Folding a tree into a map

Victor Yodaiken

September 14, 2015

Abstract

Analysis of the retrieval architecture of the highly influential UNIX file system ([1][2]) provides insight into design methods, constraints, and possible alternatives. The basic architecture can be understood in terms of function composition and recursion by anyone with some mathematical maturity. Expertise in operating system coding or in any specialized "formal method" is not required.

1 Basics

The retrieval (read-only) operation of computer file system can be represented by a map:

$$File : Identifiers \rightarrow Contents.$$

A disk drive corresponds to a much simpler map

$$Disk : BlockNumbers \rightarrow Blocks$$

[3]

where blocks are just fixed length sequences of "bytes" (8 adjacent binary digits). Until recently, file systems had to be designed around the problem of building a reasonably sophisticated map of the first kind from the simple operation of second map. The UNIX file system ([1][2]) supports variable sized files from small text files to videos and databases and also organizes files into a tree structure so that file names describe paths through the tree. The map looks like

$$F : Paths(X) \to Contents.$$

The set $Paths(X)$ is the set of finite sequences of non-null strings over some alphabet X. Implementations use special characters as separators — e.g. $/a/b/c$ where / but fundamentally paths are sequences of strings[1].

The UNIX designers split the problem of building this system on top of a disk drive into two conceptually distinct problems. First they looked at how to get past the fixed size:

$$\alpha : Indexes \to Contents$$

where $Indexes$ is a set of numbers and $Contents$ includes, at least, all sequences of bytes within the limits of the file system. The second step is to embed the tree into this first system:

$$\beta : Paths(X) \to Indexes$$

The file system map is then given by: $F(p) = \alpha(\beta(p))$. The map β relies on a clever technique where $Contents$ is the union of the set of byte sequences (ordinary files) and the set of maps $Strings(X) \to Indexes$ (*directories*). If $\alpha(i)$ is a directory, then $\alpha(i)(x)$ applies the map that is the value of $\alpha(i)$ to the argument x.

A recursive descent from an initial index processes strings starting on the left. Let Null denote the null path (0 strings). If p is a path and x is a non-null string, then xp is the path obtained by appending x to p on the left. Then every finite path is either the null path or of the form xp.

$$\begin{aligned} \mathcal{N}(i, \text{Null}) &= i \\ \mathcal{N}(i, xp) &= \mathcal{N}(\alpha(i)(x), p) \end{aligned}$$

At each step, $\mathcal{N}$ resolves the leftmost string in the path – assuming that $\alpha(i)$ is a directory that is defined on that string.

To define β then we just need to pick some $root \in Indexes$ and then:

$$\beta(p) = \mathcal{N}(root, p)$$

[1]In practice, the length of paths and lengths of strings in the path will be constrained by some bound, but we don't have to worry about that now either.

2 Consistency and extending the model

One useful property of $\mathcal{N}$ is that it is guaranteed to terminate:

$$\text{If } p \text{ is } n \text{ elements long } \mathcal{N}(i,p) \text{ terminates in at most } n+1 \text{ steps} \tag{1}$$

This property, assures implementors that their program will not cycle infinitely trying to reduce a path to an index. A second property implicit in the definition of $\mathcal{N}$ is that if a path names a file, every prefix of that path names a directory. Let px be the path obtained by appending $x \in Strings(X)$ to $p \in Paths(X)$ on the right.

$$\text{If } F(px) \text{ is defined then } F(p) \text{ is a directory} \tag{2}$$

There are a number of additional properties that α and *root* need to satisfy to make this a reasonable file system. The most important are the "no orphans" and "no dangling references" properties.

$$\text{If } \alpha(i) \text{ is defined, there is some path } p \text{ s.t. } \mathcal{N}(root,p) = i \tag{3}$$

$$F(px) \text{ is defined if and only if } F(p)(x) \text{ is defined.} \tag{4}$$

There's another useful pair of properties for two special strings $''.''$ (self) and $''..''$ (parent).

$$\text{if } F(p) \text{ is a directory, then } F(p)(''.'') = \mathcal{N}(root,p) \tag{5}$$

$$\text{if } F(px) \text{ is a directory, then } F(px)(''..'') = \mathcal{N}(root,p) \tag{6}$$

$$F(root)(''..'') = root \tag{7}$$

Do we want to permit aliases - distinct paths that don't contain any ".." or "."strings but that name the same file? Say a path is "dot free" if it doesn't contain either ".." or "." as elements (other dots are ok). We can require that for any two dot-free paths p, q: if $\mathcal{N}(root,p) = \mathcal{N}(root,q)$ then $p = q$. The most important aspect of a property like this is the elimination of loops. Consider a program that finds all the files that are rooted by a particular folder. Let $list(F(p)) = \emptyset$ if p is not defined or is not a directory and the set of all strings $x \notin \{".",''..''\}$ where $F(p)(x)$ is defined and a directory. Then:

$$find(p) = \begin{cases} \emptyset & \text{if } F(p) \text{ is not defined} \\ \{p\} & \text{if } F(p) \text{ is a regular file} \\ & \text{or } F(p) \text{ is the empty directory} \\ \{p\} \cup (\cup_{x \in list(F(p))} find(px)) & \text{otherwise} \end{cases}$$

