new vnode layer to intercept name creation and resolution calls, while relying on the existing file system for storage. The necessary modifications to the kernel are quite straightforward, and should be easy to port to almost any Unix-based operating system that uses the vnode interface. The extension produces minimal changes to the application-level file system interface, so that most application programs can run without change. Tools that are aware of the new structure can provide an enriched user environment. We currently have under development a set of tools for file system browsing and manipulation, to give users full access to the expanded possibilities for data characterization and retrieval.

The next two sections of this paper discuss the problem in more detail, including historical attempts to solve it and a description of our own approach. Sections 4 and 5 describe the implementation of our prototype file system, and 6 presents our conclusions from the work so far.

2. Points of view

The problem of characterizing and organizing information stored in file systems is not new. Over the past few years, several groups of researchers have implemented systems under Unix whose purpose is to increase the amount of descriptive metadata that can be stored and made available through file names. These systems have focused on different kinds of metadata and on different situations of use. Our own goal is two-fold: to develop an abstract framework by which to understand the problem, and to build a system that focuses on providing services to humans using the file system. This paper describes our progress on the second of these goals.

2.1 Previous work

Gifford et. al. [GJS*91] provide the most general treatment of this problem up to now. Their *Semantic File System* is an experiment in the characterization of files based upon their contents. Under this system, a database is used to associate arbitrary collections of attribute/value pairs with files. The directory tree used to generate path names is augmented by “virtual directories” which list the available attributes and, beneath these, a second level listing values. Automatic filter programs scan the contents of files as they are stored, assigning values to the relevant attributes.

The basic premise of this work is the same as ours: that the file system name space can be usefully augmented to hold additional descriptive metadata. The limitations of the project come across in two ways. First, that the system is oriented heavily toward the automatic extraction of file attributes. Little or no facility is provided for users or for processes not associated with the system. For this reason, the SFS is not well suited as a base upon which to develop interactive tools. The set of available attributes is defined globally by those who write extraction tools, rather than locally by the people who store files. This system provides one particular enhancement to the standard directory tree, but does not explore the entire space of possible constructs. Further, on the implementation level, their system works strictly through a special-purpose NFS file server. This would make it cumbersome to use in many environments, and limits the extent to which it can be integrated with other kinds of file systems. Our system, by contrast, sits in the vnode layer and can be configured to work with any underlying file system.

A different approach to the problem is taken by systems that introduce *viewpathing* or *union directories* ([KK90], [Hen90], and [Pik89]). This refers to the construction of chains of directories such that files from subsequent directories "show through" to the top one. The result is to separate path names used for retrieval from those used for management. This allows the real names of files to contain more information, without complicating the view provided for retrieval. This work provides valuable insight into the ways in which filesystem semantics can be extended, and could be easily integrated with our own work.

Other examples of systems that demonstrate novel ways of organizing file metadata are the *Inversion File System* [Ols93], which by using a relational database to support a file system allows general queries on file metadata, and the *Property-List Directory System* [Mog86] which applies a global set of attribute names to a hierarchical directory structure.

2.2 Our view of the problem

The basic goal of any scheme for file naming is to facilitate two tasks: the *characterization* of files by storing descriptive information as metadata, and their *retrieval* by querying that stored information. Consider, then, the act of choosing a path name for a new file in a strictly hierarchical name space (such as a Unix directory tree). The choice of each component in the file's name indicates some property of that file that may be relevant for later retrieval. From another point of view, the name of each directory typically represents some property common to all of the files in that subtree.

In the best of all possible worlds, the set of directories in a file system would correspond 1-1 with the set of file properties that we want to use for retrieval. In that case, we could find all of the files with a given property simply by listing the appropriate directory. Unfortunately, this is often impossible for two reasons. First, you cannot map logically independent attributes to the nodes of a tree. One property must always be made superior to the other, or they cannot be allowed to overlap. As a result, such properties will either be mapped to multiple nodes or left out entirely. Second, every component in a file's name must be replayed, in one particular order, to retrieve the file. Thus, we have the situation that the better characterized a file is, the harder it is to retrieve. This is exactly the opposite of what one would want.

