# A Search-Folder System for Unix

Jason Paller-Rzepka

jasonpr@mit.edu

Design Proposal 1

R10: Szolovits TR 2

21 March 2014

# 1. Introduction

Unix users often repeat certain `find` commands to locate files of interest. One way for them to eliminate this repetitive task from their workflow is to create Search Folders. A Search Folder is a virtual directory that contains the results of a file search query. Search Folders automatically update their contents as the results of the file search change, so they are always up-to-date. By using Search Folders, Unix users can significantly reduce the number of repetitive file searches they perform.

This proposal discusses the design of a Search-Folder system for Unix. The system maintains iterable sets, called PRADLLs, of search results, and listens for file-system updates to detect changes and update those search results. The design allows the system to operate quickly: requests to iterate through directory contents are fast, and the search results update very soon after the underlying data is modified.

In designing this system, we made two major design tradeoffs. First, we sacrificed space efficiency for time efficiency: the system maintains large sets of search results, on disk, so that it can quickly serve them to users. Second, we sacrificed startup time for freshness: registering many file-update listeners takes time, but it allows the system to maintain fresh results without doing slow, periodic sweeps of the target folders.

# 2. Design

The following subsections describe the API that the Smart Folder system exposes, the Persisted Random-Access Doubly Linked List abstraction that the Smart Folder system uses, and the implementation of the Smart Folder system.

## 2.1 API

The system's interface consists of methods to perform the following actions:

- **Create and mount a Search Folder** for a directory, with some search parameters. This functionality is exposed as a command-line utility. For example, the following command creates a Search Folder for all files in `/docs` larger than ten megabytes and mounts it at `/big-docs`:

  ```
  $ searchmount /big-docs /docs –size +10M
  ```

  Note that the criteria that a user can pass to `searchmount` are the same criteria that can be passed to whatever version of `find` is installed on that user's system.

- **Read the entries** from a Search Folder. This functionality is exposed through the two functions, `getFirstDirectoryEntry(searchFolderPath)` and `getNextDirectoryEntry(DirEntryPointer previousFile),` where a `DirEntryPointer` is a string containing the full path of some target file.
- **Produce a human-readable name** for an entry in a Search Folder. This functionality is exposed via the function `filename(DirEntryPointer entry).`
- **Get the target** of an entry in a Search Folder—that is, the full path of the file that this entry represents. This functionality is exposed via the function `readSymbolicLink(DirEntryPointer entry).`
- **Mount all Search Folders.** After a restart, the Search-Folder system needs to re-mount all the Search Folders, and begin to run the system again. Running `srchfldrd`, the Search Folder Daemon, does just that.

## 2.2 The PRADLL Abstraction

In order to implement the Search Folder system, it is helpful to have a persisted set-of-strings data store with the following properties:

- Constant-time lookup, insertion, and deletion
- Ability to iterate over items in constant-time per item
- Ability to do low-overhead insertion of large sets of items
- Existence of a global lock

