SongFS: A Music Based Filesystem Interface

By Elaine Guo, Tim Geissler, Alexis Cruz-Ayala

Final Project Report

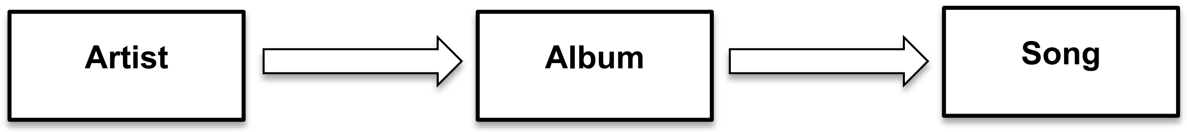
ECE 566, Spring 2024

Contents

Introduction

As consumer storage density continues to grow exponentially, cluttered directories and mismanaged files become an increasingly significant barrier to file organization and overall ease of use. This paper proposes a FUSE-based filesystem interface which automates routine organization and file sorting and provides an example in the form of SongFS - a music lover’s filesystem.

SongFS is a read-only FUSE filesystem which parses MP3 files from a source directory and presents a consistent, organized interface to access automatically organized songs. SongFS indexes files based on their metadata and generates the following directory hierarchy:



Activating the SongFS Python script on a cluttered and disorganized MP3 library will mount a consistent, indexed MP3 file structure to the client’s filesystem, and all sorting and dynamic updates are handled automatically. Missing and broken metadata is handled consistently, and the user is presented with an interface to easily interact with their music library.

Design

SongFS uses the PyFUSE interface to develop a user space filesystem. FUSE handles the ordering of files based on artist, album and song name by processing the ID3 headers of the MP3 files in the root directory. After scanning this metadata, a Python script generates a data structure containing the sorted information in the same structure it will appear in the filesystem. The FUSE module receives this data, and generates directories and files as seen in the figure below:

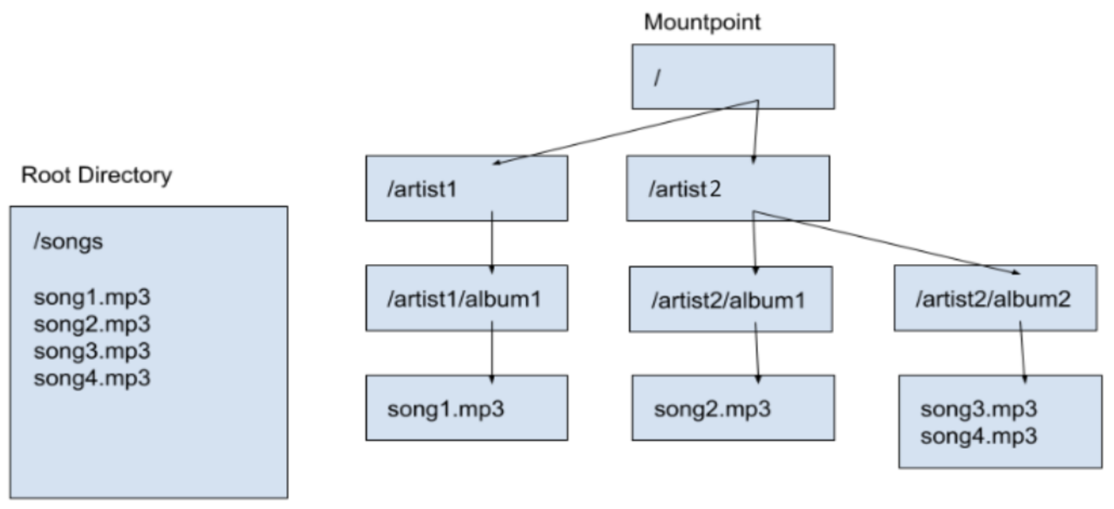


Figure 1: Diagram of how songfs would work: a root directory containing mp3 files can be mounted, splitting it into artist/album/song

The following protocols are implemented if FUSE detects erroneous metadata:

- If a file contains missing ID3 headers, it is placed in the “Unknown Album” or “Unknown Artist” directory.
- If the song name is missing, then it is replaced with the original filename.
- If song names are duplicated, then an integer is appended to the end of the song or filename for additional copies. (For example, “Here Comes The Sun.mp3” & “Here Comes The Sun 2.mp3” would appear in the mounted directory)
- If a file contains erroneous/broken metadata, it is placed in an “Excluded” directory in the root folder.

