Searchmount

a UNIX-Based Tool

to Streamline Queries

Eeway Hsu

6.033 Design Project Proposal

Peter Szolovits TR1

February 28, 2014

1 Overview

The UNIX file system implements a highly inefficient search methodology. Currently, when a user processes redundant searches, the lengthy process of repetitively executing the `find` command ensues. There are no easy ways to save, update, or manipulate search results to thereby remove this redundancy. Here, we propose the `searchmount` tool as a solution. With `searchmount`, users can create space-efficient virtual directories containing shortcuts to essential files. These directories are automatically saved to memory and dynamically updated in response to changes in the current system and user interactions. Thus, virtual directories and its files are always accessible and up-to-date. `Searchmount` drastically increases the efficiency of system searches, and thereby eases the organization and classification of files. Below, is our design and implementation of the `searchmount` tool.

2 Searchmount Tool

The goal of `searchmount` is to simplify and extend the system querying function by storing search results. Here, results are stored in persistent virtual directories. Responsible for both the creation and removal of these virtual directories, the `searchmount` tool is implemented as a removable file system. To users, `searchmount` takes the following forms:

```
searchmount  mountpoint  searchpath  [expression]
searchmount  -u  mountpoint
```

Through the first command, a virtual directory of symbolic links, or shortcut files, is created at `mountpoint`. These shortcut files point to files from `searchpath` matching the `[expression]`. At this time, `searchmount` also attaches an instance of `inotify`, a file system listener, on `searchpath`, its subdirectories and the virtual directory. The second command removes the virtual directory and all of its dependencies. This process is illustrated in Figure 1 below.

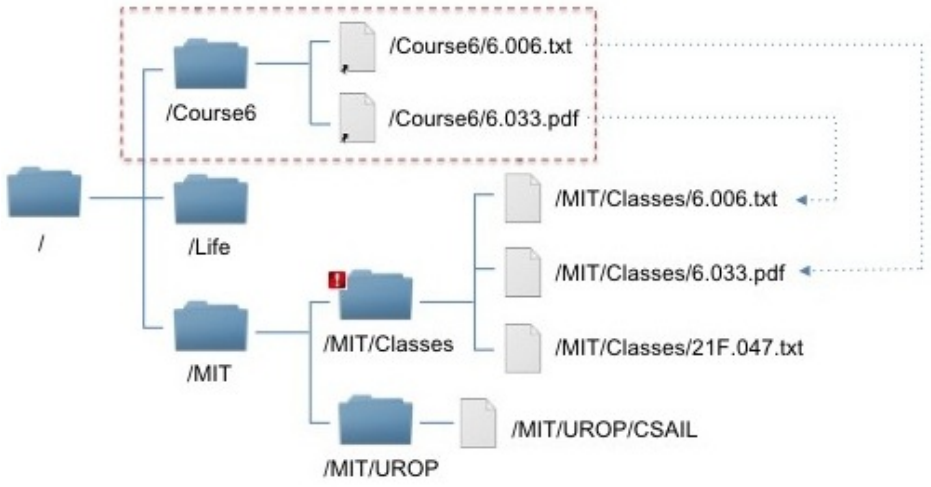

Figure 1: The result of running `searchmount /Course6 /MIT/Classes -name 6*`. for the above system is highlighted by the red box. The dotted arrows depict files in the virtual directory as symbolic links to result files.

3 Implementation

`Searchmount` is composed of four subcomponents. On the lowest level, there is the virtual directory of symbolic links. This virtual directory is stored in the database as a data structure of tables. To aid the functionality of `searchmount`, there is an API. Through the native operating system, the API can query the data structure. Next, there is a daemon process forked at `searchmount`'s initialization. Using `inotify` to monitor file system changes, the daemon ensures virtual directories are accurate upon access. Lastly, as described in Section 2, there is the user-invoked script that initializes `searchmount` to three given parameters.

3.1 Virtual Directory Data Structure

Here, we consider a data structure that effectively supports the functionality of `searchmount`. This data structure consists of three tables. The first two tables function as maps; the third table functions as a doubly-linked-list. This design ensures quick access a virtual directory and its files. The below Figure 2 illustrates the database which `searchmount` will utilize to store, find, and manipulate search results.

