Dynamically Updating Virtual Search Folders on Unix System

Elizabeth Dethy

March 21, 2014

R06 Katrina LaCurts

1 Introduction

The searchmount smart folders feature adds a new dimension of functionality to the current file system. The smart folders feature enables users to maintain virtual directories containing soft links to files in a target search directory matching a specific query. Searchmount provides a powerful tool for a user interested in compiling and maintaining an up-to-date directory of similar files distributed throughout the file system.

This design report provides an overall system design for implementing searchmount. The primary goals are to provide an efficient tool for creating and maintaining virtual directories. This design prioritizes maintaining the accuracy and up-to-dateness of the virtual directories. Significant trade-offs to ensure this functionality included locking the target search directory and sub-directories during find, continuously processing file and directory updates, and limiting the number of directories and subdirectories a user can maintain in the virtual file system.

The main components of this design report are the data structures stored in memory and on-disk, the process of detecting a change in a virtual directory and updating accordingly, initializing a new virtual directory, the behavior of the system on restarts, and the function calls the user can make to the system. The analysis section focuses on three common use cases and the performance of the system under those conditions.

2 Design Description

Most of the data stored in the virtual file system is stored on-disk in a database. Each virtual directory has a database containing all the files matching the search query in the target searched directory. Some metadata is also stored in the database. The key in-memory data structures are two dictionaries, one which maps file descriptors to a list of mountpaths and one Inotify instances watch all target searched directories and subdirectories and alerts from inotify watches are processed by threads for the related virtual folders.

2.1 Definitions

Searched target directory: The directory specified by the search path parameter
Virtual directory: The full path to the created virtual directory. Each virtual directory has a unique mountpath and each mountpath specifies one virtual directory. Thus, the terms are used interchangeably.
inotify file descriptor: The file descriptor corresponding to the file to which a particular INOTIFY instance writes its updates.

2.2 Data Structure Specifications

The key data structure used in this design is a collection of databases that store the results of each search query. Two different formats will be used. The databases will be maintained on disk and persistent across restarts.

Additional data structures used in this design are dictionaries stored in memory that store the relationships between target searched directories and file descriptors and vice versa.

The system also implements a custom datastructure, DirEntryPointer. During initialization a dictionary is initialized and utilized for the duration of the createDirectory method and deleted before createDirectory exits.

DirEntryPointer

The DirEntryPointer acts similarly to a struct. It contains three values: the absolute file path, the file name, and the virtual directory.

Dictionaries in Memory

The three dictionaries in memory are vd_directory, fd_directory and inotify_directory. The vd_directory maps mountpaths to boolean expressions representing whether or not the particular virtual directory is mounted. The fd_directory maps target searched directories to file descriptors and the inotify_directory maps file descriptors to a list of virtual directories whose inotify instance corresponds to that file descriptor.

The INOTIFY Database

The keys in the inotify database ("ID") correspond to the file descriptors of all INOTIFY instances. Each key corresponds to a list of virtual directories with the same searched target directory.

The Virtual Directory Database

The keys in the virtual directory database (' 'VDD") correspond to the absolute paths returned by the search query. The virtual folder filename, the previous key, and the next key are stored as values. The pointers to the previous key and next key adds linked list functionality to the database. This functionality makes both user calls to the system and updating the database efficient. Updating the database is described in further detail in section 2.3. Figure 1 illustrates adding a new key to an existing database.
The first key in each VDD is 1 and its values are the searched target directory, the search expression, and the first absolute file path returned from the find command (corresponding to the first file in the virtual directory). The previous key value of the first file in the virtual directory is null. This modification is necessary to identify the first file in the virtual directory as well as maintain information on the virtual folder that will be persistent across restarts. The behavior of the system on restarts is specified further in section 2.6.

2.3 Dynamically Updating the Database

A background searchmount process handles dynamically updating the virtual directories. The process contains a poll that listens for activity on all inotify file descriptors. The poll sleeps when all file descriptors are idle. When the file descriptor changes, the poll alerts the proecess to the change. The process queries the in-memory inotify_directory and iterates through the list of virtual directories. A new thread is created to update each virtual directory database. Thread safety of the database is assumed in this implementation.

Detecting Changes to the Virtual Directory

Each thread spawned from the parent searchmount process makes a database call to VDD to retrieve the search query. The specifications for handling each inotify update are described in the section, Updating the Virtual Directory Database.

Updating the Virtual Directory Database

This system handles each inotify update uniquely. The following list describes the cirumstances under which the data structures in memory and/or on-disk are updated. The file and directory indi-

cators following IN_CREATE and IN_DELETE are not supplied from inotify but can be determined form the output of the inotify watch.