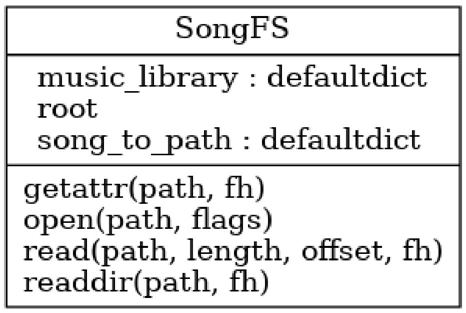
Dynamic filesystem updating is achieved using Py-Inotify. Users can add or delete files in the root directory and the corresponding updates will appear in the mounted directory.

Implementation

SongFS is implemented in Python, and is centered around two independent modules:

1. **MP3 Scanner**
    - The scanner module recursively reads all MP3 files in the source directory and scans their ID3 metadata tags.
    - These tags are assembled into a Python dictionary which mirrors the filesystem hierarchy which is presented to the user.
    - The scanner handles missing metadata tags and other edge cases, so that the dictionary passed to the FUSE module is complete and correct.

2. **PyFUSE Filesystem Layer**
    - PyFUSE is used to present a user space filesystem to mount on the client's filesystem.
    - The SongFS class overrides the necessary PyFUSE methods to implement the required functionality.
    - SongFS receives the metadata dictionary from the Scanner, assembles the directories and files accordingly, and mounts the resulting file tree.
    - This read-only filesystem provides an abstraction layer and does not duplicate song files or write any new directories to disk.


SongFS
music_library : defaultdict
root
song_to_path : defaultdict
getattr(path, fh)
open(path, flags)
read(path, length, offset, fh)
readdir(path, fh)

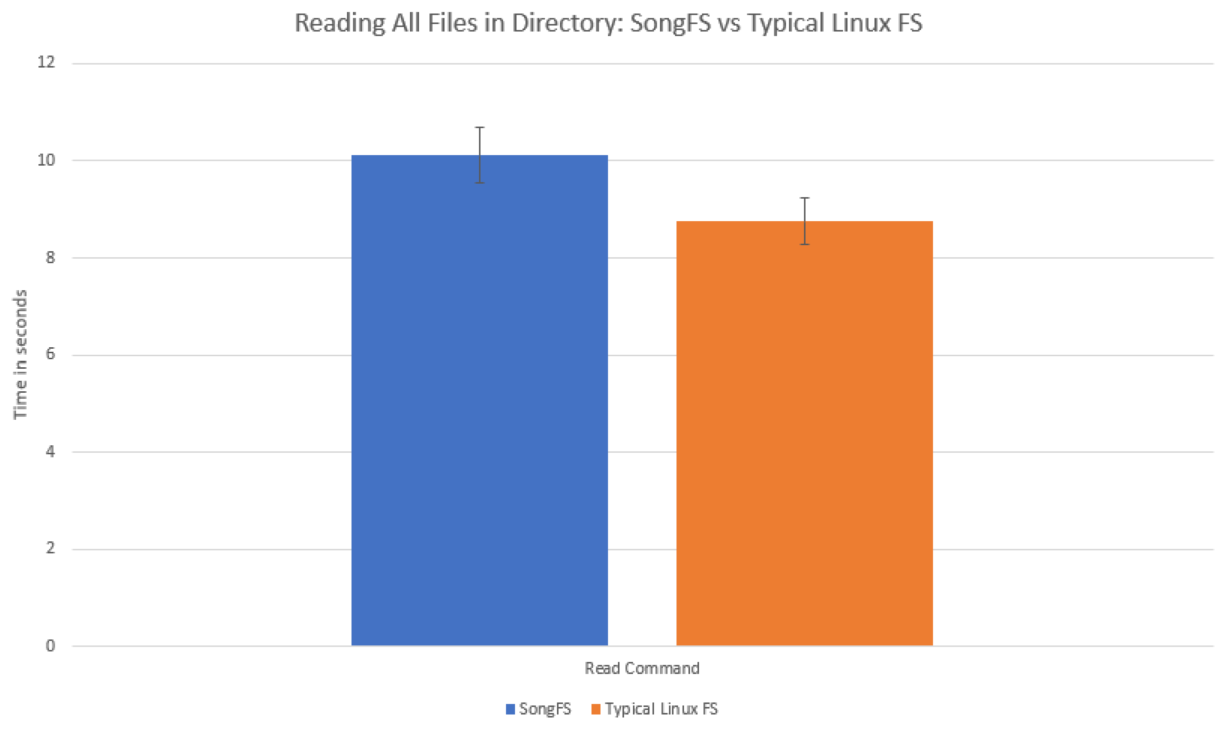
Py-Inotify is used to listen for file changes in the source directory and triggers a rescan of the source files and a refresh of the FUSE filesystem. This enables dynamic updates of the root directory to reflect automatically in the mounted SongFS directory.

