A Unix-Based
Versioning File System

Kevin Wong
Strauss T2
6.033 Design Project 1
Proposal

Overview

Programmers often need a revision control to keep a record of old codes. One form of revision control is a versioning file system (VFS). VFS allows a file to exist in several versions at the same time, allowing users to review previous versions of the file when needed. This proposal discusses the design of a Unix-based VFS that fulfills the following requirements and use cases:

- **Space**. Only one additional block of space per one changed block is allowed for each new version created. New version is created when a user writes to a file or creates a new file.
- **Efficiency**. The system has to be reasonably fast to use. This includes reading or writing a new file, searching for a string across all versions of a file, and searching for a string across all versions in the file system.
- **Customizability**. User can exclude files and directories from versioning.
- **Logging**. All relevant operations are appended to a log file.

The system will assume no failures. The block sharing approach described in the next section satisfies the above requirements.

Design Description

Data Structures

Recall that a file is associated with an inode, which contains information about the file and a list of block numbers, which are references to blocks, fixed size data units on the disk. Therefore, we can think of a file as a list of block numbers, and the block is where the data is actually stored.

The main idea of the design is to treat **each version as a file**, and to **share unmodified blocks** across versions that contain them. To support this, we need two data structures:

- An **on-disk** file, properly named, to represent each version of a file.
  The versions of a file, for example, myDir/foo.txt, are stored in a sub-directory named after the directory in which the file is contained, i.e. myDir;versions, and the current version keeps the original name of the file: foo.txt. Earlier versions are named after the original file, with a semi-colon and the version number appended to the name. For example, if the current version is the 8th version, then the previous version would be myDir/myDir;versions/foo.txt;7. Note that all the previous versions would be read-only files and no further changes to those files are allowed.

  Each version created will have the same unmodified blocks' numbers as the current version. Figure 1 illustrates how block sharing works in detail as a file is modified and versions are created.