As an example of this phenomenon, consider archiving e-mail messages by saving them to Unix files. The most obvious technique, to classify by subject and also by author, is impossible because these attributes are independent. Even if we limit ourselves to classifying by subject, file names become either too long to be wieldy or too short to be anything but cryptic. As another example, many people have difficulty organizing files that are common to more then one project (such as figures, bibliographies, and data files). If one tries to associate these files directly with each different project to which they are relevant, one has to deal with either separate copies, multiple hard links, or symbolic links (each of which can have unpleasant side effects regarding data management and consistency). If the common files are put into a separate directory, one is then unable to easily retrieve all of the files relevant to a given project.

Historically, in the case of Unix file systems, the most popular technique used to overcome these difficulties is the ‘grep’ program, which scans text files looking for strings matching a given regular expression. This utility of this program rests on the fact that throughout most of the history of Unix the vast majority of interesting files consisted solely of ASCII text. Now that this is no longer the case, ‘grep’ is not the panacea it might once have been.

In general, there are many situations in which the actual contents of a document are not sufficient to characterize it in a manner meaningful to a particular user. For this reason, the ability to associate descriptive metadata with stored files will always be crucial to successful information organization. Although auxiliary tools will always be able to aid in locating files, the most universal means of associating descriptive information with Unix files is to keep such information in the file system itself. This can be done in two ways: either by building functionality into the operating system, as was done by [KK90], [Hen90], and [Mog86], or by providing file servers that run at the user level as did [GJS*91] and [Ols93]. We have chosen the former technique, with the aim of making efficient use of existing file systems for the actual storage of data and metadata.

3. The Multi-structured approach to information system design

Although this paper focuses on Unix file systems, the aim of our research is not simply to provide another method of storing information about files. In fact, the problems described above are not limited to file systems. Similar difficulties plague users of databases, world-wide-web sites, and other systems for the interactive storage and retrieval of information.

Each kind of information system imposes its own requirements on the ways in which stored documents (i.e., discrete pieces of information) can be arranged. In a concrete sense, different systems use different data structures to associate metadata with documents. File systems use trees, the world-wide-web uses a directed graph, and relational databases use tables.

Each of these different data structures has both positive and negative consequences for the activities of characterization and retrieval. As we have seen, the tree structure sharply limits the amount of metadata that can be associated with each document. On the other hand, it allows for convenient traversal of the attribute space and is simple for users to work with. Retrieval from a tree is accomplished by an iterative process that involves at most choosing from a limited number of options at each level.

By contrast, a relational database can be used to associate arbitrary amounts of metadata with a given collection of objects. The price is that retrieval becomes much more complicated. To retrieve objects from such a database, one must use a complicated query language. There is no easy way to traverse the attribute space, and no way at all to establish relationships among different attributes.

The crux of the matter is that no one organizational structure works well for all possible collections of information. Given a set of documents and a set of attributes describing them, there is an exponentially large number of ways of arranging these attributes into a data structure that can be used by a retrieval system. The “best” choice depends upon many variables: among others, the nature of the documents and the relationships among them, the preferences of the person who is organizing the collection, and the search strategies likely to be used by retrievers.

To revisit our earlier examples: when storing files that are described by many independent attributes, it makes sense to allow any combination of terms to be attached to a given document and to allow users to execute broad queries. Mail messages and experimental data sets, for example, could be described this way. The resulting structure is organizationally similar to a database table, with the columns or fields corresponding to the attributes available to describe files. On the other hand, when using attributes that nest, such as project-name, sub-project, module, and so on, a tree structure will likely work best. Here, the advantages of tree traversal are not offset by any difficulties in trying to represent independent attributes.

Within most large collections of files, even those stored by a single person, there will be situations in which each of these structures makes sense. It is likely that most users will wish to define some attributes that fit well in a hierarchy and some that do not. A hybrid model, in which these structures can coexist and even be mixed, will provide maximum flexibility.

Whereas conventional information system designs start with a primary data structure and attempt to apply this structure to as many situations as possible, we take the opposite tack. We believe that designers must pay much greater attention to the nature of the information to be stored and to the range of organizational strategies that users are likely to employ. If users are to have the ability to choose from more than one organizational structure, this choice must be explicitly designed into the system. This is what we call the *multi-structured approach* to information system design.

4. Design of a prototype multi-structured file system for Unix

The principal goal of our prototype system is to extend standard Unix file systems to allow the inclusion of arbitrary metadata in file names. This will allow users to pursue a variety of organizational strategies within a single file system. At the same time, we have taken care to minimize perturbation of the application-level filesystem interface. Our file system can be mounted as part of the standard working environment, and users can take advantage of its new functionality while making use of existing software.