Evaluation

Benchmark Design

The core functionality of SongFS is more dependent on correctness than traditional performance metrics. However, a set of simple throughput benchmarks were designed to evaluate filesystem read speed and determine overall system overhead. The benchmarking script (shown below) times a recursive scan using `cat`, and SongFS filesystem read times are compared against that of a standard Linux ext4 filesystem.

Benchmarks



Each filesystem test is an aggregate of 13 independent trials of recursively reading the entirety of a sample music library. Error bars are provided to represent the standard deviation between each set of trials. The ‘Typical Linux Filesystem’ test is run on a standard music library directory with no additional software or interface layers, whereas the SongFS test is run on the mounted SongFS filesystem.

As expected, the additional functionality provided by the SongFS system introduces some overhead to the filesystem and is approximately 14% slower than a filesystem with no intermediary layers. However, the performance impact of PyFuse and file indirection will likely have minimal impact on typical usage patterns – especially in the context of expected use of a local digital music library. In the benchmark above, SongFS exceeds 100MB/s in recursive read throughput, and should be capable of accommodating even excessively large music libraries with minimal impact.

It should be noted that a `tar`-based benchmark (the initial method of testing) which writes the output to `/dev/null` would not accurately test the read speeds of the system, as the `tar` executable detects null destinations and skips the instruction.

Correctness Testing

Correctness tests are used to ensure that the filesystem works as expected and returns valid MP3 file data. These tests are either unit tests (testing small components or submodules) or integration tests (testing a larger system or combination of submodules). To ensure correctness, the following attributes are tested:

1. **MP3 Parsing**
    - The Scanner module is tested against real-world and simulated/synthetic MP3 libraries, and the dictionary object it generates is compared against an expected version to ensure correct scanning behavior and edge case handling.
2. **File Integrity**
    - Once the MP3 files are indexed and mounted on the FUSE filesystem, their correctness is tested using `ffmpeg`, a common terminal-based media playback tool which detects corrupted MP3 data, or the built-in VSCode media player.

Race Condition Testing

Implementing the py-inotify dynamic updating feature introduces additional synchronization issues and requires additional testing. Specifically, it was found that py-inotify would trigger a rescan before the metadata of a new file was fully copied, and this would lead to invalid behavior and system errors. A test was designed to slowly copy the ID3 header data of a new MP3 file to aid in catching when the py-inotify trigger was occurring. As a result of this test, the synchronization issue was found, and the event listening logic was modified to trigger only once the ID3 header was fully copied; thus ensuring a correct and complete metadata scan.

Conclusion

The SongFS project successfully demonstrates the merits of a user-space filesystem abstraction layer which automates sorting and indexing tasks and simplifies the ownership and management of large file libraries. SongFS successfully organizes source files, handles edge cases in missing metadata, and implements dynamic event listening and filesystem updates in real-world usage patterns. This Python implementation demonstrates acceptable overhead compared to the client's direct filesystem and could be expected to perform adequately for increasingly large file libraries.

This design offers several avenues for further exploration, including audio fingerprinting and automatic file transcoding. This concept could also be extended to support a wide variety of file types and use cases, including multimedia files, executables, documents and more. Finally, further performance improvements could be realized by rewriting this implementation in C using standard FUSE for less execution overhead.

As consumer storage density continues to grow exponentially, the value, ease of use, and simplicity offered by filesystem interfaces like SongFS are likely to become increasingly compelling, and we hope to see further innovation and application of this technology.