DEPARTMENT OF COMPUTER SCIENCE

Development of a Tag Based FUSE Filesystem

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Wednesday 3rd May, 2023

12586 words

Abstract

My project aims to create a tag-based file system using the FUSE technology for Linux. Tags are small pieces of metadata attached to file paths. The user is then able to query a tag database through a virtual file system to receive a collection of all files in system that match a certain query. The project additionally aims to demonstrate my capability to develop robust software following professional software development practices. These practices include modular design, automated testing, static analysis etc.

- Designed, developed and tested a queryable FUSE file system in the Rust programming language using professional software development practices.
- Implemented additional features to make the project suitable for the target audience of technical Linux users.
- Evaluated both the features of the software package and how professional practice contributed to creating a robust system.

Contents

List of Figures

List of Tables

Ethics Statement

This project did not require ethical review, as determined by my supervisor, Dr. David Bernhard.

Supporting Technologies

I used the following free software to help me with this project.

- The Rust Programming Language (https://www.rust-lang.org)

  Various Rust libraries with the most important being:

  - rusqlite (https://crates.io/crates/rusqlite) :
    Rust wrapper library for the database library SQLite (https://sqlite.org).
  - fuser (https://crates.io/crates/fuser) :
    a Rust reimplementation of libFUSE (https://github.com/libfuse/libfuse/)

  All Rust libraries used can be found in the dependencies section of `Cargo.toml` in Appendix A.

Notation and Acronyms

API	:	Application Programming Interface
ASAN	:	Address Sanitiser
BNF	:	Backus-Naur Form
CLI	:	Command Line Interface
CRUD	:	Create Read Update Delete
DSL	:	Domain Specific Language
FFI	:	Foreign Function Interface
FS	:	File System
FUSE	:	File System in Userspace
GC	:	Garbage Collection
GUI	:	Graphical User Interface
HDD	:	Hard Disk Drive
HTML	:	Hyper Text Markup Language
SQL	:	Structured Query Language
TDD	:	Test Driven Development
UB	:	Undefined Behaviour
UBSAN	:	Undefined Behaviour Sanitiser
VFS	:	Virtual File System

Chapter 1

Introduction

Organising files can be a tedious, error-prone, and unsatisfying activity. The file system has survived to this day mostly unchanged from its initial origins [21] after being inspired by a real physical paper filing system. The hierarchical structure enforced by the file system can feel confining and inflexible. My project aims to attempt a different file management strategy, whilst still maintaining compatibility and interoperability with the normal hierarchical method. The file management strategy of my project is based upon "tagging" files (more specifically file paths) with small items of metadata known as "tags" [22]. A tag can be either a simple string such as "favourite" or it may consist of a tag name and a tag value for example "year=2019". These tags may then be used as parameters in a search to find files that satisfy certain criteria. Examples of searches could include, finding all the files with the "year=2019" tag, but without the "favourite" tag.

The project maintains compatibility with the traditional hierarchical file system structure by exposing a virtual file system using the FUSE technology available for Linux [2]. By using this file system, the user may perform arbitrary queries against tagged files in the system by specifying their query as a file path. They may also lookup a particular file by name to discover what tags are associated with it. Using a virtual file system approach allows the user to access their files using the tag based methodology from within all the normal software applications and operating system components that they are familiar with. The integration of the tag based strategy with the traditional hierarchical method, is a much more feasible way of adopting tag based file organisation compared to requiring the operating system and every user program to be made aware of and updated to handle the new paradigm.

The intended users of the software are technically capable users who already know their way around and are comfortable on the command line. The software provides a command line interface alongside the virtual file system. This CLI tool can be used to integrate the tag-based file system with the user's existing CLI workflow and follows common CLI conventions to provide immediate familiarity for the targeted technical users.

By using the project, the intended users will be able to manage their files in a more flexible way, whilst also being able to attach extra metadata to a file path. Additionally, the more powerful querying abilities of the software will allow users to find their files more easily, and may even allow the user to be able to identify common properties between files they previously believed unrelated.

Whilst developing the project, I made use of several software development techniques such as automated testing, creating documentation, and static analysis. In later sections I will justify their usage and evaluate how they helped me to develop a robust and maintainable system.

The specific aims for my project are as follows:

- Discuss and justify the use of particular appropriate technologies for building the software.
- Design and implement a queryable tag-based file system using the FUSE technology by following professional software development strategies.
- Evaluate the software and how using professional software development techniques contributed to the success of the project.

- Explore potential improvements that could be made to the project itself and to the development techniques used to create the software.

Chapter 2

File Systems

The file system is a fundamental abstraction for almost all modern operating systems. The abstraction allows the operating system and user programs to interact with a variety of devices and services through a single unified interface [23]. The purpose of a specific file system is to implement this common interface for a particular underlying data store. The exact nature of the data store is not important to the consumer of the file system's interface. The underlying data store is commonly a physical storage device such as a hard disk drive (HDD). However, it is important to note that this is not always the case and there is no requirement for a file system to manage a physical storage device. File systems that depend on another file system as their underlying data store are known as a VFS (virtual file system) [6].

The high level filing cabinet metaphor that is built on the low level interface provided by a particular file system implementation will be familiar to most computer users. The metaphor consists of a filing cabinet containing many named files. These files can contain arbitrary data [23]. However, some of these files may not contain data themselves but rather other files, and in this case they are called directories (or folders in some versions of the metaphor). This nesting of files within directories creates a hierarchical structure where a single root directory (analogous to the filing cabinet) contains an arbitrarily deeply nested series of files and directories. The hierarchy of files within directories can conceptually be arbitrarily deep, but some particular implementations enforce a limit for simplicity. The filing cabinet metaphor is also useful in demonstrating that a file or directory can only be contained within one other directory. Each file or directory (excluding the root directory) must have one and only one parent in the hierarchy [23].

In Figure 2.1 there are two distinct files named `timeline.txt`. It becomes ambiguous to refer to a file only by its name when this is the case. It is useful to use the steps taken to arrive at a file or directory to identify it within the file hierarchy. For example, the steps taken to arrive at `timeline.txt` would be `/` then `documents` then `project1` then `timeline.txt`. This list of steps is known as a path and can be written as a `/` delimited list of the steps taken to reach a file or directory. Using the previous example the path would be `/documents/project1/timeline.txt`. As a result of the single parent rule, each file or directory has a single path that uniquely identifies it amongst all other files and directories in the hierarchy[1]. This single unique path is known as an absolute path, meaning the steps to arrive at a file or directory are completely specified all the way from the root of the file system. In contrast to this, a relative path is a path that is not specified from the root of the file system, but from some other arbitrary location. For example, the path `project1/timeline.txt` is a relative path that is specified from the `documents` directory. The directory that a relative path is specified from is known as the current working directory [23].

The strict hierarchical nature of the file system falls apart with the addition of links to the file system. There are two types of links in UNIX file systems: hard links [15] and symbolic (or soft) links [19]. Symbolic links (often abbreviated as symlinks) are a special kind of file-like object that act as a window into another part of the file system. Symbolic links have a name like a normal file, but they additionally have a target which is another path that they act as a window into. For example, you could imagine a symbolic link named `my-link` in the `images` directory with a target of `/code`. The existence of this link would mean that now the path `/images/my-link/main.c` refers to the same file as `/code/main.c` (the same property holds for `/images/my-link/Makefile` also).

[1] This is not the case when the file system contains symbolic links.

A hard link is a reference to a piece of data. Hard links are very similar to regular files, because regular files are in fact hard links. A hard link references an inode [13] by way of an inode number. An inode is a data structure internal to the file system that is used to keep track of metadata relating to some user data. An inode number is a number that uniquely identifies an inode on a particular file system. The consequence of this is that hard links, and therefore regular files, don't actually own their data; they only reference it through an inode number. This means that there can be multiple distinct files that reference the same inode, and therefore are associated with the same data.

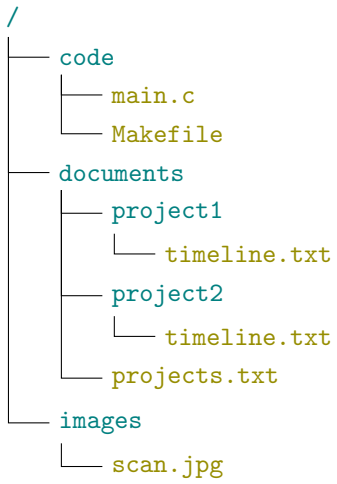

Figure 2.1: Sample file hierarchy with directories in blue and files in green.

