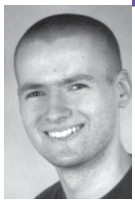
STEFAN BÜTTCHER AND
CHARLES L.A. CLARKE

adding full-text filesystem search to Linux



Stefan Büttcher received a Master's degree in computer science from the University of Erlangen, Germany. Since 2004, he has been a Ph.D. student at the University of Waterloo, Canada. His research interests include all aspects of high-performance search engines, especially index maintenance strategies for dynamic text collection. Stefan is the main developer of the Wumpus search engine.

sbuettch@plg.uwaterloo.ca

Charles Clarke is an Associate Professor in the School of Computer Science at the University of Waterloo. His research interests include information storage and retrieval, software development tools, and programming language implementation. Clarke received his Ph.D. from the University of Waterloo in 1996. From 1996 to 1999 he was an assistant professor in the Department of Electrical and Computer Engineering at the University of Toronto. He has also held software development positions at a number of computer consulting and engineering firms.

claclark@plg.uwaterloo.ca

IN THE PAST TWO YEARS, FULL-TEXT desktop search systems have experienced an amazing updraft. For Windows, there now are about a dozen independent desktop search engines from which the user can choose. For Linux, the situation is different; only a few desktop search systems exist.

In this article we report on experiences we had while developing Wumpus, a full-text filesystem search engine for Linux. We discuss major design decisions and point out some changes that, from a search engine developer's point of view, need to be made to the Linux kernel to support real-time filesystem indexing and search.

The goal of our research efforts is the development of a unified filesystem search engine that can be used by multiple users and that can cover multiple storage devices, both local and network-wide (local hard drives, USB sticks, NFS mounts, etc.). Search results returned by the engine should always be consistent with the current content of the file system. Inconsistencies resulting from recent file changes should have a lifetime of at most a few seconds.

The vehicle we are using to reach that goal is the Wumpus search engine, a hybrid filesystem search and general-purpose information retrieval system. Wumpus is free software, licensed under the terms of the GNU General Public License, and is available for download from the Wumpus Web site, http://www.wumpus-search.org/. It is work in progress and not yet suitable for everyday use as a filesystem search engine.

Wumpus is a keyword-based search engine. It supports state-of-the-art result ranking algorithms, as well as structural queries (phrase queries and near operators) and Boolean operators. Its back-end index data structure is a set of inverted files. Each inverted file realizes a mapping from terms to their respective occurrences within the file system. (For a thorough discussion of inverted files and their advantages over alternative index data structures, see Zobel et al. [4]). In conjunction, the inverted files can be used to efficiently obtain a list of all occurrences of a given term within the file system. The result of a search query (e.g., "find the set of all files containing the given query terms") can then be produced by combining the lists of all query terms in a straightforward way.

When new files are created or existing files are modified, index information for the new data is added to in-memory index buffers. Whenever the amount of these in-memory data exceeds a certain threshold, they are written to disk, resulting in a new on-disk inverted file. Several inverted files may exist in parallel and are merged in a hierarchical fashion when it is appropriate to do so. This can be done very efficiently. A detailed description of index maintenance strategies for dynamic text collections can be found in the literature [2, 3].

When we started to develop our search engine, we had to make several major design decisions. Among the most important were index locality decisions. In a typical Linux installation, the file system will contain files belonging to more than a single user. It will also span across multiple mount points, representing different storage devices. These two aspects of filesystem search define two independent locality axes (the user axis and the device axis, as shown in Table 1). We had to decide whether index information should be stored locally or globally along each axis. Other locality axes, such as the time axis, exist and also play a role in filesystem indexing, but the user and the device axEs are the most important.

		Device Axis	
		Local	*Global*
User Axis	*Local*	A separate index for each user on each device	Per-user indices, each covering all devices
	Global	Device-specific indices, each containing data for all users	A single index covering all users and all devices

TABLE 1: THE TWO MAIN LOCALITY AXES IN MULTIUSER, MULTIDEVICE FILESYSTEM SEARCH

User Axis: A Single, Global Index to Be Accessed by All Users

Most existing desktop search tools maintain per-user indices. Although this is acceptable in single-user search environments, in pure desktop search environments (i.e., without the option to search the entire file system), and in environments with a small number of users and very little interaction among them (as is the case in a typical Windows system), it is not a good idea in a true multiuser filesystem search environment. Maintaining per-user indices, where each index only contains information about files that may be searched by the respective user, leads to two types of problems:

- Redundancy: Many files (such as man pages and other documentation files) can be accessed by all users in the system. All these files have to be independently indexed for each user in the system, leading to a massive storage overhead in systems with more than a handful of users.
- Performance: If per-user indices are used, then even a single chmod or chown operation can trigger a large number of disk operations, because the respective file needs to be completely reindexed (or data need to be copied from one user's index to another user's index) each time a user executes chown. Even in a system with only two users, this can be exploited to realize a denial-of-service attack on the indexing service.