Under our system, the names of files and directories can contain auxiliary components that do not represent Unix directories. We provide the following elements which can be used to organize stored information:

- The conventional structure of directories, links, and symbolic links remains available, and links to directories are still constrained to form a tree.
- Users are able to define additional descriptive attributes. These attributes, which we call *tags*, can be associated with links to files or directories. Any number of tags can be associated with a given link. Tags do not determine name uniqueness, so they may be omitted at will when a path name is used. They can be used to associate descriptive metadata with files and directories, and to select files based on this information.
- Users are also able to define attributes for name distinction, which we call *selectors*. Like tags, these selectors are associated with links. Unlike tags, they do determine name uniqueness. Two links are allowed to have identical names if they are associated

with different collections of selectors and neither of these collections is a subset of the other. Selectors can be used to distinguish multiple versions of the same file, or to partition a set of files into distinct classes (see Figure 1).

We meet our goal of minimal perturbation by ensuring the following:

- Any existing directory tree continues to be valid under our new system. Users are free to eschew tags and selectors when they create new links, duplicating the behavior of a conventional file system.
- The syntax of file names and the basic rules regarding absolute and relative path names, directory listings, etc. remain unchanged. Thus, Unix programs that depend on these rules will continue to work without change. The standard directory-listing routines continue to return a hierarchical view of the name space, so that programs that do recursive traversals will also continue to work. A few programs which make very picky assumptions about the structure of the name space may fail, but then again they would also do so under nearly any enhanced file system.

4.1 The name evaluation method

Under a conventional file system, a path name is evaluated by starting at an initial directory and iterating over each slash-separated component. The basic operation is as follows: given a directory and a string, look up the string in the directory and return the file system object that it points to. This object can either be a file or a directory. If it is a directory and there are components remaining in the name, it is used as the context in which to perform the next lookup.

Within our system, name evaluation is handled using exactly the same loop. The difference is that each component can identify either a link, a tag, or a selector. A directory-lookup operation returns either a file or a *working directory*. This latter can be either a real Unix directory or a *virtual directory* consisting of a real directory plus a set of tags and selectors. A virtual directory represents the subset of the links in the underlying directory that have those attributes associated with them.

By listing the contents of virtual directories, users are able to, in effect, execute conjunctive queries on the file system. Looking up a component representing a tag or selector simply adds that attribute to the working directory. This restricts the working directory to a smaller subset of links. Looking up a component representing a link returns the target of the link -- provided that that link is a member of the current working directory subset.

In the case where the link target is a directory, the result of the lookup is a working directory containing all tags from the current working directory that were not associated with the link. Thus, when a virtual directory containing a given set of tags and selectors is used as the basis for relative path names, the only valid continuations are those which contain each of those tags and selectors somewhere in the path. Just as a real directory represents a subtree, a virtual directory represents a “slice” out of a subtree consisting only of files associated with certain attributes. Symbolic links whose targets are virtual directories can be thought of as named views of a portion of the file system.

In order to increase the scope of queries that the system can respond to, we are in the process of modifying one of the standard command shells to understand a new kind of wildcard. This operator, consisting of two asterisks in a row, will match any sequence of characters including a slash. This feature, in conjunction with the use of virtual directories, will provide the ability to search entire directory trees for files associated with a given set of attributes. Other querying facilities could be added, either to the shell or via other tools.

4.2 An example

The sample interaction shown in Figure 1 illustrates some of the ways in which our system could be used. The hypothetical user has, at some point in the past, created a directory called *w95* somewhere under his home directory and has populated it with selectors, tags, and links to files.

The first item in the list shows the output of our slightly modified *ls* program, in this case manually rearranged (not displayed verbatim) in order to highlight the different categories of entries displayed. The -E option indicates that all of the attributes of each link should be listed along with the link name. The user has created two papers that share some common files, and has chosen to put them in one directory. The files associated with the attribute *journal* (which happens to be a selector) are distinguished from those associated with the attribute *usenix* (also a selector). Links which point to common files are associated with both attributes. Several tags (*text, diagram, etc.*) provide additional descriptions.