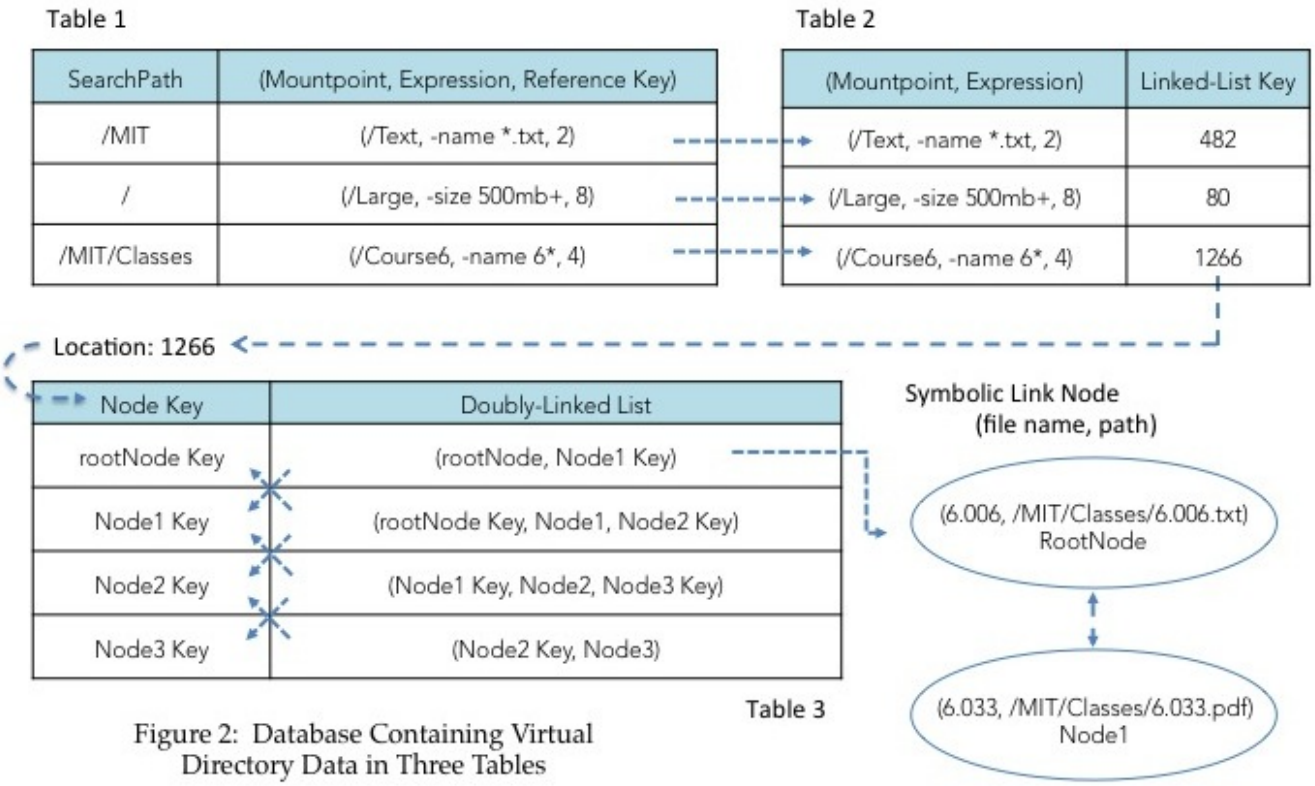

Figure 2: Database Containing Virtual Directory Data in Three Tables

As our database will be stored both in memory and on disk, we have implemented a data structure viable in both contexts. Below, we discuss the abstraction of such a data structure.

Our data structure comprises a parameter table, mountpoint table, and directory table. Using Figure 2 as a guide, we will analyze `searchmount`'s utilization of such a data structure.

3.1.1 Parameter Table

Table 1 is the parameter table. It stores the `searchpath` as a key, and (`mountpoint`, `expression`, reference key) as the value. When `searchmount` is initialized, a new entry is created. Thus, this table maintains the parameters given during initialization as a group. The reference key in the value points to a slot in Table 2.

3.1.2 Mountpoint Table

Table 2 is the mountpoint table. It stores the (`mountpoint, expression`) of all virtual directories as individual keys. The value of each key stores a pointers to the corresponding directory table. When a mountpoint is removed, the corresponding entry is deleted.

3.1.3 Directory Table

Table 3 is the directory table. It stores an ordered list of files in the virtual directory. Each value stores the (prevKey pointer, symbolic link, nextKey pointer). As shown in the lower right corner, this table can be otherwise viewed as a doubly-linked-list of symbolic links (file name, path). As this is the case, when a deletion occurs, pointers stored in the value of the previous and next entries must be reset.

Here, it is possible to have namespace collision occurrences. If resulting files from different directories have the same name, they will have the same symbolic link fileName. To resolve this issue, we attach a counter to the end of the fileName string. This resolves the issue while maintaining accuracy of the symbolic link. However, this implementation renames with a non-unique naming scheme. Depending on the order `find` returns resultant files, the same set of files can return a different set of names. As this issue is not immediate, other solutions will be considered in future implementations.