The only solution to these problems is to use a single index that is shared by all users in the system, instead of many per-user indices. This index is maintained by a process with superuser rights that can add new information to the index when new files are created and remove data from the index when files are deleted. chmod and chown operations can then be dealt with by simply updating index metadata, without the need to reindex the file content.

Of course, to guarantee data privacy, the global index, because it contains information about all indexable files in the system, may never be accessed directly by a user.

Instead, whenever a user submits a search query, it is sent to the indexing service (running with superuser rights). The indexing service then processes the query, fetching all necessary data from the index, and returns the search results to the user, applying all security restrictions that are necessary to make the search results consistent with the user's view of the file system, while not revealing any information about files that may not be accessed by the user who submitted the query. The problem of applying user-specific security restrictions to the search results is nontrivial, but it can be solved (see [1] for details).

Device Axis: Local, Per-Device Indices

When experimenting with various desktop search systems for Windows, we noticed that most of them had problems with removable media. They either refused to index data on removable media altogether, or they added information about files on removable media to the index, but removing the medium from the system later on was not reflected by the index, and search results still referred to files on a USB stick, for example, even after the stick had been unplugged.

If index data are stored in a global, system-wide index, it is not clear how to deal with removable media. Should the index data be removed from the index immediately after the medium is removed from the system? If not, how long should the indexing service wait until it removes the data? Should external hard drives be treated as removable media?

The only solution to these problems is to maintain per-device indices. In Linux, for instance, this means that each device (/dev/hda, /dev/hdb, etc.) will get its own local index that only contains information about files on that particular device. Whenever a device is removed from the file system, the indexing process associated with that device is terminated. Whenever a device is added to the file system, a new indexing process is started for the new device (or not, depending on parameter settings). Search queries are processed by combining the information found in the individual per-device indices and returning the search results, which may refer to several different devices, to the user.

For network file systems such as NFS mounts, this means that the index is not kept on the client side, but on the server that contains physical storage device. This requires additional communication between the NFS server and the client during the processing of a search query and is a potential bottleneck in situations where an NFS server is accessed by a large number of clients and where many users want to search for data on the server. Nonetheless, this is the only way to allow the index to be updated in real time, as it is impossible for an NFS client to be informed of all changes that take place in a remote file system.

Maintaining per-device indices also makes it possible to remove a storage device from one computer system and attach it to another one without needing to reindex the files stored on the device. Since the index is kept on the device itself, all index information will immediately be available on the new system. As far as we know, the same approach is followed by Apple's Spotlight.

Filesystem Event Notification

To be able to fully implement this type of filesystem search framework, a comprehensive filesystem event notification interface is needed so that the operating system kernel can inform the indexing service about changes in the file system, that is, changes to the content of a file or changes to its metadata, such as file name and access privileges. Many operating systems provide system calls that allow a process to register for changes in a certain part of the file system (usually a directory, or a subtree rooted at a given directory) and to receive notifications about all filesystem events affecting that part of the file system.

In Windows, for example, an application can use the FindFirstChangeNotification system call (and related functions) to register for a variety of filesystem events in a given directory. The system call can also be used to register for changes in arbitrary subdirectories of the given directory. The latter is called a *recursive watch* and is very useful if one wants a process to monitor the entire file system.

THE TRADITIONAL LINUX NOTIFICATION SYSTEM: dnotify

In Linux, filesystem event notification had traditionally been realized through the dnotify interface. In dnotify, a process can register for changes to the contents of a particular directory by obtaining a handle to that directory and performing an fcntl system call for the handle. Events will be sent to the process in the form of UNIX signals. As soon as the process releases a handle, it will no longer be notified of changes in the directory associated with it.

This approach has two major problems. First, the interface requires an application to keep an open handle to each directory that is being watched for changes. For very large file systems, with hundreds of thousands of directories, this is not feasible. Second, it is not possible to register for recursive watches that include all subdirectories of the given directory. Again, for large file systems this is problematic. After a system reboot, for example, the entire file system needs to be scanned to obtain a handle to every directory. Depending on the size of the file system, this can take from several minutes to several hours.

THE NEW LINUX NOTIFICATION SYSTEM: inotify

Since version 2.6.13 (August 2005), the Linux kernel supports a second event notification interface, inotify. inotify is, for instance, used by the Beagle (http://www.gnome.org/projects/beagle/) search system.

The new interface removes dnotify's main shortcoming, the necessity of having an open handle to every directory in the file system. With inotify, an application obtains a handle to an inotify queue object and subsequently registers for event notification for all directories in which it is interested. The queue handle can be treated like an ordinary file handle, allowing synchronous and asynchronous I/O.

dnotify's second main shortcoming, the necessity of scanning the entire file system after a system reboot, is shared by inotify. Recursive watches are not supported. With inotify, a process has to register for each directory separately. The rationale behind this is that it allows file permission to be checked during the registration process; the request can then simply be rejected if the process does not have sufficient access privileges. If recursive watches were supported, this check would need to be performed at notification time, potentially adding significant overhead to the notification system. Unfortunately, inotify's security model does not take into account the possibility of access privileges being changed after a user obtains a watch for a directory. If the user does have read permissions for a directory and is granted the right to watch the directory, but loses read permission for the directory later on, inotify will still notify the user about changes in the directory.