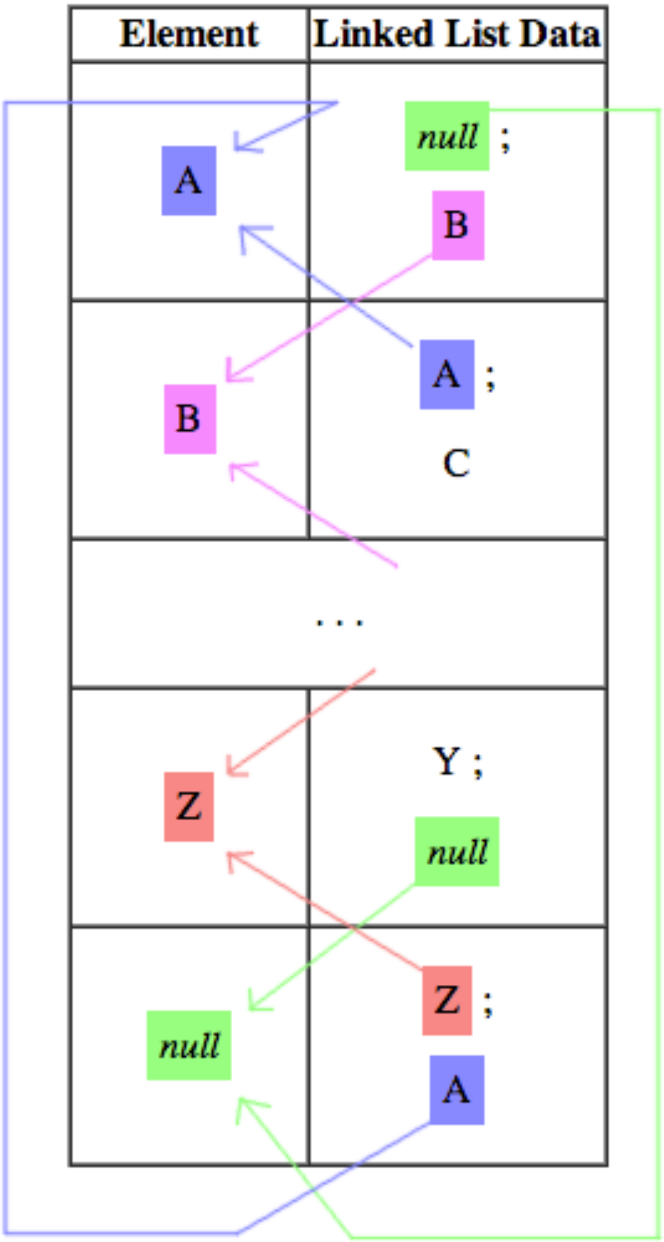
We introduce a data abstraction layer that has those properties: the Persisted Random-Access Doubly Linked List (PRADLL). PRADLLs have two components: a basic string-to-string key-value store database, and a single lock.

The key-value store is used as follows: each element in the PRADLL is represented with a row in the key-value store. The key is the element itself. The value is a pair of elements (serialized into string form): the element the came before this one, and the element that comes after this one. (This pair is sometimes called a pair of “pointers,” referring to the way that in-memory doubly linked lists are implemented.) There is also one extra row in the key-value store: the key is *null* and the value is a pair pointing to the last and first elements. In this way, the key-value store imitates a doubly linked list. See Figure 1 for an example underlying key-value store of a PRADLL.

The lock is simply used as a global lock.

Note that inserting an element involves adding one row to the key-value store, and modifying two rows (so the pointers are correct). The naïve way of adding *n* elements would be to add them one at a time, costing *n* additions and *2n* modifications. But, it is possible to do low-overhead insertion of large sets of items by building up the list of items in memory, inserting the entire list at a cost of *n* insertions, then changing only two rows to point to the beginning and end of that list.

Also note that this structure has a global lock, which must be acquired before performing any reads or writes.



| Element | Linked List Data |
|---|---|
| A | *null* ;<br>B |
| B | A ;<br>C |
| . . . | |
| Z | Y ;<br>*null* |
| *null* | Z ;<br>A |

**Figure 1: The Key-Value Store Representation of a Doubly Linked List**
Each row of the table is a row of the key-value store that underlies a PRADLL. Each key is an element of the PRADLL. Each value contains two elements: one pointing to the previous row, one pointing to the next row. The *null* key represents the edge of the list—it comes before the first key, and it comes after the last key.

## 2.3 Implementation

The main components of the design are: on-disk data structures that store the properties and contents of Search Folders, a file-update queue for collecting changes to target folders, and a daemon for processing those changes and applying them to the data structures.

### 2.3.1 On-Disk Data Structures

The Search Folder system persists some data on disk: PRADLLs containing the contents of individual Search Folders, and a few database tables for keeping track of all Search Folders and their metadata.

*2.3.1.1 Contents PRADLLs*

A Search Folder's contents are stored in a PRADLL of `DirEntryPointer`s. (Recall that a `DirEntryPointer` is a string containing the full path of a file.) This allows for quick lookup, insertion and deletion, and iteration of items. It also allows for low-overhead insertion of many items at once, which is useful when initially populating the database with potentially many matching files.

A contents PRADLLs is populated with the result of a `find` command as soon as a Search Folder is created. The daemon keeps contents PRADLLs up-to-date thereafter, as described in 2.3.3.

