Proposal for a Versioning File System

6.033 Design Project 1

Xinyue Ye

March 1, 2013

1 OVERVIEW

This proposal describes the design of a versioning file system which tracks old file versions. The approach is to create individual log files for each file. The log files contain records that represent modifications between versions. This design is space-efficient and provides all the functionality of the Unix File System (UFS).

2 DESIGN DESCRIPTION

2.1 Data Structures

The key data object in the design is the log file that maintains the changes between each version of a file.

2.1.1 On-Disk

Log File: Each file will have a *.log* file that is stored on-disk that keeps track of changes for that file between each version. The *.log* file is created and stored in the same directory as the file and consists of a sequence of records. A record is an entry that contains the version ID, or number, of the file, the byte offset at which the modification begins, the length of the new data to be written, and the original data in the file starting at the offset. Each new record is triggered by a `write()` call to the file and created by making a `write()` call to the *.log* file that writes the record metadata to the log. Then, the old data in the file will be overwritten with new data in a third `write()` call. Figure 1 gives a more detailed description of the syntax and grammar of the records in the *.log* file and Figure 2 gives a clear example of the logging process that takes place.

Figure 1

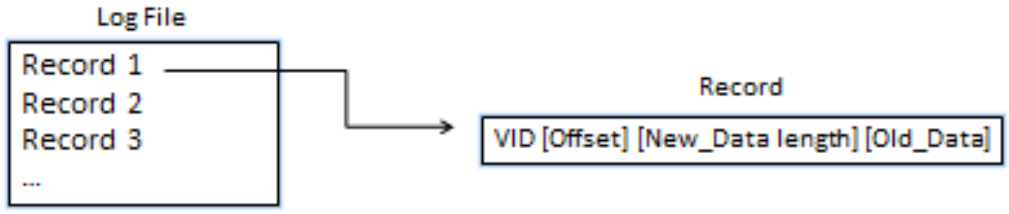

Figure 1: The grammar and syntax for a record are shown. Each record has a version ID, the byte offset of the old data, the length of the new data, and the old data itself.

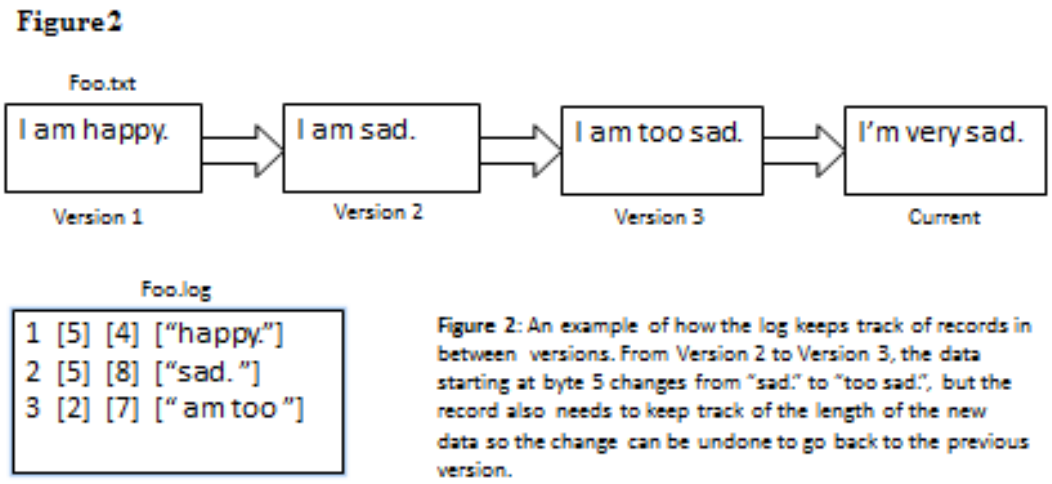

Figure 2: An example of how the log keeps track of records in between versions. From Version 2 to Version 3, the data starting at byte 5 changes from "sad." to "too sad.", but the record also needs to keep track of the length of the new data so the change can be undone to go back to the previous version.

Inode: The log file's inode needs to contain a field called `num_versions` that keeps track of the total number of versions of the file. It should also store a configurable variable, `max_size`, for the maximum size of the *.log* file; this will prove to be useful during garbage collection. The file inode needs to store one additional attribute: a Boolean variable, `is_versioned`, which represents whether the file or directory should be versioned.

2.1.2 In-Memory

B-trees: The in-memory data structure is nearly identical to the B-tree in-memory data structure of the UFS.

Shadow Files: A shadow file copy is a temporary copy of a current file version and is used when the user wants to access a previous version. After it is created, using the records in the *.log* file, the changes are made one at a time to the shadow copy until it is at the correct version. This shadow copy is then opened with read-only access and when the user closes the shadow file, it is deleted from memory.

2.2 Functionality

The versioning file system supports several Unix operations as well as additional functionality such as versioning options and garbage collection.

Versioning Options

The user can specify files and directories to be excluded from versioning. Each file's inode and directory's inode (not including *.log* files) has a configurable Boolean variable, `is_versioned`, which determines if the file or directory should be versioned.

Garbage Collection

There is default garbage collection that is built-in to the file system. If the size of a *.log* file reaches the `max_size` of that *.log* file, then the oldest version in the *.log* file will be deleted. The default `max_size` of a .log file is 1 Megabyte; however, the user is able to change that to accommodate their needs.

2.3 Application Programming Interface

Applications will use the following API to interact with the versioning file system.

- `read(filename,version_num)`: Reads the current version of the file if `version_num` is 0. Otherwise, makes a shadow copy of the current file and "undos" each change in the log file until the correct version.
- `write(filename,buffer,count)`: Makes a read() call to the file and reads in the old data that is about to be overwritten. This old data and other record metadata (see Figure 1) is written to the *.log* file. Finally, a write() call is made to the file to overwrite the old data with the new data.
- `create(filename)`: Creates a file and a *.log* file with the `filename`.
- `open(filename,version_num)`: Opens current version of the file and proceeds as `read(filename,version_num)`.
- `search_all_versions(filename,string)`: Searches the file's *.log* file and current version for the string.
- `search_all_files(string)`: Searches all files and *.log* files for the string.
- `rename(filename,new_name)`: Renames the file and file's *.log* file to `new_name`. Linking is implemented as in the UFS.

3 CONCLUSION

The proposed design is a space efficient versioning file system; however, further optimizations could be explored. For example, snapshots of older version can be stored to reduce the number of "undos" required. Furthermore, future tasks include implementation details and additional functionality.