Searchdir

Search Directories for Unix

Chelsea Voss (*csvoss@mit.edu)*
Arvind TR10 (R03)

6.033 Spring 2014, Design Project 1
March 21, 2014

1. Overview

Users of Windows or Apple computers can create "Search Folders:" special, dynamic folders which are automatically filled with the results of a user-defined search. Unfortunately, this functionality is not present in Unix; a Unix user who wishes to see all recently-changed files on a daily basis, for example, must repeat the same search each time. *Searchdir* solves this problem, bringing the convenience of Search Folders to the Unix operating system as "search directories."

Searchdir provides as its user interface a command-line tool to create or delete search directories. It is flexible and user-friendly, allowing highly customized search queries. Once created, a search directory is automatically filled with links to search results even as files are added or removed, with minimal impact on system performance.

This design for Searchdir keeps search results up-to-date by monitoring the file system in real-time with **`inotify`**. Although this decision comes at the expense of running many concurrent threads, it ensures that search directories will always be accurate and up-to-date, making this design tradeoff well worth it.

Additionally, the search directories are implemented as "virtual directories" [3]: this makes the implementation simpler, but comes at the expense of preventing users from interacting with the search directories in overly complex ways.

2. Design Description

In order to offer all of the necessary functionality, the Searchdir system must provide a user interface for creating new search directories, must maintain data about existing search directories, and must run in the background in order to respond to filesystem changes in real-time.

2.1 User Interface

A command-line tool called **`searchmount`** provides the user interface for managing search directories. Search directories are virtual: no actual directories are created; instead, **`searchmount`** stores its state in a database on the disk, and interfaces with Unix in order to simulate the functionality of a filesystem.

When a new search directory is created, an initial search step finds any matching files. After that, a single persistent process, called **`searchmaster`**, runs in the background and updates the search directories. It watches the filesystem for small changes, refreshing search directories without performing any redundant searches.

The user can manage their search directories by executing the **`searchmount`** command-line tool, with the following options:

» `searchmount mountpoint searchpath [expression]`

Creates a search directory at *mountpoint*, holding the results of a search for *expression* within *searchpath*.

expression allows search expressions formatted similarly to those supported by Unix's **`find`** tool, including by name, size, permissions, and more. [1]

» `searchmount -u mountpoint`

Delete the search directory at *mountpoint,* simply by removing its row from the database.

» `searchmount -master`

Resumes the **`searchmaster`** process and all **`inotify`** worker processes in case the machine is rebooted. To keep search directories properly up-to-date, this *must* be run *immediately* after reboot.

2.2 Data Structures

Search directories are stored in three simple key-value databases on the system's disk, at ***/etc/searchdir.db***, ***/etc/mountpoints.db***, and ***/etc/results.db***. Using a database offers advantages in quick lookup time and reliable data storage.

mountpoints.db contains a list of mountpoints where search directories are located, enabling an algorithm to iteratively access all search directories by name.

searchdir.db connects each mountpoint to data about its search directory: the search path and query expression of the original search query, and a list of search results. Each row of ***searchdir.db*** and ***mountpoints.db*** corresponds to a single search directory.

Finally, ***results.db*** contains the lists of results, one list for each search directory's contents. Each row of ***results.db*** represents a single search result, as a symbolic link within its search directory.

The technical specification of these databases is depicted in **Figure 1** (next page), and an example of how the database might be used to look up a single search result is depicted in **Figure 2** (next page).

The database is stored on-disk, but it is also cached in memory so that recent key-value pairs can be accessed quickly. Overall, the design of this database allows for multiple search directories to be concurrently maintained by a single master thread.

List of Mountpoints (mountpoints.db)

Key		*Value*
`int index`	⟶	`string` **`Mountpoint`**

Search Directory Data (searchdir.db)

Key		*Value*

`string` **`Mountpoint`**	⟶	*`object SearchDirectory:`*
		`string Searchpath`
		`string Expression`
		`int` **`ResultsListID`**

Lists of Results (results.db)

Key		*Value*

`[int` **`ResultsListID`**`,`
`int index]`
	⟶	*`object Result:`*
		`string Name`
		`string Link`

Figure 1: *Searchdir uses three simple key-value databases to store the state of search directories. Multiple pieces of related data can be stored within the same object. Certain values (bolded) may instead act as keys to other databases, bridges that connect multiple database entries together.*