IN_ATTRIB A database update happens when the find command is run against the search query and either:

a. The file was returned and the file was not previously a key in the database

b. The file was not returned and the file was previously a key in the database

IN_CREATE file A database update happens when the find command is run and returns the file

IN_CREATE directory The find command is run recursively and all file found are added to the database

IN_DELETE file A database update occurs when the file is a key in the virtual directory database.

IN_DELETE directory A database update happens when the system non-recursively iterates through the deleted files and one or more of the files exist as keys in the VDD.

IN_DELETE_SELF Same behavior as IN_DELETE directory.

IN_MOVED_FROM Same behavior as IN_DELETE directory. This assumption can be made because if the directory or file moves to a directory still in the searchpath, the inotify watch on that directory will notify the system and the files will be re-added to the database with the new absolute file path.

On insertion, the next key value associated with the key "1" is changed to point to the new file. Likewise, the prev key value associated with the first file in the linked list (previously null) is changed to point to the new file. The new file is then put into the database with the absolute path as the key, null as the prev key value, and the previously first file in the virtual directory as the next key value. Figure 1 illustrates this process.

On deletion the pointers are changed in a similar manner. The previous key's next key value is changed to be the deleted file's next key value. The next key's prev key value is changed to be the deleted file's prev key value.

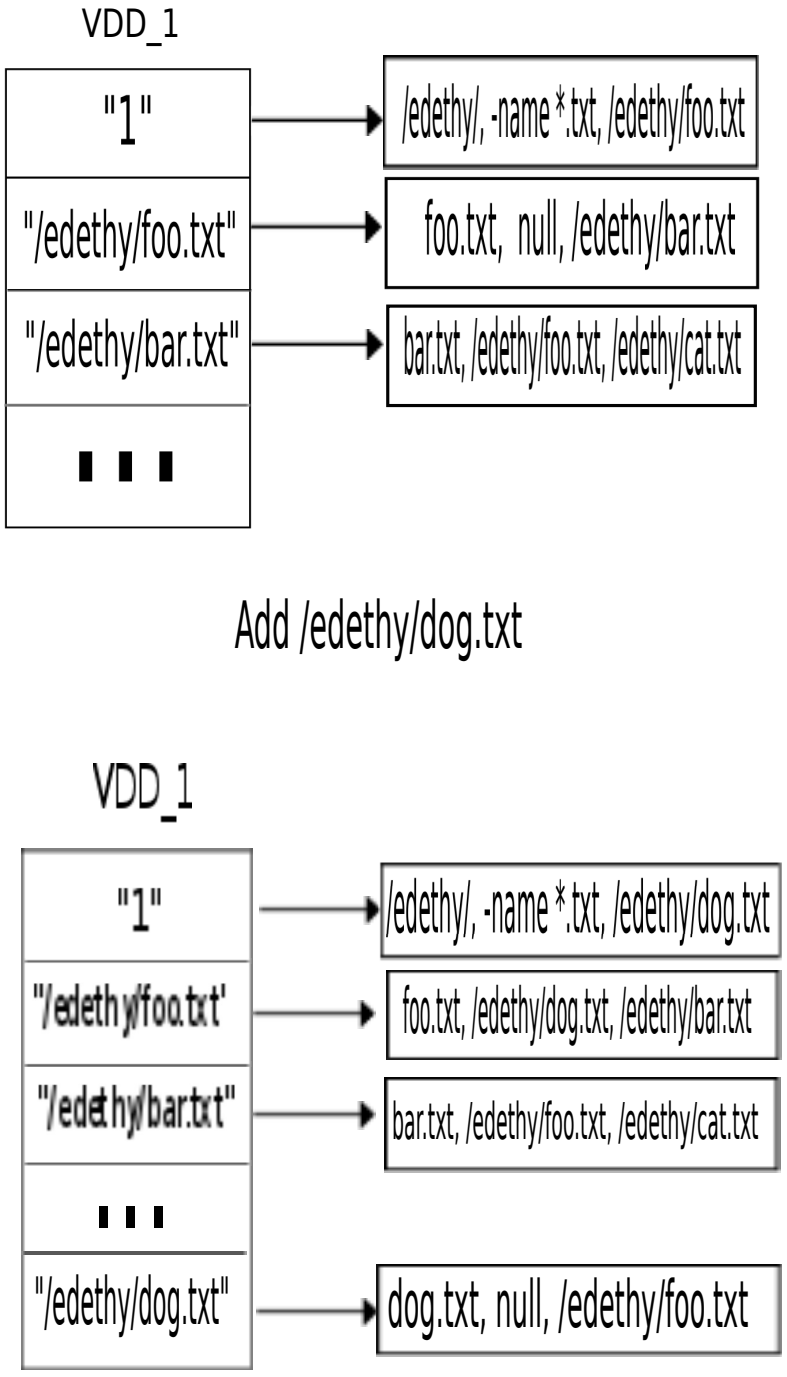

Figure 1 This figure illustrates inserting a file into a database. The pointer to the first file path associated with the key "1" updated from "/edethy/foo.txt" to "/edethy/dog.txt". The pointer to the previous file associated with key "/edethy/foo.txt" updated from ull to "/edethy/dog.txt." An important element to notice is that the new file is not inserted into the first slot in the database. The first filename refers to its position as the head of the linked list.