Items 2 and 3 demonstrate that the system can handle independent, intersecting attributes, showing all of the relevant links in each case. Item 4 shows part of the listing produced by a search for all files associated with the tag *diagram* anywhere under the user's home directory. Items 5, 6, and 7 demonstrate a different organizational construct; in this case, using selectors to organize the results of several experimental runs under two independent conditions. The -K option to our modified *ls* specifies that the name of a link should not be displayed unless all of its associated selectors are in the current working directory.

Note that in order to keep our prototype system reasonably simple we have chosen not to implement either typed attributes or attributes with separate values. Instead, every attribute is a simple string. As the example shows, it is possible to represent attribute values syntactically. Conventions for such representation can be developed in an analogous fashion to the historical development of rules for dot extensions in classical Unix file systems.

4.3 Using the system

The person who uses our system to create a set of file names has a lot of control over the shape of the name space (this is what makes the system multi-structured). One person might place all of the files in one directory, with their names distinguished solely by varying combinations of tags and selectors. The resulting directory would look somewhat like a relational table. Someone else might disdain the use of tags and selectors and create a standard directory tree. A third person might choose a middle ground and use some tags and selectors and a moderately sized tree of directories (but almost certainly much smaller than they would be forced to use if they were restricted to a single-structured file system). One of the goals of the project is to gather some real data regarding the preferences of various users, to guide in the subsequent development of more rigorous design specifications for multi-structured file systems.

```
1% ls -E                     <output rearranged for clarity>
journal/text/section1
journal/text/section2
journal/journal_paper.ps

journal/figure/diagram/arch.ps
journal/usenix/figure/diagram/protocol.ps
journal/usenix/figure/graph/performance.ps
usenix/figure/diagram/comparison.ps

usenix/text/section1
usenix/text/section2
usenix/text/section3
usenix/usenix_paper.ps

experiments

2% ls usenix
comparison.ps           section1                usenix_paper.ps
performance.ps          section2
protocol.pssection3

3% ls figure
arch.ps                 performance.ps          protocol.ps
comparison.ps

4% ls ~/diagram/**.ps
<as part of a longer listing>      papers/w95/comparison.ps
papers/w95/arch.ps                 papers/w95/protocol.ps

5% ls -E experiments
delay=50/caching=on/results.ps
delay=50/caching=off/results.ps
delay=60/caching=on/results.ps
delay=60/caching=off/results.ps
delay=70/caching=on/results.ps
delay=80/caching=on/results.ps

6% ls -K experiments/caching=off
delay=50                           delay=60

7% lpr experiments/caching=off/delay=50
8%
```

Figure 1: This interaction illustrates some of the ways in which a user might take advantage of a multi-structured file system to organize complex collections of files. See section 4.2 for a commentary. The *ls* program whose output is shown here includes several new options.

With our system in place, automatic tools could easily be written to add tags to files as they are created and modified. Features such as viewpathing (mentioned in section 2) would enhance the power of our system, just as they enhance the power of a conventional system. Such facilities are orthogonal to our work.

5. Implementation

Our implementation was guided by two primary constraints: to perturb the operating system and the user environment as little as possible, and to avoid unnecessary overhead. The prototype system we have built meets both goals. It is implemented in C under SunOS 4.1.3 and consists of a loadable kernel extension, a user-level server, and replacements for the standard file name manipulation commands. File attributes are stored in standard Unix directories, by means of specially encoded entries. This allows standard utilities (backups, for example) to operate on the files even when our server is not active.

The biggest portion of the kernel extension defines a new vnode and VFS type [Kle86]. Figure 2 shows the relationships between the new vnode layer and the underlying file system. Once the extension is loaded and the server is run, the directory tree which stores the raw metadata can be mounted on top of the base path under which the files will be accessed. As working-directories within the file system are visited, vnodes of the new type are created and stacked above the underlying directory vnodes.

Vnode operations that do not involve names are simply vectored through to the underlying vnodes. The only ones that are treated specially are *lookup*, *readdir*, *create*, *remove*, *link*, *rename*, *mkdir*, *rmdir*, and *symlink*. Each such operation causes an upcall to the server, which reads the underlying directory and decodes its entries. The server then computes the result of the operation and returns it to the kernel. If necessary, the kernel carries out the required modifications to the base directory. Operations that do not fit within the standard interface, such as the creation of tags and selectors, are handled by special-casing the vnode operations.

