File System Indexing, and Backup

Jerome H. Saltzer

Laboratory for Computer Science
Massachusetts Institute of Technology
M.I.T. Room NE43-513
Cambridge, Massachusetts 02139 U.S.A.

ABSTRACT

This paper briefly proposes two operating system ideas: indexing for file systems, and backup by replication rather than tape copy. Both of these ideas have been implemented in various non-operating system contexts; the proposal here is that they become operating system functions.

File System Indexing

Here is a fantasy property I would like my file system to have: it should help me find the files I am interested in. Such helpfulness is routine in library card catalogs and in online periodical search services, neither of which expect the user to know exactly what he or she is looking for. Academics have explored the idea of adding this property to the basic file system [1], and I am told that some handles that could provide such helpfulness were implemented in the Xerox Star file system [2], but apparently no one has yet seen the way clear to put an effective version of it into a file system in widespread use. After describing the idea in more detail, I offer a suggestion for a specific design that might be possible to add, after the fact, to an existing file system design such as that of UNIX, Sun's Network File System, Sprite, or the Andrew file system.

The file systems that come with current operating systems help the user find specific files only if the user knows, or can figure out, the precise path names of the needed files. That is, the file system provides exactly one method of identifying files: a hierarchical path name from a unique root. To go with this basic capability are usually a number of minor tools. For example, in UNIX the ls command produces a list of names of files in a given directory, and for the dedicated hacker who is not intimidated by arcane syntax, the find command can explore the file hierarchy searching for things. If the user can get as close as the directory that contains the file, he or she may also be able to use programs such as grep as crude content-search tools. Other systems provide similar tools, but none are integrated with the file system itself.

In the face of this paucity of support from the underlying file system, there is an alarming number of different special-purpose tools that people have developed to help find the right file in various specific contexts:

- Some bulletin board readers can search for postings of interest by content, author, or subject line.
- The Unix documentation command "man -k" searches manual page titles by keyword.
- The emacs editor command "apropos" searches information files by keyword [3].
- The emacs editor "tags" facility provides language-dependent indexes of functions, global variables, and data types in software source program files [3].
- It is becoming common to find users of engineering workstations running a background program that makes an index of all their files every night. Gifford's Intelligent File System is an example of the same idea applied to a remote file system, along with a remarkable, compatible search interface [4].
- The after-market products named Magellan and On Location provide searchable indexes of file contents for the IBM PC and the Apple Macintosh, respectively, and the NeXT computer operating system comes with a loosely integrated indexing system.
- The Melvyl system provides about 15 searchable indexes of the card catalog of the libraries of the University of California [5]. (There are currently about fifty such library card catalogs on the internet.)

These examples suggest that there is a need for facilities to find files based on things other than the simple string-with-no-blanks name that comes with most file systems. The Magellan and On Location products materialized precisely because people buying a 300 megabyte disk for their personal computer have discovered that a deep hierarchy is a good place in which to lose (track of) files.

I suspect that we haven't seen indexing features in file systems for four mutually reinforcing reasons:

1. New file system designs are not a frequent occurrence.
2. A full-bore design looks quite intimidating; designers are more comfortable working in small steps away from proven designs.
3. File systems have been designed by system programmers, who grew up with the both the concept of the naming hierarchy and the specific one they currently use, so they have gotten accustomed to keeping the top section of that naming hierarchy in their head, and don't feel a strong need for indexes.
4. Until recently, disk capacities were small and costs per bit high, providing an incentive to discard older files even though they might have some future value.

The fourth reason, disk storage cost, is history; disk storage costs have now dropped to the point where it is often cheaper to keep old files than to spend the time figuring out which ones should be discarded. Technology is thereby increasing the value of indexing.