3.2 Daemon Process

There are two scenarios where the user-accessed authenticity of the virtual directories can be compromised. The first case occurs when the file system is updated. Here, virtual directories must also update, depending on their initial parameters. The second case occurs when the computer is shut down. While virtual directories are stored and loaded into memory on boot, instances of `inotify` must be reattached. In both cases, the daemon process executes the necessary actions. On boot or initialization of `searchmount`, the daemon is forked; on shut down the daemon is merged back into the parent process.

In the first case, the daemon utilizes `inotify` to maintain the file system. On initialization, the daemon attaches an instance of `inotify` to the `searchpath` directory and the virtual directory itself. It then recursively runs `inotify_add_watch` on all subdirectories of the `searchpath`. These `inotify` watches are processed and associated with individual instances of `searchmount`, and therefore individual virtual directories.

There is a remote process which reads `inotify` reports on all virtual directories. When a specific virtual directory is accessed, the remote process alerts the daemon. Here, the daemon proceeds to read `inotify` watches associated to the specific directory. If deletions or creations are detected, the modified files are tested against the search query parameter of the virtual directory. When necessary, the daemon will modify the database appropriately by adding or deleting from the data structure. If a subdirectory is added to or removed from the `searchpath` of a virtual directory, the corresponding `inotify_add_watch` or `inotify_rm_watch` will execute.

In the second case, the computer has been shut down and rebooted. While existing virtual directories and their data structures are loaded back into memory on boot, corresponding instances of `inotify` are not attached. Using the API to query for the appropriate `searchmount` of a virtual directory, the daemon, as before, reattaches necessary instances of `inotify`.

There were several trade-offs to consider in implementing a dynamic virtual directory update system. First, we considered the method of keeping one instance of `inotify` instances on all directories. Here, when any virtual directory is accessed, a read on all `inotify` watches is executed. In this case, we must read all instances of `inotify` in one call. In a large file system, this becomes tedious and slow. With our implementation on updating only an accessed file, the given the trade-off is attachment of redundant `inotify` instances. However, whenever a read is called, only a subset of the instances are read. Never are multiple watches on a single directory read simultaneously. Only in cases where `searchpath` is the root directory, do we read as many `inotify` watches as the discarded method. Thus, having virtual directories update dynamically, but not excessively allows shorter read and update times per access.

3.3 Application Programming Interface

The Application Programming Interface contains methods the operating system and outside applications can invoke. Below are five methods provided for the `searchmount` tool.

∗ In our API DirEntryPointers are nodes representing symbolic links stored in the directory table.

`1. DirEntryPointer getFirstDirectoryEntry(name):`
Get a pointer to the first entry in the directory `name` or null if there are no files in the directory.

This method can easily access and return the first directory table entry by looking for the value in the rootNode key of the directory table storing symbolic links. In Figure 2, invoking this method at `/Course6` will return the `6.006`, the symbolic link node. This is the rootNode value stored in Table 3, the `/Course6` directory.

`2. DirEntryPointer readNextDirectoryEntry(DirEntryPointer previousFile):`
Get a pointer to the next entry after `previousFile`, or null if there are no more files in the directory

Here, given a DirEntryPointer node, we can again easily access the previous DirEntryPointer node. By looking at the value of the current node, we can find the pointer to the previous node. In Figure 2, given `6.033`, we would again return `6.006`, the symbolic link node.

3. `String fileName(DirEntryPointer entry):`
Return the name of the file represented by directory entry `entry`.

A symbolic link node stores both a string of it's symbolic name and a path to the file it represents. In Figure 2, taking the DirEntryPointer node `6.006`, we will be able to extract and return `"6.006"`.

4. `String readSymbolicLink(DirEntryPointer entry):`
Return the target of a symbolic link stored in the directory entry pointed to by `entry`.

In this method, we similarly return the path from a symbolic link node. In Figure 2, taking the node `6.006`, we will be able to extract and return the path `/MIT/Classes/6.006.txt`.

5. `fileAdd(String directoryPath, String changePath):`
Adds to virtual directories depending on the original search query and the new`changePath`.

6. `fileRemove(String directoryPath, String changePath):`
Removes from virtual directories depending on the original search query and the new `changePath`.

Methods 5 and 6 are executed once the daemon has determined that a file should be modified. if a file is to be added, it is added to the end of Table 3. If a file is to be removed, the file is first found then set to null. The previous and next elements, neighbors of the current node, are access and reset. The pointers in the value must reflect the current system.