The question is whether $find$ is guaranteed to terminate. Even if F describes a file system with a finite number of files (always the case in practice), loops would cause $find$ to build longer and longer paths without ever hitting one of the first two terminating cases. If there are at most n files in the system, then a path of more than n elements must visit the same directory twice — implying there is an alias. So prohibiting aliases is sufficient to make $find$ and many other related algorithms work properly. The original UNIX file system did not prohibit all aliases, but had a weaker constraint that is enough to assure that $find$ terminates. Define $links$ as follows:

$$links(i,j) = \begin{cases} 1 & \text{if } \alpha(i)\text{is a directory} \\ & \text{and there is some } x \in list(\alpha(i)) \text{ s. t. } \alpha(i)(x) = j \\ 0 & \text{otherwise} \end{cases}$$

The requirement is that $\Sigma_{j \in Indexes} links(j,i) < 2$ for all i where $\alpha(i)$ is a directory plus a requirement that $links(j, root) = 1$ only if $j = root$. That is, at most one directory j links to directory i and none link to the root directory. This constraint is a consequence of the consistency properties above. Suppose that $j_1 \neq j_2$ violated the constraint - so that $\alpha(j_1)(x) = \alpha(j_2)(y) = i$ and $\alpha(i)$ is a directory. Because of property 4 there must be p and q so that $\mathcal{N}(root, p) = j_1$ and $\mathcal{N}(root, q) = j_2$. But then $F(px)(''.'') = F(qy)(''.'') = i$ and $\alpha(i)(x) = F(px)(''..'') = F(qy)(''..'')$ so $j_1 = j_2$ which contradicts the premise. If $links(i, root)$ then the same reasoning tells us $i = root$ by 7.

In later versions of UNIX things became more complex because of so called "soft links" (symbolic links). To include soft-links in the current analysis, add a third file type to ordinary files and directories: a soft link type where the contents is just a path. Then modify $\mathcal{N}$ as follows:

$$\mathcal{N}(i, xp) = \begin{cases} \mathcal{N}((\alpha(i))(x), p) & \text{if } \alpha(i) \text{ is a directory map} \\ \mathcal{N}(root, Concatenate(\alpha(i), xp)) & \text{if } \alpha(i) \text{ is a soft link} \end{cases}$$

Sadly, this new version of $\mathcal{N}$ lacks property 1. The normal method for fixing that is to count the number of soft-links visited along a path.

$$\mathcal{N}(i,p) = \mathcal{N}'(i,p,1)$$

$$\mathcal{N}'(i, xp, n) = \begin{cases} \mathcal{N}'((\alpha(i))(x), p, n) & \text{if } \alpha(i) \text{ is a directory map} \\ \mathcal{N}'(Concatenate(\alpha(i), xp), n-1) & \text{if } \alpha(i) \text{ is a soft link and } n > 0 \end{cases}$$

Soft links, however, introduce loops into the tree and $\mathcal{N}$ might visit the same directory twice.

We have to use a similar trick to make $find$ safe with some $k > 0$ as the number of acceptable soft links to traverse.

Extensions such as mounted file systems and union file systems are easy to add to this model.

3 Coding

A disk block can be considered to be just a fixed length sequence of "bytes" (8 binary digits representing numbers in the set $\{0 \ldots 255\}$). Ordinary files can be specified on the disk by a number n indicating how many bytes are in the file and a sequence of disk block numbers $b_1 \ldots b_m$. The file contents is then the result of concatenating $Disk(b_1) \ldots Disk(b_m)$ and then truncating to get n bytes of data. If the file is a directory, then this is just the first step and the second step is to decode the directory from the data. For example, if file names are composed of 16-bit unicode, then the contents of a directory file might be sequences of two byte quantities coding unicode characters, terminated by two 0 bytes, followed by, say, 4 bytes coding an index number. If "passwords" should be mapped to 34832 then hexadecimal encoding looks something like this:

$$\texttt{70}, \texttt{61}, \texttt{73}, \texttt{73}, \texttt{77}, \texttt{6F}, \texttt{72}, \texttt{64}, \texttt{73}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{88}, \texttt{10}$$

The actual coding is interesting and important, but for this paper, I just want to sketch out one method for concreteness so that

$$file : Integers \times SequenceOfBlockNumbers \to Contents$$

and

$$DecodeDirectory : Contents \to \{Strings \to Indexes\}$$

seem plausible.