The server runs with superuser privileges, and as part of each upcall is given the credentials of the process that initiated the operation. This allows it to read and cache the contents of the underlying directories containing the file metadata. Access control for all other processes on the system continues to be handled by the kernel in the usual fashion. The use of a user-level process was based strictly on convenience during development; there is no reason why the required computations could not be carried out entirely by the kernel. Concurrency control is avoided for this prototype by providing only one server process. Requests are queued up, and satisfied one by one.

Aside from the modified kernel, our prototype system includes replacements for the basic file-manipulation programs *ls*, *mv*, *cp* and *rm*, and some special-purpose programs for manipulating tags and selectors. The replacement utilities have an expanded set of options, to take advantage of the additional features provided by the file system.

5.1 Performance and Compatibility

Although it is true that under our system some system calls necessitate two extra context-switches, this does not add appreciably to the overall latency of the system. In fact, the predominant factor continues to be the latency involved in fetching data to main memory from the disk or across the network. As shown in Table 1, the amount of overhead is fairly small even for this unoptimized, prototype system. An implementation located solely within the kernel would come close to parity with existing file systems.

Note, also, the fact that we use ordinary vnode operations to access the underlying directories. This means that we can use nearly any other file system for the actual storage of data and metadata. This includes NFS and, with slight modification to properly handle authentication tokens, AFS.

6. Conclusion and future work

Naming is a modeling process, in which the concrete relationships among the name components reflect abstract relationships among concepts. As database researchers