We've realized that many accessor methods are supported by linked-lists. Though in disk databases mainly support tables, we have designed the directory table to function as a doubly-linked-list. Storing the symbolic link nodes in such a table eases updating files to reflect changes in the system.

4 Analysis

The current search process implements the `find` command. `find` goes through and individually checks each existing file in the specified directory against a search query. This means, with n files in a directory, we will preform in $\Theta(n)$ time, assuming $O(1)$ per search as an over-simplification. Here we will discuss the optimizations made with the `searchmount` tool.

4.1 Seachmount Create Directory

4.1.1 Initial Searchpath Seach

Using `searchmount` for an initial search will create a new virtual directory in the in memory database. Here, a `find` is executed. Symbolic links are created to point to all matching results. The daemon attaches instances of `inotify` watches and results are shown to the user. The initial search, including virtual directory creation, runs in suboptimal time. It will be slightly slower than the standing `find`.

4.1.2 Large-scale Root Search

However, subsequent commands go from running in $\Theta(n)$ to $\Theta(1)$. One the search results are stored in the persistent data structure, the `find`, which is slow due to its individual file accesses, will not run again. The only additional time present in subsequent searches is an update time. This includes executing reads on watches of the requested virtual directory's dependencies. As we are storing the search results, there is a space complexity trade off. Now, we must store three tables. In Figure 2, Table 1 will be of size k, representing the number of `searchmount` initializations. Table 2 will be again size k. Table 3 represents one of the k tables of $O(n)$ search results. Thus, the space complexity is $\Theta(kn)$. Yet as we are storing most "virtual" data, such as symbolic links and other pointers, this memory usage is minimal. With the memory allocation of current computers, lowering the time complexity will be more significant for average users on the surface level.

4.1.3 Virtual Directory Search

While seemingly tricky, searching with a virtual directory as the `searchpath` is the same as searching a main directory. This is because given our implementation and storage of `searchmount`, virtual directories can also be used as a parameter within the `find` command and can also have instances of `inotify` attached and watching. One modification we must make is that here, we will have a resultant directory of symbolic links will point to symbolic links. The secondary pointer is redundant and should be removed in future implementations.

4.2 On Boot Behavior

The database is stored both in memory and on disk. As manipulating the persistent data structure on disk is significantly more expensive than manipulating in memory, we have optimized and reduced writes to disk. While the computer is running, the database is used and modified in the memory. On shut down, the in memory database is loaded into the database on disk; on boot, the database on disk is loaded back into memory. Thus, for each session, there is one write to disk. These writes are guaranteed to be accurate

Here, there is a scalability issue. Though we want most virtual directories to be stored on memory for quick and easy accesses, we must limit the in memory usage in database to avoid cluttering. Here we implement a method of limiting the in memory database to storing 150 virtual directories. As users can repeatedly search for the same query, rotate through their searches, or randomly choose searches, it is still unsolved as to what the best algorithm is for choosing the 150 database entries.

Here we implement a simple, "most accessed" sorting in which virtual directories can maintain an incrementing counter for whenever the remote process receives and alerts the daemon of a virtual directory `inotify` access. This is not necessary the best solution. By tracking user data and analyzing how often we must access disk, we can determine whether the algorithm for the first 150 database entries is effective. In future work, we can explore random access, most recent access and other similar efforts.

4.3 Correctness

Our implementation is dependent on the daemon. If the daemon fails to add `inotify` watches to all subdirectories, updates will not occur. If the daemon is corrupt, system errors may propagate. One possible solution alleviate such errors is to merge and re-fork the daemon on set intervals. If we reinitialize after $O(n)$, steps, we can maintain an amortized runtime of $O(1)$ despite initialization. While the daemon is running properly, we should return correct results with garbage collected.

5 Conclusion

The `searchmount` tool creates a virtual directory, executes a search query, and stores all results into the specificied virtual directory. Instead of repetitively executing the same querying, the results of a single command are stored and updated whenever necessary. Error accumulated from creating links and removing files is removed completely.

Directories created by `searchmount` are considered "virtual" because they are a collection of shortcut files. Shortcut files are pointers to, rather than copies of, existing files. Thus, while these virtual directories allows us to quickly access past search result, there is no significant increase in memory allocation. In a space efficient manner, `searchmount` automates and simplifies human input and work. The `searchmount` tool is a cost-efficient UNIX-based tool that will dramatically increase efficiency in everyday work.

While we are confident in our the current proposal, many questions remain unexplored. Our next focus is on the optimizations of data accessing and the specific addition and implementation of API methods to simplify the directory update process.

Word Count: 2490
Consult: Tiffany Chen, Jenny Shen, Ming Yang Ong