2.0.1 Operating Systems and the File System

The primary way most users interact with file systems is through a GUI (graphical user interface) file manager such as `thunar` or `nautilus`. However, it is also possible to manage a file system without a GUI and use CLI (command line interface) tools such as `ls` or `mv`. Both GUI file managers and CLI tools utilise the low level interface exposed by the operating system and implemented by a particular file system. The traditional way to use this interface is via the platform library exposed by the operating system[2]. On UNIX-like operating systems this library is `libc` [14]. There are multiple qualifying implementations for `libc` on Linux with `glibc` being the very common, but all implementations of `libc` must conform to the standard interface. The overwhelming majority of programs that run on UNIX-like operating systems use `libc` to help them interact with the platform. The operating systems' `libc` contains functions that require access to the file system (such as `stat` [18]). To get access to the file system the `libc` function will execute a system call [20] (often called a syscall). A system call is a way for an unprivileged user program to ask the operating system to perform an action on its behalf. When executed, a system call performs a context switch, by saving the state of the current process, and jumps into kernel code[3]. The kernel is the part of the operating system that, amongst many other functions, responds to system calls. When the kernel responds to a system call that requires accessing the file system, it must first ascertain which file system implementation is the correct one to call for this particular action. On Linux this is the VFS layer in the kernel [6]. Once this is determined, the kernel asks the file system implementation to perform the action specified by the system call. It may also return some data to the calling user program via a buffer or a return value.

2.0.2 FUSE File Systems

Most file systems are built into the kernel; either directly compiled in or as kernel modules. This means they will run in kernel mode (ring 0 on x86 platforms). If code running as part of the kernel crashes or hangs, there is a danger that this could crash or hang the entire system. Additionally, code running in kernel mode has the power to do anything on the system, so a security issue in any code run

[2]On some platform like Microsoft Windows, using the platform library is required because the system call interface is not guaranteed to be stable between releases.

[3]This is not strictly true, some system calls do not require jumping into kernel code.

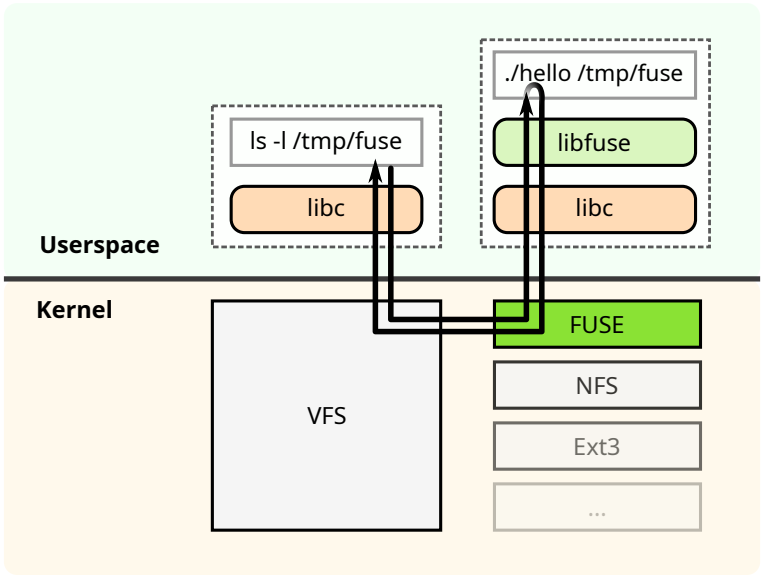

Figure 2.2: A diagram to show the flow of data in a FUSE file system. Taken from [3]

at this level code could result in a compromised system. To reduce the danger of writing a file system, Linux allows file systems to be run as userspace processes using a technology called FUSE [2] (file system in userspace).

Looking at Figure 2.2, there are similarities between the process of accessing a file system built into the kernel (in the diagram NFS and ext3 are used as an example) and the process of accessing a FUSE file system. For both built in file systems and FUSE file systems, the calling program uses `libc` to ask the kernel to perform a file system action, and the kernel dispatches the call to the correct file system implementation using the VFS layer. When the VFS layer requires action from a FUSE file system, it passes the request to the FUSE kernel module. The FUSE kernel module then communicates (the nature of the communication and the medium across which it takes place are discussed in Section 2.0.2.1) the request to the user space process. The user space process can then do whatever it pleases to generate a response which is then passed back through the layers into the kernel and back to the calling user space program that made the request.

FUSE file systems provide more flexibility and safety compared to file systems compiled into the kernel. However, this flexibility comes at performance and hardware utilisation cost with an average of a 31% increase in CPU utilisation when file system requests are routed through the FUSE layer [24]. The performance decrease has a larger variance with some cases having as low as 5% overhead. However, in other cases that are "unfriendly to FUSE" such as concurrent random access to large files can cause an 83% decrease in performance even if optimised [24].

2.0.2.1 Mounting FUSE File Systems

To mount a (non-root) file system means attaching the root of a child file system to an empty directory in another parent file system. The two file systems remain entirely independent; their connection is only simulated using the kernel's VFS layer. However, to other programs it appears that the contents of the child file system are available as a sub-directory in another parent file system. This allows the composition of many independent file systems into one virtual file system with the kernel handling dispatching requests to the correct underlying file system.

On Linux mounting a file system is a restricted operation that can only be performed by the root user [16]. However, FUSE has the explicit goal of "allowing secure, non-privileged mounts" [2]. To allow FUSE file systems to be mounted by non-privileged users, FUSE provides a mounting utility `fusermount` which is installed with the setuid [17] permission set to the root user. A binary with this permission can be run by a normal user, but whilst it is running it has all the privileges of the root user. This allows a normal user to mount a FUSE file system.

After the FUSE file system has been mounted there needs to be a channel for communication between the FUSE kernel module and the user space program managing the file system. This channel is the `/dev/fuse` pseudo device file. However, reading and writing to this file is a privileged action so

`fusermount` (with its elevated permissions) is responsible for opening a handle to `/dev/fuse` and then returning it to the user space file system. `fusermount` returns this handle over a socket given to it via an environment variable at launch [4].

Chapter 3

Memory Safety and the Rust Programming Language

To understand why I chose to use the Rust programming language to develop my project, I believe it is important to briefly explain the defining feature of Rust: memory safety. In this chapter I will attempt to give a brief overview of the Rust programming language and how it ensures memory safety without the use of a runtime garbage collector. Also I will talk about why memory safety is important, and the problems caused when it is not adhered to.

Memory safety is a property of a program that means that the program does not under any circumstances have any bugs relating to invalid pointer access or accessing uninitialised data [11]. These two classes of bug contain many common sources of security issues in programs such as use-after-free, buffer overflow, and invalid pointer dereferences. In fact memory safety related bugs are some of the most frequent security bugs in many widely used software packages such as Chromium [5], with the project estimating that 70% of their security bugs are related to memory safety issues.

One way of ensuring memory safety is through the use of a runtime garbage collector. This, in addition to disallowing pointer arithmetic, ensures memory safety by not allowing the user program to directly allocate and free memory itself, and by not allowing access to invalid memory locations. However, for some programs runtime garbage collection is not suitable for performance reasons. For example, real time systems with hard deadlines may not be able to use a garbage collector due to the unpredictability of the moment of and the duration of garbage collection pauses. It is in these case where a language like Rust can prove its value by having fast and consistent performance as well as memory safety.

Rust ensures memory safety by refusing to compile programs that do not satisfy a particular set of rules at compile time. The sub-system in the Rust compiler that checks these rules against the given code is called the "borrow checker". The borrow checker works via the concept of "ownership" [8]. Ownership means that every value in a Rust program has one and only one owner. When the owner is no longer accessible i.e. it has gone out of scope, then the value it owns is freed (or in Rust parlance "dropped"). Ownership of a value can be transferred or handed over to another owner, but once this has occurred the original owner is unable to access the value therefore preserving the single owner rule. Ensuring that a value only has one owner in this way is enough to be able to statically insert calls to free values when they go out of scope. Transferring ownership of a value can be cumbersome when it is only temporary, to reduce the burden of always transferring ownership Rust has the concept of "borrowing" a value and receiving a "reference" to it. A reference is similar to a pointer in other languages, but it is additionally constrained by the fact that all references must always be valid at all times. Valid meaning that the pointer is not null and that the pointer has not been freed or deallocated. There are two types of reference in Rust: mutable and immutable. To ensure memory safety Rust enforces the rules that there can be either one mutable reference or many immutable references at any given time, but never both.