Casual computer users don't get to take advantage of the third reason, total familiarity with the top level of the hierarchy. System hackers can get a bit of a feel for the casual user's perspective by trying to explore someone else's file system hierarchy. This experience is usually frustratingly disorienting, precisely because you don't know how things are organized at the highest levels, yet finding things requires having that high-level organization wired in to your mental model. In systems with tens or hundreds of thousands of files one needs a native guide (or special tools) to find things.

The remainder of this paper suggests a way to avoid the second reason, the intimidating design problem, by offering a small step that may be easy to add to existing designs, yet be very useful.

A Possible File System Design

Just adding the ability for a file to have key words seems pertinent, but it is not enough, because there must also be a mechanism to supply those key words, and then to allow searching on them. The supply mechanism must include defaults derived automatically from the environment, it probably includes information from the application program that called to create or write the file, and it might include an interactive query system that allows the owner to specify indexing, though that interactive system is likely to be the least used component of the scheme.

For purposes of discussion, here is an example design, probably quite far from the right target, but indicative of its direction. A high priority in this design is an extension that could be added to a current file system without being too disruptive, yet still provide a significant level of usefulness. Suppose each file were to have, in addition to its contents, three new fields, named "title", "author", and "language". (A fourth "collection" field will also be needed; but more on that in a moment.) These three fields are maintained by the file system in much the way that file systems usually maintain names, usage dates, and permissions; they are set by the file system to various defaults when the file is created (or perhaps when it is closed for writing), unless the program that created the file overrides those defaults by providing alternative values.

The default setting for "title" might be the first line of the file contents, if those contents appear to be text, otherwise empty. The default setting for "author" could be the login name of the user who created the file. The extension string of UNIX or DOS {e.g., .com .mss .txt .c .o .h .tex or .fortran} from the file name might be a suitable default for the "language" field. If the file is created by an editor such as Word, that editor could take it on itself to set the contents of these three fields to something other than the default, or at least to provide semantics for the user to edit them. The mail system might create files to store incoming messages with "title" set to the contents of the message subject line, "author" set to the value of the from-field, and "language" set to "ascii message". A compiler might set the fields in files of binary text with information it currently stores inside the binary file; "author" might be the compiler version, "title" the string "compiled from <source-file-name>". The generator of UNIX man pages might set "title" to the identifier string that is conventionally placed as the first paragraph. Scribe, LaTex, TeX, and troff formatters might fill in "title" from a user-supplied @title or .title keyword, and stuff things like "PostScript" or "DVI" in the "language" field.

Permission to modify these fields would probably be identical with permission to modify the contents of the file itself.

Whenever a file is closed, an indexing system needs to be invoked, with pointers to any modified fields, so that the new file will immediately appear visible to search requests. This need to update the indexes whenever a file is created or an indexed field is modified is one of the reasons why indexed search requires at least some cooperation of the underlying file system.

To go with the newly available indexes must be a new user interface command to report lists of files. For example, using a command-oriented interface one might have:

```
list <index1> word [...] [and <index2> word [...] ] [...]
```

where <indexi> could be "author", "titleword", "content", or "language", probably accepting two-letter abbreviations of index names as equivalent for convenience.

And more concretely, one might say

```
list tw network links and la PostScript
```

to get a list of every file in the PostScript language that has both the words "network" and "links" in its title. (For simplicity in expression, "and" in this syntax is a keyword that one cannot search for.) Whether or not "content" is an available index is an interesting question; adding it provides substantially more function but it also increases the scale of indexing effort (and implementation cleverness required) by a couple of orders of magnitude.

Does the search for candidates that meet the specification extend to every file in the system? Almost certainly that would not be what most users would expect most of the time. I probably shouldn't even be allowed to discover that there is a PostScript file that has these title words in someone else's personal directory. One needs to be able to restrict the scope of the search that is invoked by the list command, by introducing the concept of a named "collection". A collection can be an arbitrary set of files; there are some conventional collections, such as all the files belonging to a named individual, all the public files in the system, and the set of system sources. The mechanics of creating and using collections are straightforward. For each file, in addition to its contents and the title, subject, and author fields, there would be a fourth field, which holds a set of collection names. The list command would implicitly add "and collections <my collection list>" to every request. The default setting of the collection field for a file might be the collection field of the directory in which the file is created. (Whether or not the default should automatically be adjusted when a file is moved from one directory to another is an interesting question.)