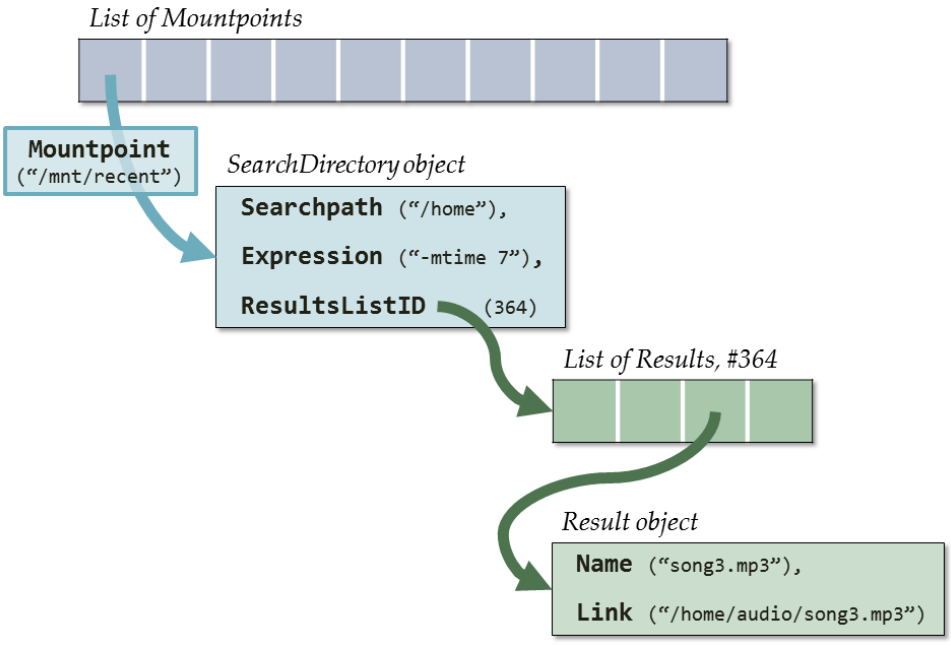

Figure 2: *This example depicts the location of a single result within the database structure. The database is designed to be conceptually simple, using objects and lists as design abstractions to simplify the underlying database implementation. Each search directory and each search result has a dedicated object to store its state. The colors of this figure reflect those of Figure 1, indicating the database corresponding to each structure.*

2.3 *Processes and their Algorithms*

The behavior of its processes is central to Searchdir's efficacy. **`searchmount`** is run whenever the user creates or deletes a search directory, whereas **`searchmaster`** runs continuously in the background and maintains accurate search results. A preview of these processes and their relationships is depicted in **Figure 3**, below.

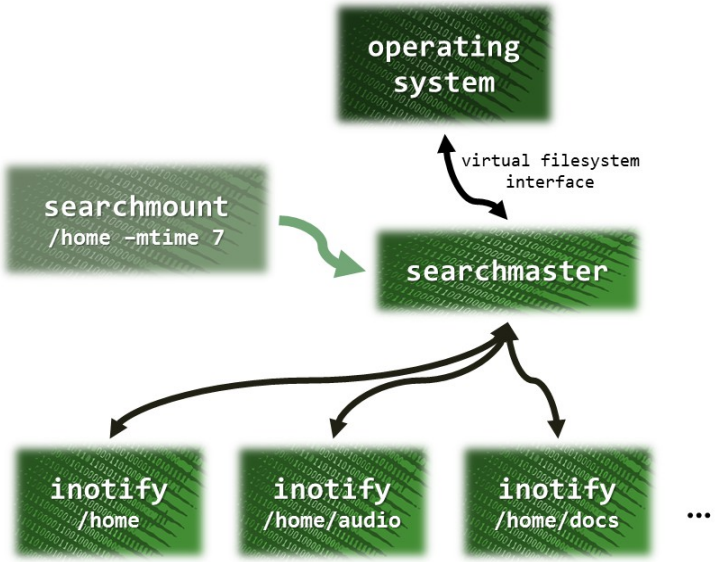

Figure 3: *Several processes are responsible for maintaining the Searchdir framework.* `searchmount` *starts off the persistent thread* `searchmaster`*, but is not itself persistent.* `searchmaster` *manages many persistent* `inotify` *worker instances, and also provides an interface with the operating system to create a virtual filesystem. Even if multiple search directories are mounted by* `searchmount`*, only one master instance of* `searchmaster` *is ever running.*