An example of memory unsafe code that Rust prevents is shown in Figure 3.1. In the first line a vector is allocated and initialised with the values 1, 2, and 3. The vector is given the name "data" and prefixed with the "mut" modifier to state that this value is allowed to be mutated (in Rust values are immutable by default). In the second line we borrow from the vector to obtain an immutable reference to the first item in the vector and then store the reference in the "first" variable. Then on the third line the value 4 is pushed onto the end of the vector. The "push" method requires mutable access to the vector

to perform its modification to the vector[1]. On the final line, the reference stored in the "first" variable is dereferenced to access the first value of the vector and print it out. This code is disallowed by the Rust compiler because, on the third line, the code attempts to acquire a mutable reference to the data vector whilst an immutable reference is already held, thereby breaking the rules of the borrow checker.

Additionally, one of the reasons for the enforcement of the "mutable xor immutable" reference rule is illustrated well by the example in Figure 3.1. If this code were allowed to compile then, in certain situations, the code could attempt to read data from memory that has been freed. This situation could occur if upon pushing to the vector, there is no more space in the block of memory underlying the vector. Then to make more space for the new element, it becomes necessary to grow the vector by allocating a new larger chunk of memory and copying the existing elements to the new memory. After the reallocation, the old chunk of memory is freed and potentially released to the operating system. Now after the push call completes, the reference held by the "first" variable is still pointing to the location of the old memory allocation which has been freed. When the reference stored in "first" is dereferenced it could cause a segmentation fault or more insidiously it could only sometimes cause a segmentation fault, making it very difficult to debug. This is an example of a use-after-free bug, and one of the ways that Rust prevents them. Interestingly, the code compiles and runs correctly if the last line is removed. This is because the Rust compiler is clever enough to be able to notice that the reference stored in the "first" variable is not used on or after the third line. Therefore, it is able to drop the borrow before the third line thereby allowing the mutable borrow to take place on line three as there are no other references to the "data" vector at that time.

```
1 | let mut data = vec![1, 2, 3];
2 | let first = &data[0];

3 | data.push(4);
4 | println!("The first element is: {first}");

-----------------------------------------------
error[E0502]: cannot borrow ‘data‘ as mutable because it is also borrowed as immutable
let first = &data[0];
             ---- immutable borrow occurs here

data.push(4);
^^^^^^^^^^^^ mutable borrow occurs here
println!("The first element is: {first}");
                                 ----- immutable borrow later used here
```

Figure 3.1: Example memory unsafe code disallowed by Rust's borrow checker with corresponding error message. Taken and modified from Section 3 in [9].

The rules of Rust combine to create a memory safe environment for writing programs that perform real work, but sometimes it is not possible to express a particular program in this safe environment. This can even been seen in the small example used previously in Figure 3.1. On the second line a reference is taken to the first element in the "data" vector, under the hood this is classic C style pointer arithmetic - adding an offset to a base pointer. However, from the rules of Rust we know that all references must always be valid, so it is necessary to ensure that the base pointer combined with the given offset does not overflow or underflow the buffer underlying the vector. This can of course be easily checked by ensuring the offset is less than the number of items that may fit into the buffer. However, even with the check, at some point dereferencing the calculated pointer, which is an inherently unsafe operation, has to take place (if the value is going to be used at all). This means that Rust needs a mechanism to express unsafe operations within the language, if it is to be possible to implement a container like a vector in the language. Since the Rust standard library (which includes lots of container data structures including a vector) is implemented in Rust itself, this must and in fact does exist.

Rust has the concept of the "unsafe" keyword that can be attached to certain Rust constructs such

[1]This can be seen explicitly in the type signature of the "push" method on vectors "`pub fn push(&mut self, value: T)`". The key is the "mut" keyword after the `&` to indicate a mutable reference is required.

as blocks and functions. Adding the "unsafe" keyword to a Rust construct grants the programmer additional capabilities that are not possible in the normal safe Rust context. The main capabilities granted by unsafe Rust are the ability to dereference raw pointers and the ability to call other "unsafe" functions[2] [9]. However, it is now up to the programmer to manually ensure that undefined behaviour (UB) is not invoked within the unsafe context, and additionally to ensure that any of the normal Rust rules regarding ownership and references are upheld. If these rules are not upheld then Rust's memory model has been violated and UB is likely to follow. An example usage of an "unsafe" block is in Figure 3.2 where an unsafe block is used to implement an operation that would be impossible in entirely safe Rust. The example function can be used to obtain the nth byte in a byte array in a safe way. Providing a safe wrapper around unsafe functionality is a very common pattern in Rust, and is what allows useful programs to be written in a safe environment. The reason why the unsafe block is required in the example function is because the `get_unchecked` function is marked unsafe at its declaration, and then the raw pointer returned by the `get_unchecked` function is dereferenced with the `*` operator. Additionally, the if statement in the function is used to uphold the necessary invariant (i.e. is the given index in range) to make this function "sound". Soundness being the property that it is impossible for the caller to invoke UB by using this function.

```
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx < arr.len() {
        unsafe { Some(*arr.get_unchecked(idx)) }
    } else {
        None
    }
}
```

Figure 3.2: Example function that necessarily uses an unsafe block. Taken from Section 1.3 in [9].

[2]There are more capabilities than these, but they will not be discussed here as they are less important.

Chapter 4

Project Execution

My project is a tag-based FUSE file system (from now on creatively referred to as "`tagfs`"). `tagfs` is written in the Rust programming language and relies on the "fuser" library to communicate with and decode messages from the FUSE kernel module. Tagfs also makes use of the SQLite database library through the Rust wrapper interface provided by "rusqlite". Specific features of tagfs are discussed in more detail in Section 4.2.

4.1 Design

4.1.1 Intended Users

When developing a software project it is important to keep the intended users of the project in mind as much as possible throughout the development of the project. This is to ensure that the project does not lose focus and make poor design decisions that will negatively impact the intended user base. For `tagfs` I decided that my intended users would be technically minded and capable users that are comfortable and feel at home on the command line. By aiming for technical users, I was able to focus on developing the main functionality of the software rather than spending time developing a user interface that is more suitable for less technical users. However, this did not mean that I did not want to provide a good user experience for my user base. To provide a good user experience for technical users, I decided to create a command line interface to interact with `tagfs`. Additionally, I wanted to ensure that the CLI followed common interface designs and patterns seen in other CLI programs. This is to help the user feel at home quickly if they have experience with other popular CLI programs. Therefore, I decided to create a subcommand style CLI similar to `git` and `cargo` (the Rust build tool).

Additionally, I decided to restrict my user base to Linux users. This meant that the development process would be much easier due to only having to develop for and test on one platform. This way also meant that I could focus on the FUSE technology rather than having to create a wrapper between FUSE and the Windows equivalent. I felt that restricting the user base in this way would not be too confining because I thought that many of the technical users that I was already targeting would be using a Linux platform anyway.

4.1.2 Technology Choices

In my experience of creating software for myself, I have found that choosing suitable technology is a vital component of a successful software project. Unsuitable technology can result in a project being much more complex, brittle, and unmaintainable than it otherwise could have been.

I chose Rust as the implementation language for my project. One of the factors that pushed me towards Rust was my prior experience creating small hobby projects using the language. In these smaller past projects I had found the features of Rust, such as its strong static type system and focus on correctness, to lend themselves to producing robust and maintainable systems. Additionally, the potential performance of a Rust solution (due to Rust's lack of a runtime garbage collector) attracted me, because file system code requires good performance to avoid frustrating the user or slowing down other programs that depend on it. However, I did have some concerns about the choice of Rust. My main concern was, due to the fact

that Rust is a relatively young programming language[1], there would not be many high quality libraries available. It was a pleasant surprise then that when browsing the Rust community repository crates.io[2], I found many projects with active communities that appeared to be reasonably mature and usable. My heuristic for identifying these libraries is based on the number of downloads on crates.io and the last commit date on GitHub. I found that libraries with a high number of downloads tended to represent the best the community had to offer in that particular category. Libraries with lots of recent commits tend to have more active communities and that often means that there are people asking questions and receiving answers which is useful to fill in any gaps in the documentation.