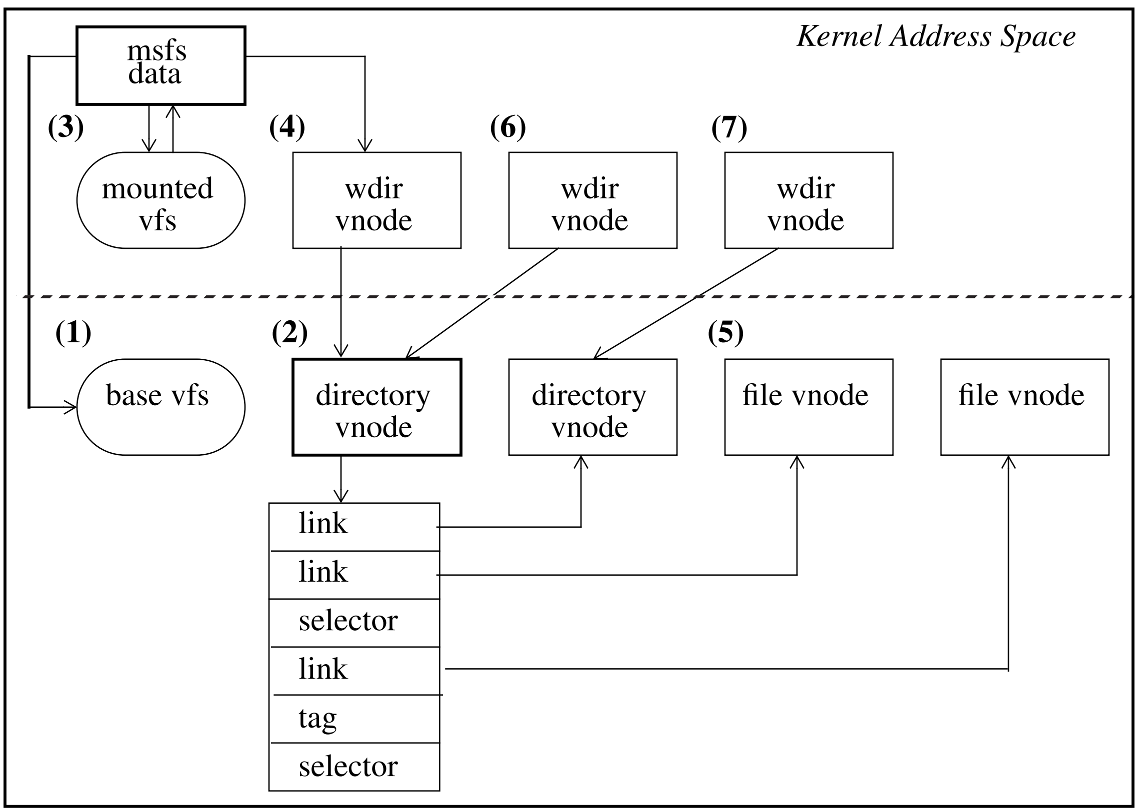

Figure 2: To mount a file tree under our prototype system, the file system that is used for underlying storage must already be mounted (1). The structures above the dotted line are created dynamically by our kernel extension, while those below belong to the underlying file system and represent objects on disk. The *mount* call causes a subtree of that file system (rooted at vnode (2)) to be mounted onto some other path name through which it will be accessed as a multi-structured file system. As a result of this call, a proxy vfs is created (3), along with a working directory vnode representing the root of the msfs (4). A lookup operation performed on that working directory causes the contents of the underlying directory (2) to be read, and may return either a file vnode (5) or a new working directory. The new wdir may point either to the same underlying directory with an additional tag or selector (6) or to a new underlying directory as the result of following a link (7).

experiment	elapsed time (UFS)	elapsed time (MSFS)	% change
cr 500	27.0	37.4	38
cr 50	2.6	2.8	7
cr 1	.05	.06	16
ls	.17	.20	18

Table 1: We performed four experiments to measure the overhead imposed by our prototype file system when layered over the standard Unix file system. The first three consist of creating and then deleting a number of files (500, 50, and 1) in a working directory containing one selector, compared against creating and deleting the same umber of files in a standard Unix directory. The fourth consists of running the standard *ls* program in a working directory whose contents comprise 50 files against a standard Unix directory containing 50 files. Each figure reports the average over ten trials. The principal cost of these operations is the extra context switches to the user-level server. An implementation contained entirely in the kernel would avoid this cost.

discovered long ago, hierarchies are a restrictive modeling tool. File system designers have been relatively untroubled by this fundamental weakness, because until recently the number of files with which a user has had to contend has been relatively small. Since most files are retrieved only by their creators users have generally been able to rely on their own memories. Yet, as file systems grow in volume and in scope, and as a much wider range of information is placed on line, system designers must pay much closer attention to the limitations of current systems and seek ways of overcoming them.

Hierarchical naming was introduced into file systems thirty years ago [DN65] and was part of the design of Unix from its inception. It is therefore somewhat surprising how easy it is to add nonhierarchical elements to the conventional file naming scheme. Most applications are dependent on only the syntactical conventions of file naming. By adopting a scheme that preserves these conventions, new schemes can be explored within the context of existing environments.

The particular naming constructs we have implemented, tags and selectors, are only two possibilities for adding more information to file names. Continuing research involves both the development of new constructs and querying capabilities and the development of new tools to use them. Such tools include both those oriented toward the presentation of information to the user (i.e., file system browsers) and those oriented toward aiding and automating the characterization of files. Organizing large collections of ad hoc data remains one of the great challenges for the online world. Moving away from strict hierarchy in file naming is a first step in this direction.

References

[GJS*91] D. Gifford, P. Jouvelot, M. Sheldon., and J. O'Toole. "Semantic File Systems". In *Proc. 13th ACM Symposium on Operating Systems Principles*, pp. 16-25, Pacific Grove CA, October 1991.

[Hen90] D. Hendricks, “A Filesystem for Software Development”, in *Proc. USENIX Summer Conference*, pp. 333-40, Anaheim CA, June 1990.

[KK90] D. Korn and E. Krell. “A New Dimension for the Unix® File System”. *Software–Practice and Experience* **20**(S1), pp. 19-34, June 1990.

[Kle86] S. Kleiman, “Vnodes: An Architecture for Multiple File System Types in Sun UNIX”, in *Proc. USENIX Summer Conference*, pp. 238-24, Atlanta GA, June 1986.

[Mog86] J. C. Mogul, “Representing Information about Files”. Technical report STAN-CS-86-1103, Stanford University, March, 1986.

[ND65] P. G. Neumann and R. C. Daley. “A General-Purpose File System for Secondary Storage”. In *AFIPS Fall Joint Computer Conference*, pp. 213-229, 1965.

[Ols93] M. Olson, “The Design and Implementation of the Inversion File System”, in *Proc. USENIX Winter Conference*, pp. 1-14, San Diego CA, January 1993.

[PPTT90] R. Pike, D. Presotto, K. Thompson, and H. Trickey. “Plan 9 from Bell Labs”. In *Proc. UK UUG, 1990*