`searchmount` ***(initialization and createDirectory)***

`searchmount` first creates a new row in each of ***mountpoints.db*** and ***searchdir.db*** to represent the newly-created search directory as a *SearchDirectory* object. The "*Searchpath*" and "*Expression*" attributes are set to their values. The "*ResultsListID*" attribute identifies a list of files within ***results.db***: this list must be initialized by searching within ***`searchpath`*** for the search query ***`expression`*** using the Unix **`find`** tool, and creating a *Result* object for each result. [1] The output of **`find`** is a list of file paths, which the *Result* object will save both in full ("*Link*") and as a shortened filename ("*Name*"). Finally, so that the search directory remains up to date, **`searchmount`** must start **`searchmaster`** if it is not running already.

Searching within other search directories may create performance complications, and is disallowed; users can achieve the same functionality, instead, by using Boolean operators (AND, OR, NOT) to combine search expressions together.

`searchmaster`

Watching for Updates: **`inotify`** is a Unix tool that can watch a directory and send an event to **`searchmaster`** whenever a change occurs within that directory. [2] Using its databases to determine which search paths to monitor, **`searchmaster`** creates and

keeps track of many **inotify** worker processes. Since each **inotify** watches a single directory, the processes must be created recursively: one for each directory and subdirectory to monitor. The events that are sent back should be small, only indicating modification to a single file, and should contain that file's path – enough information for **searchmaster** to be able to find it. Upon receiving an event, **searchmaster** might need to check the file against search queries, add or remove search results from its database, or create new **inotify** instances for new subdirectories.

Interfacing with Unix: In order to implement the virtual filesystem, **searchmaster** must allow Unix to call the following methods:

» `DirEntryPointer` **`getFirstDirectoryEntry`**`(`*`name`*`)`
» `DirEntryPointer` **`readNextDirectoryEntry`**`(DirEntryPointer` *`prev`*`)`
» `String` **`fileName`**`(DirEntryPointer` *`entry`*`)`
» `String` **`readSymbolicLink`**`(DirEntryPointer` *`entry`*`)`

The **DirEntryPointer** data structure is a pair: **[mountpoint, result_index]**. This uniquely identifies a *Result* object within the database (see **Figure 4**, below), and allows **searchmaster** to easily read the filename or symbolic link of the corresponding result. Implementing these methods requires at most two database lookups.

Since the first element in each directory is at index 0, the method **getFirstDirectoryEntry** is trivial: it returns **DirEntryPointer = [name, 0]**. The decision to use *mountpoint* as the key for ***searchdir.db*** instead of an index – which forces Searchdir to use an extra database, ***mountpoints.db*** – is a tradeoff that was made so that **getFirstDirectoryEntry** would be fast; otherwise, it would have been necessary to conduct a time-consuming search through the list of mountpoints in order to find **name**.

DirEntryPointer: **["/mnt/recent", 2]**

Search Directory Data (searchdir.db)

Key		*Value*

"/mnt/recent"	→	*SearchDirectory* = {
		"Searchpath": "/home",
		"Expression": "-mtime 7",
		"ResultsListId": **364**
		}

Lists of Results (results.db)

Key		*Value*

<**364**, **2**>	→	*Result* = {
		"Name": "song3.mp3",
		"Link": "/home/audio/song3.mp3"
		}

Figure 4: *This example DirEntryPointer uniquely identifies a single search result by its search directory and its index within that directory's list of results. Only two database lookups (shown) are required in order to find the search result. From these two lookups, it is easy to return the next DirEntryPointer, to return the filename, or to return the symbolic link.*

3. Performance Analysis

3.1 Space

Suppose the user creates x search directories, each of which contains up to y different search results. Then, the size of each database is bounded as follows:

- ***mountpoints.db***: x rows
- ***searchdir.db***: x rows
- ***results.db***: xy rows

This use of storage space is justified: it allows search directories to remain accurate despite file modifications and system reboot, without requiring that a new **`find`** query be executed every time something changes. The databases are not excessively large – there is no way that their size can grow faster than the filesystem itself. Furthermore, since each row stores only a small amount of data (less than a kilobyte), it is conceivable that the databases could be fully cached using just 8-16 gigabytes of RAM.

3.2 Time

Searchdir has been designed so that each of the many operations it must perform can be carried out in a reasonable amount of time.