The other programming language that I considered for my project was C. The main reason that I considered C was the fact that the reference implementation of a FUSE library (libfuse[3]) is written in C and most tutorials and documentation assume the use of C. Using C would have meant that I would have a much wider array of examples to help me implement `tagfs`. However, I decided against the use of C because I felt that I would miss the higher level of abstraction provided by other languages with features such as generics and tagged unions built into the language. Additionally, I did not want to deal with painfully manual memory management and I did not feel confident in producing code without memory safety bugs such as use after free and segmentation faults. These memory safety bugs can be somewhat prevented through the use of modern sanitisers such as ASAN and UBSAN, but these sanitisers are only effective at runtime and only if the pathological case that causes the bug appear. Rust on the other hand can catch these bugs at compile time and prevent them from ever occurring.

4.1.2.1 Data Storage Back-end

Early on in the planning phase of my project I knew I was going to need some mechanism to persist data to the file system between mounts of the virtual file system. I knew my data would be at least vaguely relational, because I knew that I needed to model the relationship between a file system path and the tags associated with it. This is what initially drew me to a database solution rather than a simpler flat text file. However, the main reason I chose to use an SQL database was the ability to constrain my data and enforce invariants at the database level. While it is possible to do this with a flat file approach, I believed that this would be difficult to implement and less performant than using a database. Additionally, it made sense to use an off-the-shelf solution because the particular underlying data store is not directly relevant to the goals of the project and therefore I could have spent a lot of time reinventing the wheel rather than creating `tagfs`.

Once I had decided that I was going to use a database, my choice of which database implementation to use was very easy. SQLite is the most deployed database software on the planet [10], and it is perfect for my use case of a local data store. It is fast, battle-tested, and well documented. Unsurprisingly, there is a Rust wrapper library for SQLite[4]. Rusqlite packages up SQLite into a more Rust-like interface suitable for use from entirely safe Rust code.

4.1.3 Database Schema Design

After having decided to use an SQL database, I needed to design a schema for each table in the database. I settled on two tables initially: one to store the tags and one to store the mapping of tags to paths. The second table contains a foreign key field that references a tag in the first table. The reason I decided to have a table entirely dedicated to storing tags was to enforce a consistent usage for each tag. This means that for every usage of a tag, the tag must always be given with or without a value, but it must be consistent across all uses of a tag. If I did not have a dedicated table to restrict each tag to one usage style, then each usage of a tag could have potentially different behaviour which would be confusing.

I also use the "trigger" feature of SQL to ensure that tags that are no longer used by any path are removed from the database (the trigger code is shown in Figure 4.1). This helps to avoid the problem of having unused tags in the output of the CLI and file system. The trigger is run whenever a row is deleted in the "TagMapping" table. When the trigger is run a "DELETE" SQL command is issued that searches for any usage of the deleted tag in the "TagMapping" table, if there is no usage found then the tag is removed from the "Tag" table.

[1] Rust 1.0 was released in 2015 (https://blog.rust-lang.org/2015/05/15/Rust-1.0.html) most other widely used programming languages were first released at least 20 years earlier.

[2] https://crates.io/

[3] https://github.com/libfuse/libfuse

[4] This highlights another important Rust feature: its foreign function interface (FFI)

```
CREATE TRIGGER IF NOT EXISTS
RemoveUnusedTags AFTER DELETE ON TagMapping
BEGIN
  DELETE FROM Tag
  WHERE
    Tag.TagID = OLD.TagID
    AND NOT EXISTS(
      SELECT TRUE
      FROM TagMapping
      WHERE TagMapping.TagID = OLD.TagID);
END
```

Figure 4.1: SQL trigger code to remove tags that are no longer referenced from the database.

4.2 Features

Tagfs exposes two different interfaces for the user to interact with: a CLI and a file system interface. The CLI uses both the classic UNIX style options (`--help` or `--database`) and a more modern (at least as far as CLIs go) subcommand style interface similar to the `git` CLI.

```
$ tagfs help
Tag-based virtual file system backed by SQLite.

Usage: tagfs [OPTIONS] <COMMAND>

Commands:
tag             Apply a tag to a path
untag           Remove a tag from a path
mount           Mount the filesystem [aliases: mnt, m]
tags            Display tags associated with a path
query           Query the database [aliases: q, search]
autotag         Autotag a directory tree or file
prefix          Modify the prefix of paths in the database
edit            Edit the tags database using a text editor
stored-queries  List, create and delete stored queries in the database
help            Print this message or the help of the given subcommand(s)

Options:
    --database <database>  Path to database to use or create
-h, --help                 Print help
-V, --version              Print version
```

Figure 4.2: Help for tagfs. Shown when the `tagfs` or `tagfs help` command is run.

Each subcommand acts as if it is its own self contained program with its own options and arguments. However, there are some options that all of the subcommands share, these can been seen at the bottom of Figure 4.2. The `--database` option specifies the SQLite database to use for a specific subcommand. If this option is given, but the database does not exist at the given location then a new database is created at the specified location. If this option is not given at all then, following the XDG base directory specification [12], a sensible default location is found. If a database is found at this location then it is used, but if there is no database then an empty one is created.

The `tag` and `untag` subcommands are both straightforward CRUD actions in the database, create and delete respectively. Both subcommands take a path and a list of tags. The `tag` subcommand tags the given path with each tag in turn. The `untag` subcommand removes each given tag from the given path. If the list of tags given is empty then `untag` removes all tags from the given path.

The `mount` subcommand mounts the FUSE file system at the given mount point. The FUSE file

system is discussed in more detail in Section 4.2.2.

The `tags` subcommand lists all of the tags stored on a given path. The information provided by this command is also exposed through the FUSE file system. The situation is similar for the `query` subcommand. The subcommand takes a query and lists all of the paths that match. Queries are discussed in more detail in Section 4.2.1.

The `autotag` subcommand takes a path. If the path is a directory then the subcommand will recursively search the directory for all its descendants that are files. Each of these files is checked to see if they are of a type that the `autotag` subcommand recognises. If `autotag` recognises a file's type then it will attempt to extract or generate tags based on the file's content. For example, if `autotag` comes across an MP3 file it will try and extract the file's metadata tags and build tags based on them. The tool is able to extract album, album-artist, title, and date information from music files. The software also recognises image files and can extract the time the image was taken from the file's metadata. Additionally, the tool can also handle video files. If the tool detects a video file (matroska or MP4) named with the common convention of "`Title (Year)`" then it will assume the file is a film and will search TMDB[5] for related metadata. The tool uses the TMDB API to access information such as the film's genres and the top billed actors and will then store this information as tags in the database.

The `prefix` subcommand is used for searching and replacing components of the paths stored in the database. It takes two strings as arguments: the term to search for and the term to replace it with. It then scans the database for any paths matching the search term and replaces the match with the given replacement text. This subcommand is designed to reduce the burden of updating the database after you have moved some files. For example, you might have moved your music directory to another location on your system, and using the prefix command you can update all of the paths for your music files to use the new location in one go.

The `edit` subcommand is used to make fine-grained edits to the database with the comfort of a familiar interface. This subcommand dumps the contents of the database to a temporary file in a human-readable format. Then the subcommand invokes the user's preferred editor[6] on the temporary file and waits for the editor process to terminate. Once the editing session is complete, the `edit` subcommand checks to see if the user has made any changes to the temporary file and, if they have, commits them to the database.

```
--------
/path/to/tagged/file
tag1=value
tag2
tag3=long\ value
--AUTO--
autotag=autovalue
--------

--------
/path/to/another/tagged/file
tag1=other-value
tag2
--AUTO--
another-autotag=autovalue
--------
```

Figure 4.3: Example of the `edit` subcommand editing session.