The map α depends on similar encoding. To start, assume we can encode both the length n and the sequence of block numbers of a file in a single block. Let's also encode in that block a "type" that tells us if a file is ordinary, directory, or soft link. The disk drives that were the design targets of the UNIX file system could store somewhere around 2^{22} bytes of data in 2^{13} blocks. Disk drives at the time of the writing of this paper can easily hold

2^{42} or more bytes in 2^{33} blocks. In either case, a single disk block cannot hold enough disk block numbers for a really big file so some of the disk block numbers in the sequence encoded in the block are used as indirect numbers - they point to blocks that encode more numbers. The details of this are not covered here - see [**?**] for explanations.

$$
\begin{array}{l}
\alpha : Indexes \rightarrow Contents \\
DecodeSequence : Blocks \rightarrow SequencesOfBlockNumbers \\
DecodeSize : Blocks \rightarrow Integers \\
DecodeType : Blocks \rightarrow \{ordinary, directory, softlink\} \\
\alpha_1 : Indexes \rightarrow BlockNumbers \\
\alpha 2_(i) = file(DecodeSize(Disk(\alpha_1(i))), DecodeSequence(Disk(\alpha_1(i)))) \\
\alpha(i) = \begin{cases} \alpha_2(i) & \text{if } DecodeType(Disk(\alpha_1(i))) \in \{ordinary, softlink\} \\ DecodeDirectory(\alpha_2(i)) & \text{if } DecodeType(Disk(\alpha_1(i))) = \text{“}directory\text{”} \\ undefined & \text{otherwise} \end{cases} \\
file(n, \mathbf{x}) = truncate(n, Concatenate(Disk(x_1) \ldots Disk(x_n)))
\end{array}
$$

4 Discussion

The efficiency advantages of the decomposition above are reasonably obvious to anyone with an intuition about system programming but we can also make an informal complexity analysis. Think of file system maps as finite sets of pairs: in practice file systems are finite. Searching for a file in a map $Paths(X) \rightarrow Contents$ would, on average take $n/2$ steps where n is the number of pairs. This search would involve testing sequences to see if they are equal on each step. This means each step of the search involves multiple steps to compute the match. To speed up this search, we need to map paths to some sorted data rather than an unordered set. That is what the embedding does. Directory maps involve much simpler matching because we are matching strings not sequences and directories should generally be small. In practice, file systems can easily contain billions of files, but directories tend to contain just a few entries. If the average size of a directory is k elements, then an n element file system will average a depth of only $\log_k(n)$. So for a file system containing one billion ordinary files with average of 10 elements in a directory, 9 steps through the tree would resolve an average path with each step requiring comparison of a string (not a path) to an average of 5

other strings taking us to 45 string matches plus 9 lookups of directories. Compare that to 500,000,000 path matches.

Consider common queries on the file system such as "find(p)". This is efficient to compute using the tree structure. The UNIX command for a detailed list is also efficient to compute with the embedded tree structure. Detailed list involves extending out the index block(s) to contain additional information - such as last modification and security/permission data. In this case, $\mathcal{N}$ provides an index and α_1 provides the block itself. During the 1980s a number of development groups all made the same discovery that detailed list was easy to make inefficient for network file systems - because the control block for each file has to be accessed.

Of course, we could use alternative structures and it may well be that the tradeoffs have changed sufficiently since the 1970s. Maybe a detailed analysis of the kinds of file traversals and lookups common in a web site would reveal a need for a different design. Similarly, disk drives are different both in scale and operation and flash storage is common. Maybe a hash table would be more efficient. Modern implementations usually involve an in memory hash table used as a cache so that $hash(p) = i$ only if $\mathcal{N}(root, p) = i$. This cache design amortizes lookup costs.

Some of the advantages of the UNIX file-system design are purely semantic. The recursive structure ensures that there are no holes - no paths that terminate at a file that skip over inaccessible directories. This property would require additional work to guarantee in a hash-table implementation if the architects considered it important. Another set of issues becomes obvious once we consider modifications.

References

[1] D. M. Ritchie and K. Thompson, "The unix time-sharing system," *Commun. ACM*, vol. 17, no. 7, pp. 365–375, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361061

[2] R. C. Daley and P. G. Neumann, "A general-purpose file system for secondary storage," in *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, ser. AFIPS '65 (Fall, part I). New York, NY, USA: ACM, 1965, pp. 213–229. [Online]. Available: http://doi.acm.org/10.1145/1463891.1463915

[3] S. Hitsu, "Persimmon tree," in *Metropolitan Museum of Art*, 1816, http://www.metmuseum.org/collection/the-collectiononline/45392.