One might consider replacing the four specific fields with a general property list for files, such as that found in the Lisp Machine operating system [6]. That would be a perfectly suitable implementation technique, but it covers only part of the requirement. In addition, it would be necessary that certain properties, such as title, author, language, and collection, be universal, that the file system supply default values for those properties, and that the file system provide a mechanism for updating indexes when property lists change.

Conclusion: Indexing supported by the file system seems like a winning idea. Carefully engineered defaults may turn it into a workable idea in practice. The key mechanisms are four: (1) storage of four properties with each file, (2) automatically set defaults for the four properties, (3) some mechanism to cause index updating when properties change, and (4) a search interface.

Backup by Replication

The usual operating-system-provided file storage system accomplishes backup with the aid of some kind of a daemon program that comes along periodically, identifies things that have changed recently, and makes copies onto tape of things thought to be valuable. On a less-frequent period, the backup system usually makes a complete copy of the file system contents to tape.

Three distinct reliability threats are addressed by such a backup system: disk storage failures, disk allocation errors in the file system that lead to disk tracks being incorrectly overwritten, and user mistakes in deleting files that are really still wanted. In each case, by reading the backup tapes one can retrieve copies of lost or damaged files.

This backup architecture, largely unchanged since initial designs in the early 1960's, has several problems:

1. Things that are too short-lived to be noticed don't get backed up. So some user mistakes in deleting files can't be undone.

2. Because the backup system is usually implemented as a privileged application program, rather than an integral component of the file system itself, its correct operation can depend on quite a number of things themselves working correctly. Experience suggests that this dependence makes backup fragile. Stories are legion about backup systems that didn't do what was hoped because:

- They were accidentally configured not to backup the most important files, with noone noticing.
- They were accidentally turned off, or the person responsible for running them has forgotten to take some action
- They were discovered not to have been working at all, but the users or even the system manager doesn't realize it until the backup tapes were needed.
- The wrong backup tapes were recycled.

3. The time and number of tapes required to do a complete backup of a large storage system are daunting, so most backup systems economize by using the incremental backup scheme described above. But these schemes are complex and error-prone, and require a system wizard to set them up and to check that they are operating as intended.

4. Finding things on backup tapes is often quite difficult, because the file system, not having been directly responsible for the backup, usually does not maintain any connection between the on-line file and its backup copies. The user must make a connection by recalling the approximate creation date or other properties of the file, and search backup tape maps in an attempt to identify the most recent backup copy.

There are several current projects (Coda at CMU [7], Harp at M.I.T. [8], Ficus at UCLA [9], and Echo at DEC/SRC [10]) underway exploring replication, which means that when one writes a file to a disk, the underlying file system actually writes the file to several different disks, preferably on distinct systems separated widely enough that common accidents are unlikely. However, none of the four projects intends to abandon the traditional full and incremental backup copy schemes. In each case, the replicated copies are seen as increasing data availability and perhaps reducing recovery time following certain kinds of media failure. In addition, because these systems are new and their developers are understandably wary of new software, backup is seen as essential to gaining user confidence in the new system.

When M.I.T. Project Athena distributed its system libraries across the campus, it placed two complete, read-only copies on each local subnetwork, a total of about 25 such copies. When the question of whether or not to run backup on these libraries came up, the answer was apparent: why bother? If a disk fails, after it is replaced its contents can be reconstructed by simply copying from one of the other 24, identical, servers.

Depending on replicated, on-line copies addresses the threat of disk head crash, but it provides no protection agains a user accidentally deleting a still-wanted file. Since the Athena system library was read-only, this second threat was not of concern, but in a more general application it would be.