Figure 4.2 shows a potential editing session. The file is split into blocks delimited by eight dashes. Each block represents one path in the database. The path is specified as the first non-empty line inside the block. Each non-empty line after that specifies a single tag to be applied to the path given at the start of the block. The tags are read in the exact same way as a query (more detail about query syntax

[5] https://themoviedb.org

[6] It finds the user's preferred editor by first reading the `$VISUAL` environment variable and failing that it tries the `$EDITOR` environment variable before falling back to a default value.

⟨*query*⟩ ::= ⟨*term*⟩ ⟨*whitespace*⟩ ⟨*binary-bool-op*⟩ ⟨*whitespace*⟩ ⟨*query*⟩
| ⟨*unary-bool-op*⟩ ⟨*whitespace*⟩ ⟨*query*⟩
| '`(`' ⟨*maybe-whitespace*⟩ ⟨*query*⟩ ⟨*maybe-whitespace*⟩ '`)`'
| ⟨*term*⟩

⟨*term*⟩ ::= ⟨*tag*⟩ ⟨*maybe-whitespace*⟩ ⟨*comp-op*⟩ ⟨*maybe-whitespace*⟩ ⟨*value*⟩
| ⟨*tag*⟩

⟨*tag*⟩ ::= ⟨*tag-char*⟩ ⟨*tag*⟩ | ⟨*tag-char*⟩

⟨*tag-char*⟩ ::= {Set of non-whitespace, non-control Unicode characters} \ {'`=`', '`<`', '`>`', '`(`', '`)`'}

⟨*value*⟩ ::= '`"`' ⟨*value-quoted*⟩ '`"`' | ⟨*value-unquoted*⟩

⟨*value-quoted*⟩ ::= ⟨*value-char-quoted*⟩ ⟨*value-quoted*⟩ | ⟨*value-char-quoted*⟩

⟨*value-char-quoted*⟩ ::= {'`\"`'} ∪ {Set of non-control Unicode characters} \ {'`"`'}

⟨*value-unquoted*⟩ ::= ⟨*value-char-unquoted*⟩ ⟨*value-unquoted*⟩ | ⟨*value-char-unquoted*⟩

⟨*value-char-unquoted*⟩ ::= { '`\ `', '`\"`', '`\)`' } ∪ {Set of non-whitespace, non-control Unicode characters} \ {'`)`', '`"`'}

⟨*comp-op*⟩ ::= '`>`' | '`<`' | '`=`' | '`==`'

⟨*binary-bool-op*⟩ ::= '`and`' | '`or`'

⟨*unary-bool-op*⟩ ::= '`not`'

⟨*whitespace*⟩ ::= '` `' ⟨*whitespace*⟩ | '` `'

⟨*maybe-whitespace*⟩ ::= ⟨*whitespace*⟩ | ϵ

Figure 4.4: BNF description of the query language of `tagfs`. Note that set notation is used as a shorthand to describe production rules with many terminals. For example the set $\{a, b, c\}$ produces the rule $a|b|c$. There is the additional constraint that tags cannot share the same name as a boolean operator.

in Section 4.2.1) with the additional constraint of only being allowed to contain one tag and an optional value. There is also the possibility of an '`--AUTO--`' heading somewhere in the block. Any tags read after this special heading, are marked as being generated by the `autotag` subcommand. This is useful to separate user defined tags from automatically generated ones. This separation is important so that the `autotag` subcommand knows which tags are safe to overwrite. The user may make changes to any part of the temporary file, including removing entire blocks, removing the '`--AUTO--`' special heading, and changing the path entirely.

The `stored-queries` subcommand is a simple interface to manage the queries that are visible by default in the file system. It allows the creation of new stored queries, the deletion of existing stored queries and the listing of all queries stored in the database. How the stored queries are used and displayed is explained in more detail in Section 4.2.2.

The final subcommand is the `help` subcommand. As seen in Figure 4.2, the `help` subcommand prints out a static message informing the user how to use the software and what options the software has available. The `help` subcommand is also able to take another subcommand as an option and then it will display the specific help and options for that particular subcommand.

4.2.1 Queries

Whilst using tagfs there are multiple contexts in which the user may be asked for a query. These are when using the FUSE file system and when using the `query` subcommand. A query is an expression which any particular path in the database either satisfies or does not satisfy.

At the most basic level, a query can contain a single tag, for example `favourite`. This query matches all the paths in the database that have been tagged with the tag "favourite". This is useful on its own, but not very powerful. However, it is possible to combine multiple tags into a single query with boolean operators. One such operator is the `and` operator. It matches a path if both its operands match. For example, the query `favourite and watched` matches all the paths tagged with both "favourite" and "watched". Additionally, there is a union operator `or` that matches a path if either or both of its operands match. A query can also be inverted by using the `not` operator. For example, the query `not favourite` matches all paths in the database that are not tagged with "favourite". These operators can be combined in many ways to produce arbitrary expressions. For example, the query `not favourite and (not watched or recent)` matches any paths that do not have the "favourite" tag and have the "recent" tag or don't have the "watched" tag. In the previous query parentheses are used to group parts of the expression differently than without. Query expressions follow the conventional precedence rules that specify that parentheses have the highest precedence, followed by `not`, followed by `and`, followed by `or`.

In the database it is possible for tags to be associated with values, so the query language supports matching on a tag's value as well as just the tag's presence. Query expressions can be formed by using a comparison operator along with a tag and a value. The most common comparison operator is `=`, it is used to test whether a tag's value contains a particular substring[7]. For example, the query `title=before` matches any paths that have a tag called "title" with a value containing the string "before". Another comparison operator is `==` (strict or exact equals) this is similar to the standard `=` operator except the given value must be exactly the same as the value attached to the tag in the database for it to match. The final two comparison operators are the less than `<` and greater than `>` operators. These match a path if the path has the given tag and the tag's value is less than or greater than (depending on the operator used) the value given in the query. The ordering used to determine whether a value is less than or greater than another is lexicographical. This can produce behaviour that may be surprising such as the fact that '2' > '10'. These comparison operators can of course be combined with the earlier described boolean operators to create more complex query expressions. The values used with the comparison operators must be escaped properly to ensure the query's semantics. For example, the query `genre=science fiction` is invalid because values are terminated by a space character, so this is parsed as two separate tags "genre" and "fiction". Since there is no comparison operator between them, the query is invalid and is therefore rejected. To achieve the desired meaning the value associated with a tag can be escaped or quoted. For example, to query for a path with the tag "genre" and value "science fiction" the two forms `genre=science\ fiction` and `genre="science fiction"` can be used and both have the same meaning. The query syntax is specified more precisely in Figure 4.2.1 using BNF notation.

4.2.2 File System Features

Once the file system has been mounted using the `tagfs mount` subcommand it can be interacted with as if it were any other file system on the system. Figure 4.5 shows a possible hierarchy for the FUSE file system. Within the root directory (in Figure 4.5 this is `fuse-mount-root/`) a directory is created for each tag in the database. In Figure 4.5 these tag directories are `favourite`, `genre`, `type` and `year`. Each tag directory has different children based on whether the tag is associated with a value or not. If a tag does not have an associated value (this is the case for the `favourite` tag directory in Figure 4.5) then the corresponding tag directory will contain a symbolic link for each path that shares the tag. Each symbolic link points to the tagged path and shares the same name as the last component of the tagged path. An example of one of these symbolic links is `Heat (1995)` in the `favourite` directory. Alternatively, if a tag is associated with a value then the children of the tag directory are the enumerated values associated with that tag. An example of this type of tag directory is `genre` with two associated values represented by the value directories `crime` and `rock`. Each of these value directories contains symbolic links to the paths that are tagged with the tag of the parent tag directory and the value of the value directory. These two types of directories provide an easy to access method of performing simple queries of the database for a single tag. However, it is desirable to be able to perform more advanced queries from the file system.