2.4 Mounting and Initializing A New Virtual Directory

Locking the target searched directories and subdirectories and prompting the user to modify the search path rather than increase the number of inotify watches allowed in the system are two major design trade-offs discussed in this section. Figure 2 contains the pseudocode for the createDirectory method.

```
lock target searched directories and subdirectories
if searched target directory in fd_directory.keys() then
    intofiy_directory[fd_directory[searched target directory]].append(mountpath)
else
    Initialize inotify instance
    fd_directory.put(searched target directory, file descriptor)
    inotify_directory.put(file descriptor, [mountpath])
    add watches to inotify instance. Watch for IN_CREATE, IN_ATTRIB, IN_DELETE,
IN_DELETE_SELF, IN_MOVED_FROM.
    if inotify throws too many watches exception then
        Exit and ask the user to modify the search path
    end if
end if
vd_directory.put(mountpath, true);
open a new database on-disk: open(mountpath)
put(mountpath, "1", "searched target directory, search query, first absolute file path")
Run find target_searched_directory search_query
Remove lock
Initialize new dictionary in memory, file_names
for file in file_list do
    Check if filename (most nested name, not absolute path) in file_names dictionary
    if in dictionary then
        Determine shortest distinguishing path and change slashes to underscores for both files
        Update file_names to include both new file names and disregard old one
        For old file, update database entry for new filename
        For current file add to database, put(mountpath, absolute path, "filename, previous file,
next file")
    else
        Add file name to dictionary
        Add file and metadata to database
    end if
end for
delete file_names dictionary
```

Figure 2 The pseudocode for the createDirectory method. If a new virtual directory with a new target searched path is being created, a new inotify instance is created, a new database is created, and the in-memory dictionaries are updated.

A significant trade-off in this implementation is locking the searched target directory and subdirectories during find. The locking limits other users of the file system from accessing all the files in the file system. It is, however, a necessary tradeoff. If a file is not found during the initial search, there is a high likelihood it will never be found because the search is never run again, except on reboot. If these locks are not in place, however, there is no guarantee that on reboot or any other remounting of the searchmount command all the files will be included in the virtual directory.

Prompting the user to modify the search path rather than the number of inotify watches allowed provides an intrinsic upper bound of the depth and breadth of directories the user can search. This ensures the system maintains efficiency not just of its own operations but also of the other applications running on the operating system. If the user sets the limit of inotify watches arbitrarily high, the number of watches becomes limited by the size of memory and if the number of implemented watches starts using too much memory the rest of the user's system could slow down. To avoid duplicated inotify instances and watches, the system does not instantiate multiple inotify instances for the same target searched directory. The distribution of the inotify updates to the different corresponding virtual directories is described in more detail in section 2.3.

2.5 API Calls

The system supports the following four system calls. The system calls are read-only.

getFirstDirectoryEntry

Performs two sequential queries to the corresponding VDD for the first directory entry in the virtual directory. Two sequention database queries are made. The first, to key "1" to retrieve the value of the first absolute filename. The second, to the absolute file name specified by the next key value. The absolute file name, filename, and virtual directory name are stored in a DirEntryPointer and returned to the user.

readNextDirectoryEntry

Performs two sequential queries to the corresponding VDD. The first, to retrieve the next key value from the file associate with the current DirEntryPointer object passed in as a parameter. The second, to retrieve the metadata to store in the returned DirEntryPointer object. The DirEntryPointer contains the VDD, absolute file path, and filename.

fileName

Returns the value fileName from the supplied DirEntryPointer.

readSymbolicLink

Returns the value absolute file path from the supplied DirEntryPointer.

2.6 Behavior on Restarts

On restart, the inotify database, and the virtual directory databases persist. The three dictionaries in memory also persist because on shut down the data in each of those dictionaries is stored to a corresponding database on-disk. The system iterates through the search path keys in the fd_directory database to re-instantiate all the inotify watches. The file system is locked while the inotify watches are added. The system recreates the fd_directory and inotify_directory in memory with the updated file descriptor values. The vd_directory is also re-initialized but the boolean values are set to false because the user has not mounted any directories yet. When the user chooses to mount a previously mounted vitual directory then the boolean value changes to true.

The trade-off with this approach is a performance hit on restarts while waiting for the inotify watches to be initialized. This is a one time cost, however, and enables the user to efficiently access previously mounted vitual directories without an additional delay.