That observation leads to the following straightforward idea for protecting users against accidental deletions or incorrect modifications: the file system "remove" operation should be implemented by copying the removed file to tape. If the file system permits files to be modified, then the file system "open for modification" operation should also copy the removed file to tape. (It would of course suffice to enqueue a copy intention and save a disk copy so as to be able to implement the intention later.)

Thus the tape becomes simply a chronological log of materials that have been changed or deleted from the system. Nothing on the tape is thought to be of great importanance (after all, to get there someone had to declare it obsolete.) Things that are deleted or modified accidentally can be retrieved, location of deleted items is strictly chronological, and recycling of old backup tapes can be based on a simple model of what the each tape contains: files deleted between two specified dates.

The idea of implementing deletion (modification) by moving the removed (changed) data to a different storage site rather than actually deleting it is not new; it is the basis for version file systems, as were used in TENEX, for backup versions as produced by editors such as gnuemacs, it is used widely in mail handling software such as the Rand Mail Handling system, and it is implemented in the Project Athena "delete" command, which renames the file and holds on to it until space is really needed. However, in none of these systems does the concept replace the traditional backup scheme. The Plan 9 distributed computing system [11] comes a little closer to implementing the spirit of the current proposal. Its backup system makes a daily snapshot of the complete file hierarchy, including full copies of any files modified that day, onto a write-once optical store.

Acknowledgement: Some of the ideas in this note came up in discussions while the author was on sabbatical assignment at the Computer Laboratory of the University of Cambridge and the Digital Equipment Corporation Systems Research Center. Particularly helpful participants in those conversations were Paul McJones, Mike Burrows, David Redell, Jeff Mogul, and Roger Needham. More recent conversations with David Gifford helped focus several of the ideas.

---

1. Jeffrey C. Mogul. *Representing information about files.* Ph.D. Thesis, Department of Computer Science, Stanford University, March 1986. Available as technical report STANFORD-CS-86-1103.

2. David Redell. Private communication.

3. Richard M. Stallman. *Gnu Emacs Manual,* Sixth Edition, Version 18. [Cambridge, Massachusetts: The Free Software Foundation: March, 1987]. Section 21.11, pp 147-151.

4. David K. Gifford, et al. "Virtual Directories for Intelligent File Systems", to be presented at the 13th ACM Symposium on Operating Systems Principles, October, 1991, Pacific Grove, California.

5. Robert K. Bradriff and Clifford A. Lynch. "The Evolution of the user interface in the MELVYL online catalog, 1980-85," in *ASIS '85: Proceedings of the 48th ASIS annual meeting 22*, Las Vegas, Nevada. [White Plains N.Y.: 1985: Knowledge Industry Publications for the American Society for Information Science.] pp 102-105.

6. Daniel Weinreb and David Moon. *Lisp Machine Manual.* Third edition. [Cambridge, Massachusetts: M.I.T. Artificial Intelligence Laboratory: March, 1981]. Section 21.9.2, pp 326-329.

7. Mahadev Satyanarayanan, et al. "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers 39*, 4 (April 1990), pp 447-459.

8. Barbara Liskov, et al. *Replication in the Harp File System.* To be presented at the 13th ACM Symposium on Operating Systems Principles, October, 1991, Pacific Grove, California.

9. Thomas W. Page, Jr., et al. *Architecture of the Ficus Scalable Replicated File System.* UCLA Computer Science Department Technical Report CSD-910005, March 1991.

10. Hisgen, Andy, et al. "Granularity and Semantic Level of Replication in the Echo Distributed File System," *Proceedings of the Workshop on Management of Replicated Data,* Houston, Texas (November 8, 1990), pp. 2-4.

11. Rob Pike, et al. "Plan 9 from Bell Labs," *Proceedings of the Summer 1990 United Kingdom Unix Users Group Conference,* July 1990, pp 1-9.