*2.3.1.2 Search Folder Metadata*

The following data structures are used to keep track of all the Search Folders and their metadata:

- One PRADLL of all the Search Folder paths.
- One key-value store mapping Search Folder paths to individual Contents PRADLLs.
- One key-value store mapping Search folder paths to query parameter strings.

### 2.3.2 The File-Update Queue

The system maintains a file-update queue, which can be the target of many `inotify` listeners. The system registers `inotify` listeners to any folders it is searching over (both when `searchmount` is run, and when the daemon detects the creation of a new folder under a searched folder). All these listeners point to the same queue, so the daemon can read changes from this single source, rather than listening to potentially many queues.

### 2.3.3 The Daemon

In steady-state operation, the daemon has four jobs: add `inotify` listeners to newly added folders, update Contents PRADLLs when relevant files are added, modified, or deleted; accept and acknowledge the creation of new Search Folders via inter-process communications from the `searchmount` utility, and buffer changes when a PRADLL is locked when the `searchmount` process that created it is populating it initially.

*2.3.3.1 Starting Up*

Before it can get to the steady state, the daemon must start up. First, the daemon, iterates through the all-Search-Folders PRADLL, and registers an `inotify` listener for each folder (taking care not to register duplicate listeners). Then, it reads the mapping of Search Folders to query parameter strings into memory.

*2.3.3.2 Adding inotify listeners*

Whenever a new folder is added to a directory that is being listened to, the daemon adds an `inotify` listener to that directory, as well. Additionally, it artificially adds that directory's contents to the file-update queue, so they can be processed.

*2.3.3.3 Processing Changes*

The daemon reads notifications from the file-update queue. When it gets a notification, it iterates through all the Search Folders and associated query parameter strings that it loaded into memory and decides whether the changed file both lies inside the searched folder and matches the query parameters (delegating to the `find` utility for the second decision). If so, then it makes the corresponding addition, or does nothing if the file was already present in the PRADLL. If not, then it makes the corresponding deletion, or does nothing. If the notification was of a deletion, then it makes the corresponding deletions, or does nothing.

*2.3.3.4 Receiving Updates*

When a `searchmount` process creates a new Contents PRADLL, it sends a message to the daemon so the daemon adds that Search Folder to its in-memory mapping of Search Folders to query parameter strings. Only after the `searchmount` process receives an acknowledgement does that process populate the PRADLL. Thus, there will never be a time that a PRADLL has been already been populated, but is not being updated with file changes. (See 2.3.3.5 for a description of how the daemon handles changes relevant to a not-yet initially populated PRADLL.)

So, the daemon must listen for these messages from `searchmount` processes, and respond to them.

*2.3.3.5 Buffering Changes*

When `searchmount` is initially populating a PRADLL, it holds that PRADLL's lock. So, if changes come into the file-update queue that are relevant to that PRADLL, the daemon must buffer them into a PRADLL-specific queue and wait to apply them until the other process releases the lock. This means that, before writing a change to any PRADLL, the daemon must check that it's not maintaining a queue for that PRADLL. If it is maintaining a queue, then it must either flush the queue by applying it to the PRADLL, or it must continue buffering changes.

Note, it is also possible that the changes will be redundant—it's possible that the affected file was added to the PRADLL during `searchmount'` s initial sweep *after* the change was made. The daemon must be sure not to duplicate any entries, and skip the deletions of any non-existent entries. If it does this, it is guaranteed to leave the PRADLL consistent with the filesystem, eventually.

**2.3.4 Implementing the API**

When a user executes `searchmount`, the following actions are taken:

1. Recursively register `inotify` listeners on target directory.
2. Add relevant rows to metadata data stores.
3. Create Contents PRADLL.
4. Obtain lock of Contents PRADLL.
5. Register new Search Folder with daemon.
6. Populate Contents PRADLL with results of a `find` command over the target folder.
7. Release the Contents PRADLL's lock.

This process ensures that there will never be a moment where a file could change without that change propagating to the Contents PRADLL.

Reading the entries of a Search Folder is as easy as iterating through its Contents PRADLL.

Getting the name and path of an entry is as easy as getting the filename from a path, and getting the path from a path (the identity function!).