Creating a new search directory

When a search directory is created, **`searchmount`** must add one row to ***mountpoints.db*** and ***searchdir.db***, must execute one **`find`** search, and must add one row to ***results.db*** for each of the search results found. Of these steps, the **`find`** search is the most time-consuming, since it must access and check every file within the search path. This is reasonable, because at least one **`find`** execution is clearly essential in order to find any files at all. We can assume that **`find`** has been optimized for its task.

Updating in response to inotify events

Each *inotify* event indicates the file path of a specific file that has been recently modified. This file must be checked against every search directory to see whether each search directory's results need to be changed; with x search directories, this require $2x$ database lookups (one lookup, per search directory, to each of ***searchdir.db*** and ***mountpoints.db***). If the number of search directories remains small, this is fast.

Interfacing with Unix – the virtual filesystem

As described in "2.3 Processes and their Algorithms", the execution of each virtual filesystem method only requires, at worst, two database lookups. For example, to list all y contents of some search directory, **`readNextDirectoryEntry`** and **`fileName`** must be executed y times each, for $4y$ directory lookups in all.

3.3 Selected Use Cases

For analyzing the performance and scalability of the system under various edge cases, assume that a disk seek takes 10 ms, that disk reading and writing can be performed at 50 Mb/s, that memory (cache) access takes 1 ns, that each row of any database is no larger than 1 kb, and that all of the databases are stored in cache.

Immediate examination of search results

Suppose that the user runs **`searchmount`** in a directory with a modest (~300) number of files, and wishes to see the search results immediately. The most costly steps will be the **`find`** search and writing up to 300 rows to ***results.db.*** We assume that a single **`find`** search has a reasonable running-time. In the worst case, the second step takes 15 ms (disk seek + memory write), a length of time that is negligible to the user.

Very large directories

Suppose that the user runs **`searchmount`** in a directory with a large (~10,000) number of files, and wishes to keep the new search directory constantly up-to-date after the initial time cost that is required to initialize the search directory. Recall that **`inotify`** events were specified to be minimalistic: no details about the directory as a whole, only about the single file that was modified. Responding to a single event will therefore be fast, even if the file the event indicates is in a very large directory.

System reboot

On system reboot, all Searchdir data remains stored in databases. **`inotify`** processes must be restarted – this means **`searchmaster`** must check all search directories in the database, and create new **`inotify`** processes to watch relevant search paths. If the number of search directories remains small, and if search paths are not highly nested, this task will remain reasonable and will not adversely impact system reboot.

Highly-nested directories

If a search directory is used to search **`/root`**, a highly nested directory containing many files, many **`inotify`** processes will be spawned, which might affect system performance. Although implementations of **`searchmaster`** should make sure to protect against this by make sure never to create redundant **`inotify`** processes, it is not clear whether this failure mode could ever be fully prevented. Spawning an **`inotify`** process for each directory is necessary if search directories are to be constantly and consistently accurate.

5. Conclusion

As this system is implemented, one issue for implementers to consider is the loss of **`inotify`** events: when the event buffer overflows, as might happen if a user makes many file modifications very quickly, events are lost. [2] Searchdir should be robust to this possibility, and handle it gracefully.

Another improvement that implementers should consider is to ensure that **`searchmount -master`** runs immediately on startup, so that Searchdir does not need to trust that the user will remember to do so if the machine reboots.

Regardless of these improvements to make, the present design for Searchdir will efficiently address the many use cases of Search Directories. Using existing Unix tools and an intuitive, object-oriented database, this design maintains up-to-date and accurate search results, allows flexible user-defined search queries, and preserves system performance even at large scales.

Acknowledgments

I thank Jared Berezin for thorough writing feedback and Qian Long for suggesting technical details to consider. I also thank Favyen Bastani for peer review of the design proposal and Brandon Miranda for helpful discussions about figures and about database design.

References

[1] P. Christias. "UNIX man pages: find." Internet: http://unixhelp.ed.ac.uk/CGI/man-cgi?find, 1994 (Mar. 21, 2014).

[2] J. McCutchan. "inotify(7): monitoring file system events - Linux man page." Internet: http://linux.die.net/man/7/inotify, 2005 (Mar. 21, 2014).

[3] D. M. Ritchie and K. Thompson. "The UNIX Time-Sharing System." *The Bell System Technical Journal*, **57** no. 6 part 2, July 1978. <https://web.mit.edu/6.033/www/papers/protected/cacm.html>.

[4] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design: An Introduction*. Waltham, Massachusetts: Morgan Kaufman, 2009.

Word count: 2430