The nonexistence of recursive watches in inotify introduces potential race conditions, for example when files and directory hierarchies are extracted from an archive and files are moved to other directories before the indexing service can register for changes in the new directories. This adds additional complexity to the indexing system and could have been avoided if recursive watches were supported.

SHORTCOMINGS OF THE EXISTING LINUX NOTIFICATION SERVICES

In addition to the absence of recursive watches, the existing filesystem notification facilities of the Linux kernel lack a few other features that are desirable for full-text filesystem search and imperative for the framework we propose:

- Fine-grained file change notification: When the content of a file is changed, inotify (and dnotify) rather laconically reports "file changed" but does not elaborate on which exact part of the file is affected by the change. Suppose a user has a large mailbox file, containing thousands of messages, and a single message is appended to the existing file. With inotify, the indexing service will have to guess that the change was only an append operation, but it can never be sure without rereading the entire file, which might take a long time, depending on the size of the mailbox file. A more detailed notification message, including the start and the end offset of the part of the file affected by the change, is desirable. This feature is trivial to implement but will probably require a change of the current inotify interface to userspace processes.
- Unmount request notification: Maintaining per-device indices requires the indexing system to have open files on each mounted device. This is imperative, as all index maintenance strategies for dynamic search systems rely on the ability to buffer updates in memory and only perform physical index updates from time to time. As a consequence, devices cannot be unmounted any more ("umount: device is busy"). To be able to unmount a device, the indexing process for that device needs to be terminated first. However, during the short period of time between shutting down the indexing process and unmounting the device, files can be changed. Those changes will never be detected by the indexing system unless it performs an exhaustive scan every time a device is added to the file system. To solve this problem, the operating system needs to provide atomic unmount operations that can include actions of userspace processes. Although this would probably require major changes to the Linux kernel, it seems to be the only clean solution to the unmount dilemma.

AN EXPERIMENTAL SOLUTION: fschange

The problems discussed here are addressed by the experimental fschange notification system. fschange is a patch for the Linux kernel and is available online (http://stefan.buettcher.org/cs/fschange/). After the kernel is updated, it can be accessed by a userspace process through the proc file system: /proc/fschange. In contrast to the existing notification interfaces part of the Linux kernel, fschange does not require a process to register for each directory individually. It provides a global view of the file system. By reading from /proc/fschange, the process obtains information about all changes taking place in the entire file system. Consequently, a process needs to have superuser privileges to be allowed to read from the file.

Because fschange provides a global view, exhaustive disk scans after a reboot or after mounting a new device are no longer necessary. Race conditions stemming from the necessity to register for each directory individually are eliminated, too. In addition to filesystem indexing, the interface can also be used by other types of applications (e.g., backup and file replication systems).

fschange supports most of the message types provided by inotify, plus a few others, such as mount notifications (needed to create a new indexing process when a storage device is added to the system). When a file is changed through a write or an mmap operation, fschange tells the user-space process not only the name of the file that was changed but also the start and end offset of the part of the file affected by the change.

The unmount problem discussed in the foregoing is addressed by providing two unmount events: UNMOUNT_REQ, indicating that a process requested unmounting an active storage device and that the request was rejected owing to open files for the device; and UNMOUNT, indicating that a storage device was successfully unmounted. When the indexing service receives a UNMOUNT_REQ notification, it terminates the process for the storage device affected by the unmount, closing open files for that device. In our prototype system, the umount system tool was modified in such a way that it sends a sequence of unmount requests to the kernel until the kernel reports a successful execution of the unmount operation or until a time-out (usually a few seconds) is reached. This strategy does not really solve the unmount dilemma, but at least it allows one to unmount file systems without losing excessive amounts of index data, which would otherwise be impossible.

Conclusion

We believe that a true filesystem search engine for Linux, providing each user with a global view of the searchable file system, is badly needed. We have outlined some important properties of such a search engine and discussed why it is difficult to implement a search engine with these properties, given the current support for filesystem notification provided by the Linux kernel. We hope that some of the functionalities we suggest will be added to the existing kernel services in the future, opening the way for real-time filesystem search in Linux.

REFERENCES

[1] S. Büttcher and C.L.A. Clarke, "A Security Model for Full-Text File System Search in Multi-User Environments," *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST 2005),* San Francisco, U.S.A., December 2005.

[2] S. Büttcher and C.L.A. Clarke, "Indexing Time vs. Query Time Trade-offs in Dynamic Information Retrieval Systems," *Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM 2005)*, Bremen, Germany, November 2005.

[3] N. Lester, A. Moffat, and J. Zobel, "Fast On-Line Index Construction by Geometric Partitioning," *Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM 2005),* Bremen, Germany, November 2005.

[4] J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted Files versus Signature Files for Text Indexing," *ACM Transactions on Database Systems,* 23(4):453–490, 1998.