3 Analysis

The analysis of this design implementation of the searchmount system will focus primarily on three use cases. For each use case the space requirements (both on-disk and in memory) of the system and the speed of the system will be analyzed. Assumptions were made regarding the size of the information stored in the VDD, the time to query memory, the time to query the database on the first call, and the time to query the database on sequential calls. Experiments were also performed to estimate the run time of a find query on file systems of various sizes and of a single file where the absolute path is known. The assumptions and results of the experiments are illustrated in Table 1 and Table 2.

Speed estimates of common operations

Operation	**Speed Estimate**
Find query on file system of $\sim$ 30000 files	1m 15s
Find query on file system of $\sim$ 100000 files	4m 30s
Find query on file system of $\sim$ 300 files	750 milliseconds
Find query on absolute file path	2 milliseconds
Retrieve value from database	$\sim$ milliseconds
Subsequent database queries	$\sim$ nanoseconds
Retrieve value from memory	$\sim$ nanoseconds

Table 1 This table shows the approximate time for various queries on disk and in memory, in addition to common operations.

Approximate space usage on-disk and in-memory

Data Structure	**Storage Estimate**
Absolute file path	50 bytes
Meta-data stored in database	50 bytes
Virtual Directory Database	100 * Number of files in database bytes

Table 2 This table shows the approximate space requirements on-disk and in memory of the data and data structures maintained by the system.

The timing analysis of the database updates is independent of the size of the database. Inserting into the the database requires two database queries and three insertions (re-inserting the previous file, the new file, and the next file) to upate the pointers. The performance hit occurs on the first read to disk, before the database is cached in memory. Each use case below describes the estimations that allowed that assumption to be made. Every subsequent database interaction occurs on the order of nanoseconds. Inserting a file into an existing database requires an upper bound of a couple of milliseconds. Deleting a file requires three reads from the database, two writes, and a deletion. Again, the first call causes the biggest performance hit and the sequential operations only take on the order of nanoseconds. Deleting a file from a database requires an upperbound of a couple milliseconds.

User uses searchmount on a target directory of a modest size ($\sim$ 300)

Assuming some files are targeted on multiple directories and some files will not be target in any virtual directories, an expected upper bound on the number of files across all the virtual directory databases will be around 400. The total size of the virtual directory databases is approximately 400*100 = 400,000 bytes or 4 MB. This is small enough to assume that after the first database call, the database will automatically be cached in memory and sequential calls will be on the order of nanoseconds rather than milliseconds.

The time to run the find command will take roughly 750 milliseconds. This is sufficiently fast for the locking of the target searched directory to not have an impact on other user's operations.

User uses searchmount on a large target directory (e.g. root $\sim$ 100000 files)

Performing a similar analysis as the first use case, the total size of the virtual directory database assuming 100000 total files stored is approximately 100000*100 = 10 MB. Most operating systems support up to 4 GB of data in main memory so the entire will be easily cached making our caching assumption above valid.

the time to run the find command will take roughly 4.5 minutes. This is a long time to lock users out of the system. However, the system aims to optimize for accuracy and the edge case of a file never being found due to being modified before the inotify watches are in place becomes a bigger issue. The penalty is a one-time penalty that solves persistent problem in the system.

An additional trade-off in this design implementation is the potential to be unable to specify the entire root directory as a search path. Although it is highly unlikely a root directory contains over 700 sub directories, if that is the case for a file system, the system will prompt the user for a more specific search path.

Users mounting search folders automatically at startup or login
The system maintains a table in memory of all created virtual directories. On startup or login, as specified in section 2.6, all created virtual directories are copied into a dictionary in main memory with boolean values corresponding to whether or not they are mounted to false. The boolean value of true indicates the user can interact with the virtual directory. On restart, the system runs a script to re-instantiate all inotify instances to keep monitoring the files in the virtual directories. When a user mounts an already created search folder at startup, the boolean value in vd_directory is changed to true and there is no additional performance hit. The behavior of the virtual directory is as if it had never been unmounted.

4 Conclusion

This design report provides a detailed implementation of the searchmount feature. Many of the design trade-offs were selected to optimize for accuracy and/or taking a one-time performance hit to create a more efficient system overall. A key concern to keep in mind is the effect on other users of locking the searched target directories and subdirectories and aiming to find a better way to handle that condition. Additionally, because the data being stored for the virtual directories isn't that large it should be possible to move the storage from a disk-backed database and into main memory. If storing the virtual directory entries in main memory does not cause performance delays in other parts of the operating system then the cost of making the first database query could go away and all accesses to the data would just be accesses to memory.

Overall the system is efficient, up-to-date, and provides the specified functionality.

5 Acknowledgements

http://linux.die.net/man/7/inotify

http://linux.die.net/man/2/poll

Word Count: 2600