This is where the query directory (`?/`) is useful. The query directory contains sub-directories representing each of the stored queries saved by the `stored-queries` subcommand. The stored query directories are named with the format `<query-name> @ [<query>]`. Each stored query directory contains the results of executing the stored query the directory represents. In Figure 4.5 the stored query directory is named `favourite-crime` and contains the results of executing the `type=film and genre=crime and`

[7] By default the `=` operator is not case sensitive. This can be adjusted via a flag on the `query` subcommand.

```
fuse-mount-root/
├── ?
│   └── favourite-crime @ [type=film and genre=crime and favourite]
│       └── Heat (1995) -> /film/Heat (1995)/
├── tags
│   ├── film
│   │   └── Heat (1995).tags
│   └── music
│       └── The Smashing Pumpkins
│           └── Siamese Dream
│               ├── 01 Cherub Rock.flac.tags
│               └── ...
├── favourite
│   └── Heat (1995) -> /film/Heat (1995)/
├── genre
│   ├── crime
│   │   └── Heat (1995) -> /film/Heat (1995)/
│   └── rock
│       ├── 01 Cherub Rock.flac -> /music/The Smashing Pumpkins/Siamese Dream/01 Cherub Rock.flac
│       └── ...
├── type
│   ├── film
│   │   └── Heat (1995) -> /film/Heat (1995)/
│   └── music
│       ├── 01 Cherub Rock.flac -> /music/The Smashing Pumpkins/Siamese Dream/01 Cherub Rock.flac
│       └── ...
└── year
    ├── 1995
    │   └── Heat (1995) -> /film/Heat (1995)/
    └── 1993
        ├── 01 Cherub Rock.flac -> /music/The Smashing Pumpkins/Siamese Dream/01 Cherub Rock.flac
        └── ...
```

Figure 4.5: Sample `tagfs` FUSE file system hierarchy with directories in blue, files in green, and symbolic links in pink. An arrow (‘->’) following a symbolic link shows the target that the link is referring to.

```
fuse-mount-root/?/genre=rock
├── 01 Cherub Rock.flac -> /music/The Smashing Pumpkins/Siamese Dream/01 Cherub Rock.flac
└── ...
```

Figure 4.6: Result of looking up an arbitrary query directory with directories in blue, files in green, and symbolic links in pink. An arrow (‘->’) following a symbolic link shows the target that the link is referring to.

```
favourite
genre=crime
type=film
year=1995
```

Figure 4.7: Contents of the file `fuse-mount-root/tags/film/Heat (1995).tags` from Figure 4.5

`favourite` query, which in this example is one path represented by the symbolic link `Heat (1995)`. Stored queries are useful for saving frequently used queries, but it would be very restricting to only be able to access the results of queries previously stored in the system. To make running one-off queries less tedious, the query directory also supports running arbitrary queries by looking up 'hidden' directories[8]. These directories are children of the query directory (`?/`) named after the query they represent. Using the tags established in Figure 4.5 as an example, looking up the directory `?/genre=rock/` would result in a directory containing the results of the `genre=rock` query, which in this case would be a symbolic link named `01 Cherub Rock.flac` pointing to the target `/music/The Smashing Pumpkins/Siamese Dream/01 Cherub Rock.flac`. This is illustrated in Figure 4.6.

The file system also exposes another special directory: the `tags` directory. The `tags` directory provides a method of viewing all of the tags associated with a particular path. Under the `tags` directory a structure mirroring the hierarchy of all the paths in the system is created. Browsing this structure is just like browsing the normal file system hierarchy outside of `tagfs`. However, the last component of the path is replaced by a file that has the same name as the last component of the path with an extra `.tags` extension. An example of this hierarchy can be seen in Figure 4.5. Inside each of these `.tags` files is a list of all the tags on the path represented by the file. For example, the file `fuse-mount-root/tags/film/Heat (1995).tags` contains a list of all the tags on the path `/film/Heat (1995)`. The particular tags for this example are shown in Figure 4.7.

The file system is also able to handle modifications to the database whilst it is mounted. This means that whenever the database is modified, the file system will adjust to the new database state seamlessly. As such, mounting and remounting the file system to handle changes is completely unnecessary.

4.3 Implementation

The `tagfs` project has two main components: the FUSE file system and the CLI tool. There is a significant overlap in functionality between the two components, therefore it made sense to only implement functionality once and share the code between the two components. To do this I developed the shared functionality as a separate library. The CLI program is then a separate binary that is statically linked to this library and the file system is a sub-module within the library. This separation ensures that the CLI can operate on the database and mount the file system, but only through the API exposed by the shared library. Separating the concerns of the software (separating the user interface from the core functionality) also makes the software more easily testable. This is discussed in more detail in Section 4.3.2.

When I began to start implementing `tagfs` (after having done some broad design work) I had to decide which part of the project to work on first. I decided to first work on the part of the project which I anticipated would be the most the difficult. This was the file system portion of the project. I decided to work on, what I thought would be, the most difficult part of the project first to ensure that it was achievable. I did not want to leave a critical, but difficult, part of the project until much later on and find out that it was not possible or that I could not implement it satisfactorily. Following this strategy, I started by implementing a file system that could simply be mounted and contained no files or directories. Then building on this prototype, I created a file system that contained a static list of empty directories. Next I used a hash map structure to represent a file system containing a series of directories that each have their own children (which were limited to symbolic links for simplicity). This iterative development process formed the basis for all progress in the project and allowed me to increase the complexity of the project gradually rather than trying to build the entire thing at once which would have been overwhelming.

After creating a simple file system based on hard coded values, I decided to try and connect the file system to a database to allow for non-hard-coded paths in the file system. I wanted to ensure that any direct communication with the database via SQL was abstracted away into a separate module to avoid "polluting" other parts of the code with implementation details specific to the database. This made the file system code easier to read due to the abstraction of the database and has the additional benefit of making the database code easier to reuse. After connecting the database to the file system, I needed a way to modify the database so I started to work on the CLI. I added a simple interface to add and remove tags from the database by using and extending the database module I previously developed for the file system. I decided not to handle command line argument parsing myself and instead used the popular "clap[9]" library to build a command line interface in a declarative fashion. Clap integrates with the Rust

[8] These directories are not actually hidden, rather they do not actually exist until they are looked up.

[9] Clap is an acronym for "command line argument parser" and can be found at the following url https://crates.io/crates/clap

documentation system to generate command line help messages from a single source of truth. This source of truth is the special documentation comments within the structures and enums used by clap to build the CLI. This makes it extremely easy to keep both the documentation and command line help in sync, and makes it easy to remember to update the documentation as it is right next to the corresponding code.

Next I created the concept of queries within `tagfs` and decided to make a mini DSL to specify them. This required me to create a tokeniser that splits a given query into a stream of tokens. This stream of tokens can then be converted to SQL via another function. I then packaged these functions into a sub-module of the database module, so that they could be used from the CLI (via a dedicated subcommand) and from the file system. When developing the query sub-module, I decided to simplify the implementation by using the principle of "Rubbish in, rubbish out" such that if the user gave a malformed or nonsensical query then it would be allowed for the query builder to output nonsensical or malformed SQL. My theory was that generating invalid SQL was not a problem because it would be caught and rejected by SQLite. This greatly simplified the query builder because it meant that I would not have to write a fully fledged parser and SQL generator for the query language. However, there were downsides to this approach such as the inability for `tagfs` to tell the user where the error in an invalid query is because `tagfs` does not have this information. I think this was a fair trade-off, because queries tend to be on the shorter side and spotting errors is usually fairly simple.

4.3.1 Software Development Practices

During the development of `tagfs` I tried to follow good software development practices. The aim of these practices being to improve the robustness of the code and to ensure that the code is readable and therefore maintainable. One of these techniques is static analysis. Static analysis is used to detect potential problems in the code without actually running it . Rust has a first party static analysis tool called clippy[10]. Clippy can detect style problems and suggest alternatives, such as transforming iterator chains that contain a `filter` followed by a `map` into a single `filter_map` operation. Lots of the clippy style detections are for seemingly trivial problems (for example preferring the use of `xs.first()` over `xs.get(0)`) however, in my experience I found that these "trivial" problems accumulate over time to make the code more difficult to read. In my opinion, putting in the effort to correct these small issues is important for improving the clarity of the code. Additionally clippy can also detect potential performance issues, for example clippy can detect useless heap allocations that would be better served by a simple reference to an existing allocation. Using clippy in this way allows me to use the experience of the wider Rust community when developing my project. The community opinion on best practices is reflected in the suggestions made by clippy. This allows me to be more confident that my code base follows good coding practice.