Starting the daemon was already described in 2.3.3.1

# 3. Analysis

In this section, we list some expected use cases for this system. Then, we will evaluate the temporal performance, spatial performance, correctness guarantees, and scalability in view of those use cases.

## 3.1 Use Cases

We will examine the following use cases:

1. A user creates a Search Folder over a directory containing 300 files, then examines the resulting Search Folder.
2. A user creates a Search Folder over her root directory, which contains 100,000 files, then examines the resulting Search Folder.
3. A user creates a Search Folder over her root directory, which contains 100,000 files, then reboots.
4. A user creates 100 search folders over the same directory, which contains 1,000 subdirectories, then reboots.

## 3.2 Temporal Performance

We will examine the speed of various operations:

### 3.2.1 Create

Creating a search folder is fast—nearly as fast as just running a `find` command. There is some overhead. For example, registering `inotify` listeners and adding/modifying data stores takes time. However, this overhead causes only constant-factor slowdown. Since the overhead operations probably involve fewer disk seeks than the same `find` command would take, and since `find` is actually used to perform the search, we expect no more than a factor of two slowdown—and possibly much less. This is perfectly reasonable for interactive use. Whether the user searches over 300 or 100,000 files, fast `find`s are met with fast `searchmount`s, and slow `find`s are met with slow `searchmount`s.

### 3.2.2 Update

Updating Search Folders is very fast—as soon as a change is made, the daemon picks it up, and updates the PRADLL quickly. Our use of a single file-update queue, instead of per-Search Folder queues, makes this process even faster, because we do not need to poll multiple queues.

This process is fast on small and large target folders—our use of `inotify` eliminates the need to do costly re-searches to keep the data fresh.

### 3.2.3 Query

Querying is very fast, since we store folder data in a PRADLL. Additionally, it is kept up-to-date, since the daemon works quickly. An alternative would have been to simply perform a `find` call whenever the user performed a query, but that would have been disastrously slow in the case of large target directories.

### 3.2.4 Mount

Mounting takes time proportional to the number of target directories and subdirectories in the system, because it must setup an `inotify` listener for each one. This means that the mounting process will take a fairly long time if, say, the root directory is a target. However, it performs better than the alternative possibility, in which each directory/subdirectory has one listener per target folder it sits under. In Use Case #4, our system performs 100 times faster than that alternative system! It also performs faster than a system which requires a full `find` call at mount time, since we persist the `find` result on disk.

## 3.3 Spatial Performance

Our system uses disk space proportional to the number of matching files for each Search Folder. We could have drastically reduced disk usage by simply running a `find` command whenever the user queried a Search Folder, but that would have been imprudent: disk space is much cheaper than our users' time!

## 3.4 Correctness

This system is fragile, in the sense that even one missed file-update notification can cause long-term differences between the Search Folder contents and the correct results. But, careful use of locks and queues allows us to ensure that there will never be a missed notification. An alternative would have been not to use the queues, to populate the Contents PRADLL first, and then to register the file-update listeners, just hoping that no file updates would happen in the mean time. This system would have the benefit of being simpler, but the disadvantage of being susceptible to missed notifications.

## 3.5 Scalability

In order to scale to large systems, the long creation and mounting times would need to be addressed. This could presumably be achieved through parallelization. With those problems solved, the system would scale until the throughput of the daemon's file-updating became a problem.

## 4. Conclusion

The Smart Folder system provides a time-efficient way to maintain auto-updating virtual directories that contain the results of file searches. It does so by maintaining iterable sets of search results, and listening for file-system updates to detect changes and update those search results. It carefully orders operations to ensure that the Search Folders are eventually consistent with the correct results.

Before finalizing this design, the following challenges should be addressed: handling unexpected crashes, and parallelizing search file search and `inotify`-listener registration.

## 5. Acknowledgements

I would like to thank Jane Kokernak for giving me feedback on the memo and proposal that led up to this report. I would also like to thank Tiffany Chen for giving me feedback on my initial design.

## 6. References

[1] Saltzer, J.H., and Frans Kaashoek. *Principles of Computer System Design: An Introduction.* Burlington, MA: Morgan Kaufmann, 2009. Print.

Word Count: 2607 (Limit: 2500 ± 5%)