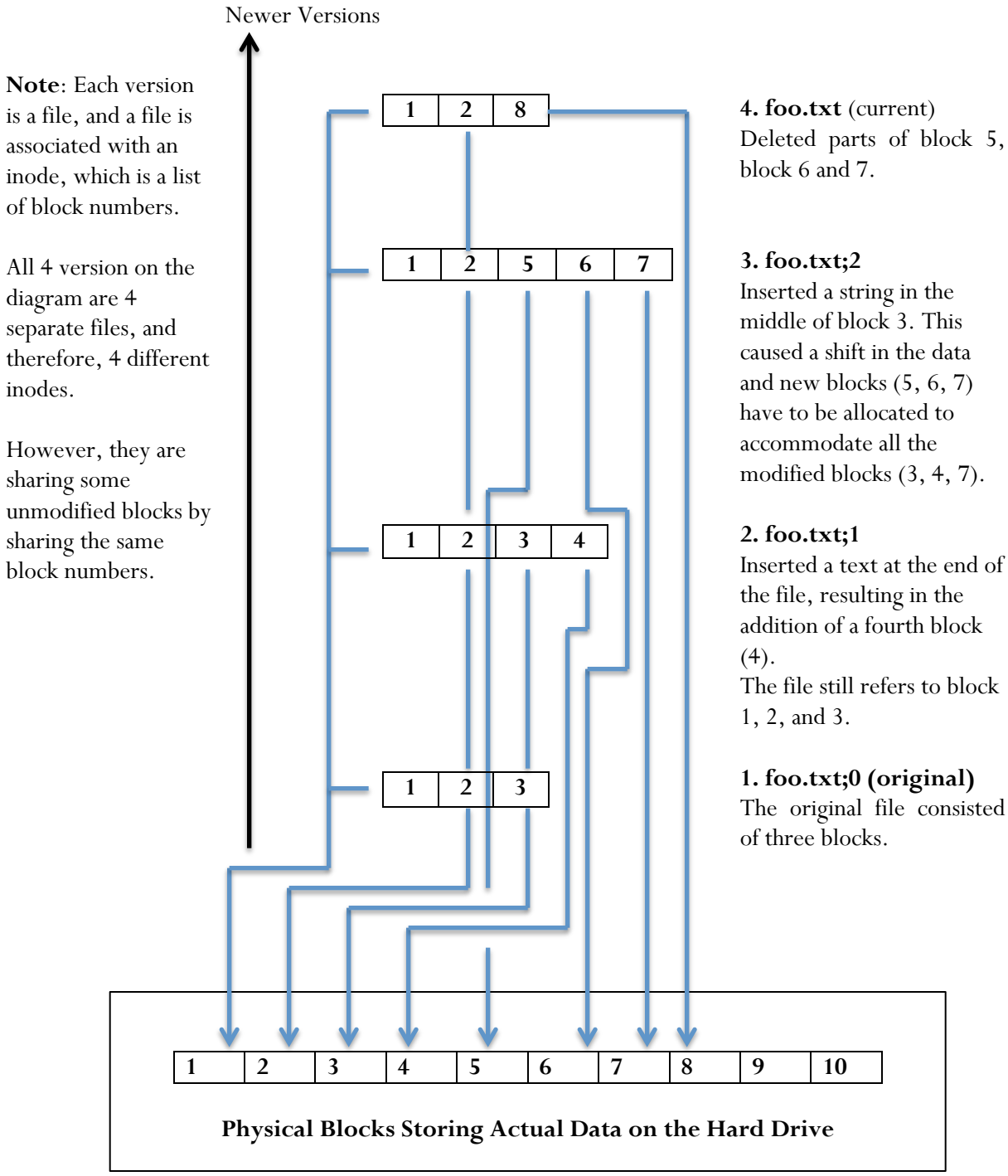

Figure 1: The on-disk data structure for snapshotting. Start reading the texts from the bottom. A file is basically a list of block numbers, and the blocks are storing the data. Note how the number of new blocks allocated satisfies the space requirement for all operations.

- An **in-memory** table that stores the modified blocks for each open file.
  Creating a version requires knowing which blocks have been modified. Therefore, we need a data structure to contain this information while the user is modifying a file. When a version of a file is created, the table entry will be cleared, necessary blocks will be allocated, and unmodified blocks will be passed onto the version.

The way the data structure is defined fulfills the space requirement. As the user made some changes to blocks in the file (Figure 1), new versions will be created and new blocks representing the changes will be created, but unmodified blocks are shared/referenced by all the versions that contain them. This sharing of unchanged blocks meets the requirements since no space is wasted for copying unchanged blocks.

Operations and Interfaces

The following operations will be supported by the system:

- **Modifying data in a file.** New versions will be created. How often the file is versioned depends on the strategies used as described in the next section.
- **Moving/Renaming/Deletion**. Appropriate changes will be made to the version files, depending on the strategy employed.
- **Reviewing a previous version**. Since each version is a file, all the user must do is to read the file representing the version to review the version.
- **Excluding certain files or directories**. Each directory will have a .version_ignore file that lists all the files in the directory that will not be versioned. Users can then manually edit these files to exclude certain files or directories.
- **Search**. Since each version is similar as a regular file, searches will be efficient as there is no need to expensively reconstruct previous versions.

Note that versioning a directory is not supported to avoid possible complications and edge cases.

Strategies and Policies

Considerations have to be made concerning how frequently a new version should be created, and how many versions to be stored. Possible new version creation strategies include versioning after each write call, time-based, and size-based. Possible strategies include a fixed number of versions, time-based, or size-based. Each strategy has its own benefits and drawbacks. The best solution will likely to be a combination of the above and this will be explored fully in the report.

Conclusion

The above design meets the requirements as outlined in the overview section by utilizing the idea of sharing unmodified block across versions of a file. Many questions remain to be explored, including implementation details which will require the modification of I/O system calls, methods of implementing the file-changed blocks table, and searching algorithms.