Whilst developing the project, I endeavoured to document the public API functions of `tagfs` and some of the internal private functions also. Rust and the accompanying cargo[11] tooling makes creating and maintaining documentation simple and hassle-free by keeping the documentation next to the code that it documents. Code is documented through the use of special comments, placed directly above the relevant function or module, that describe what the code should do. When cargo is invoked to build the documentation using the `cargo doc` subcommand, all of these documentation comments are compiled into structured and browsable HTML documentation[12]. Additionally, cargo will also build the documentation of all the dependencies of the project to create a cohesive collection of documentation for all code in the project (this was extremely useful in periods of internet outage). I found that the act of creating the documentation was more important than the actual documentation itself. Creating the documentation helped me to think clearly about what a given module should do and also helped me to discover any invariants that must be accounted for when calling particular functions. An example of this is the documentation for the "fs" module that helped me to both document and understand exactly how and in what order the "fuser" library will call my code when an "ls" is performed in the file system.

Another software development technique I made use of during development was conditional compilation in the form of cargo features [1]. Conditional compilation is a method for compiling the code base in different ways depending on a set of input parameters. For example, it might be desirable to compile a particular block of code only on certain platforms - conditional compilation allows this to be expressed

[10] Named after everyone's favourite paper-clip. Can be found at https://github.com/rust-lang/rust-clippy

[11] Cargo is the Rust programming language's build system and package manager, but it also includes an interface to the rust compiler's built in test runner and documentation builder. The cargo documentation can be found at [1]

[12] It should be noted that none of these ideas about documentation are unique to Rust. The Rust documentation process is heavily inspired by and very similar to older systems such as javadoc and doxygen.

directly in the source code. In `tagfs` I use conditional compilation to allow the user to specify, at compile time, whether they would like to include the autotagging functionality. The way that I achieve this is by using cargo features to define an "autotag" feature and to specify the additional dependencies that are required to build this feature. Then in the source code the module that implements the autotagging feature is preceded by the declaration "`#[cfg(feature = "autotag")]`" which tells the Rust compiler to only compile this next block if the condition is true i.e. the "autotag" feature is active. This provides a similar behaviour to preprocessor macros in C such as `#ifdef` and `#ifndef`. The main reasons I chose to increase the complexity of the build process and use conditional compilation was to provide the user with flexibility regarding the number of dependencies required to build the code and to allow the user to disable features that may not be useful for them. I think this is especially relevant for the autotagging feature, because it is a feature that runs tangential to the rest of the code and therefore is easy to "split out" without causing issues elsewhere in the code base. This would not be the case for other more fundamental features, such as tags being able to have an optional value. This feature runs deep throughout the project and therefore would not be a good target for conditional compilation as the additional maintenance complexity would be very high.

I also made use of logging to help me develop `tagfs`. My main usage of logging was to be able to trace which FUSE functions are called and in what order when performing a particular file system operation. It is through this logging that I was able to produce the documentation for how exactly the "fuser" library calls my code. When commands such as "tree" are run on the file system, there can be a lot of logging output. To reduce the logging noise, I used the Rust standard logging package to attach levels to each log statement, so that, at runtime, the logging level can be specified using an environment variable. This means that it is easy to control the logging output without having to tediously guard each logging statement with a conditional. Additionally, since I used the standard Rust logging package I was able to integrate my logging with the logs from the libraries used in `tagfs`. This meant that I could peer into the logs for code that I depend on and use it to help me understand what the code was doing, whilst also being able to easily disable it with one standard mechanism.

4.3.2 Testing

Whilst implementing `tagfs` I made sure to test the project's functionality frequently, both manually and automatically. Manual testing is initially important to check that a feature seems to work as expected, but automated testing provides confidence that a feature is working and stays working throughout changes to the project. To facilitate the creation and running of tests I followed the standard testing methodology for Rust projects specified in the Rust book [7]. This includes the usage of both unit and integration tests. I used unit tests to isolate and test specific functions within my project. An example of this is the `sanitise_path` function that is used to ensure that no two files in the same directory are given the same file name. This function is a good target for unit testing because it is a pure function that does not rely on any external state - it is a simple mapping from inputs to outputs. Additionally, I found that test driven development (TDD) was an appropriate and useful technique for developing the lexer component of the query subsystem. Writing the tests before the implementation ensured that I had a good grasp of what the lexer needed to achieve and helped me to foresee tricky cases that the lexer would need to handle. Figure 4.8 is a code snippet from one of the tests in my project. It uses the `assert_eq!`[13] macro to test equality between its arguments and to provide a useful error message if the test fails. These tests were particularly useful when it came to adding the `==` operator at a later date to the initial implementation of the query subsystem. It ensured that the addition of a new operator did not break the existing functionality, which reduced the fear associated with modifying already working code.

I used integration testing to test the public API of the library component of `tagfs`. This is the API that both the command line interface and the FUSE file system use to operate the tag database. Within these tests, the database is created entirely from scratch and stored in memory, so that the tests are easily reproducible. The integration tests can be found in the `tests/` directory in the root of the project. In Rust, integration tests are entirely separate crates[14], so by design they cannot access anything that normal consumers of the library cannot.

I also used integration testing to test the FUSE file system. This is done by creating a new temporary database and a temporary mount point. The file system is then mounted at this mount point using

[13] In Rust, macros that can be called like normal functions are terminated with a '`!`' to help differentiate them from normal functions.

[14] "Crate" is the Rust specific term for compilation unit.

```
assert_eq!(
    super::lex_query("      (     not nothello   = \"(wor\\\"=ld)\""),
    &[
        LeftParen, Not, Tag(String::from("nothello")), Equals,
        Value(String::from("(wor\"=ld)"))
    ]
);
```

Figure 4.8: Example unit test for testing the lexer functionality of the query subsystem.

the new database. The database is then modified using the public API methods exposed by the `tagfs` library. These updates are then seen reflected in the file system where they are programmatically checked to ensure correctness. To make it easier to write tests that check the file system, I developed a series of macros (similar to the built in `assert!` and `assert_eq!` macros) to test file system state. These include `assert_symlink!` to ensure that a path exists and is a symlink, and `assert_dir_children!` to ensure that a directory is located at the given path and that the directory's children in the file system match those given to the macro. These tests can also be found in the `tests/` directory.

Chapter 5

Evaluation

I believe that the `tagfs` software is useful and achieves its goals of implementing a method of tag based file management via a virtual file system. This is realised by the file system component of `tagfs` that allows the user to browse and query their files in a more abstract way by using tags instead of file system paths. The additional task of tag management takes place using the command line tool. Together these allow a user to have a reasonably complete tag-based file management experience. The implementation of the virtual file system of `tagfs` is robust and handles edge cases gracefully without unintended crashes. This is achieved through extensive use of Rust's "Result" type to easily propagate errors higher up the call stack to where they are more relevant, and via attaching context messages to lower level error messages to make them more user friendly for both the programmer and the user. Using the Rust "Result" type also makes thinking about error handling mandatory, because to access a value wrapped in a result type you must handle the error or explicitly ignore the error. With the use of this pattern, it is impossible to forget to handle an error as all error handling is explicit. This is in contrast to languages such as C where error handling is often handled through negative return values (this is the case with the fork() libc function) this makes it very easy to forget to handle error cases and as a result can reduce the robustness of the software.

Rust also helped me develop the project via its strong static type system. This caused many errors that would be runtime errors in dynamically typed languages to become compile time errors in Rust. Compile time errors are very much preferable to runtime errors because it is not necessary to run the code path that causes the error. Static typing in this way also reduced the amount of trivial tests I would have had to write in a more dynamic language. For example, I would have had to test how a function behaves with unexpected input types (e.g. giving a string when the function expected an integer) and ensure that the program did not crash when run this way. I believe therefore that static typing reduced the development time of my project (because I had to write less trivial tests) and increased the robustness of the software.

Another way that I found Rust to have been a good fit for my project was via the use of the Rust build system `cargo`. The cohesiveness of `cargo` in containing a documentation generator, testing harness and static analysis tool all in one package, meant that I could more easily focus on developing the project correctly rather than fighting with tooling. The fact that I did not have to choose a testing library and configure it, allowed me to just get on with writing tests. This meant that I was more likely to write more tests as there was such a low barrier to entry for creating them.

The project's robustness is further improved by "standing on the shoulders of giants". In the case of `tagfs` this means the use of high quality libraries, and development tools. Both of the two main libraries used (`fuser` and `rusqlite`) proved to be as high quality as I had hoped them to be. However, the documentation for the `fuser` was a little lacking, but I was able to remedy this by reading the source code. Thankfully the code seemed to be written with care, and I was able to navigate the foreign code base successfully to solve my issues. The `rusqlite` library had no issues with documentation, and I could also use most of the existing documentation for SQLite when it was insufficient. I believe that using an SQL database proved to be a good decision for this project as it was very easy to integrate with the code that I had already iterated on up until that point. This was compounded by the fact that I was already familiar with the basics of SQL and I had experience with SQLite in other projects.

The design of the system is flexible and does not restrict the user or funnel them towards any particular

tagging methodology. This allows the user to create their own tagging system backed by `tagfs` rather than forcing a structure onto them. This flexibility is especially appreciated by the target user base, because (as power users) they do not want to be artificially constrained within a system that limits them to only the ideas that the developer originally thought of. The idea of flexibility is also extended to the command line component of `tagfs`, which is specifically designed to allow extensibility via the traditional UNIX method of pipes and shell scripts. This again allows the user to mould the system to their preference rather than relying on the developer to have conceived of every use case beforehand. Sticking to established platform conventions for the CLI, allows the user to easy integrate `tagfs` into their existing command line based workflow, and allows the software to feel more natural to existing technical users. Additionally, the file system component of `tagfs` naturally fits into a technical UNIX user's environment by virtue of the old adage "*everything is a file*"[1]. This further strengthens the integration between `tagfs` and the user's everyday tools by allowing them to interact with `tagfs` like any other part of the system. The last way in which `tagfs` integrates with the user's system is through the `edit` subcommand. This allows the user to edit the tag database with their preferred editor. I believe this is a much better solution for editing the tag database than developing a full custom editor. This is because it allows the user extreme flexibility in how they edit the tag database, and it reuses the user's editor to do so, which as a power user they are surely familiar and comfortable with. This allows the user to forgo learning a single use specific editor, and instead focus on the task of editing the tag database.

Using a DSL (domain specific language) for querying the tag database, is another way in which the system delivers the flexibility required by the target user base. The language allows users to be specific about defining a query that returns exactly the results they are interested in. This in combination with the stored queries feature, allows the user to spend the time to set up queries exactly as they desire, and then save them for later easy access. Setting up these stored queries and learning the query language itself does require initial investment into the software, but I believe that this is not a problem for the targeted users. This is because, as power users, they are used to software requiring initial effort to learn, but ultimately providing lots of value once over an initial hurdle. Examples of software in this category are the traditional UNIX power user utilities such as vim, emacs and the shell.

I believe `tagfs` to be an example of a reasonably professionally developed software project. The usage of specific professional development techniques are spoken about in detail in Section 4.3. In general, the use of professional practices greatly helped me throughout the project. An example of this is building confidence in my code via testing, which allowed me to feel comfortable in modifying and extending my code, because I knew that any accidental modification to existing functionality would be caught by the tests. The use of static analysis tools helped me to write clearer and more correct code by warning me about potential problems with the code base. Keeping on top of these potential problems resulted in the code base being easier to read and therefore reason about. This could also have contributed to the robustness of the project through improving the clarity of the code and therefore making bugs easier to find.

Using modular design principles such as encapsulation allowed me to more easily reason about my code, because it allowed me to hide implementation details that distract from the higher level data flow. This results in code that reads more nicely as a list of actions to perform rather than an unnecessarily complex information overload. An example of this in the code is the database module. Encapsulating all database functionality within a dedicated module, allowed me to expose a simple interface to the rest of the program for working with the database. This means that I did not have to give direct access to the underlying database connection to other parts of the application. As a consequence of this, I can rely on the program only performing known actions via the interface, so there is no ad hoc SQL scattered around the program. The increased code clarity provided by this sort of modular design results in more maintainable code which, for `tagfs` means that in the future someone new to the codebase could implement some of the features described in Section 5.1 with a minimal onboarding period as they work to understand the project's code base. More maintainable code is not just a benefit for other developers, as when coming back to your own code after a long period of time it can be difficult to remember why certain design decisions were made. This is where documenting and making the code as obvious as possible is important, for which modular design is a great helper.

[1] I am unsure who said this originally, but it is has been a very common expression in the UNIX / Linux community since at least the 90s.

5.1 Potential Future Work

Due to the modular design and iterative development style used to develop the `tagfs` project, it would be easy to extend the project with new features and even entirely new interfaces. For example, it could be interesting to create a GUI for `tagfs`. This would allow users to browse and edit the tag database in a more visual way. A more visual style of tag manipulation could broaden the intended user base considerably. This is because allowing the users to modify the tag database through a GUI would remove the requirement for a user to have experience with the command line, and since almost all non-technical users are inexperienced with the command line this could potentially create a more feasible route to them using `tagfs`. For technical users however, I do not think a GUI would bring a great deal of value. This is because I think that most technical users would want to automate their interactions with `tagfs` via scripting, and only rarely would they need to manually interact with the program. In this case of rare manual interaction, it would not make sense to open a dedicated GUI and attempt to remember its idiosyncrasies. I also believe technical users would prefer to interact via the CLI because it would be more familiar to them after having development scripts around it.

Another potential avenue that the project's development could be taken down is allowing the user to modify the tag database from the file system. This would make using the CLI tool less necessary, and make editing the tag database more intuitive and natural. There are lots of ways a feature like this could work. One way would be to allow the user to remove a tag directory within the virtual file system, the file system would then intercept the request and as a consequence remove the tag represented by the deleted directory from all file paths that it is currently tagged on. Another way could be to allow the editing of a file's tags by editing the ".tags" file that is automatically generated by the file system. This could work similarly to the edit subcommand from the CLI. Even deleting a ".tags" file could be implemented to remove all traces of that file path from the tag database.

One feature that could be useful for users would be the ability to customise the autotagging process. This could be done with custom user created rules that use regular expressions to match path components or specific file metadata. After a rule has been matched, an action can take place to generate tags for the matched file. This could be as simple as assigning a static string to a given rule or as complex as allowing an external script to be used to generate a set of tags. This would allow the user to be more confident in using the autotagging feature as they would be sure it would tag files exactly to their specifications, and an increase in the usage of autotagging means there is less friction in adding new files to the system.

Chapter 6

Conclusion

I believe that my project has been successful in achieving its goals to create a tag based file system. This can be seen to be the case through the documentation given in Section 4.2 regarding the features of the CLI tool and the virtual file system. Additionally, the code that implements these features can be found in the `src/` directory of the accompanying code repository. The definition of the CLI interface is located in the `cli.rs` file and the entry points for each subcommand can be found in the `tagfs.rs` file. The code for the file system resides within the `lib/fs.rs` and `lib/fs/entries.rs` files. The database layer can be found in the `lib/db.rs` file and its sub-modules can be found in the `lib/db/` directory. These include the query builder and the edit subcommand representation format. Combining these components together results in a reasonably useful package for managing files using a tag based strategy and accessing them from a virtual file system.

The project has been developed using good professional practices. These practices include automated testing (both unit and integration style), modular design, static analysis, and creating documentation. I have discussed these practices and how they have helped me to develop `tagfs` in more detail in Section 4.3. Evidence for these practices can be found throughout the project code, such as documentation comments on public functions and comments used throughout the code to explain particularly tricky or confusing sections of code. Additionally, evidence for testing can be found in the `tests/` directory in the root of the project. These are the integration tests. Unit tests are located in the main project directory (as per Rust testing guidelines found in the Rust book [7]). An example set of unit tests can be found in `src/lib/db/query/tests.rs`.

The project was developed with a specific group of users in mind, namely technical users on Linux. I believe that I have produced a piece of software that is tailored for these users and reflects their needs. This is discussed in both the design period of the project in Section 4.1 and in more detail in the evaluation in Chapter 5. I believe the software is useful for these technical users because it integrates with their existing work flow. This is achieved via exposing the tag database via the file system, and via the usage of the convention following command line tool. These compose together to produce a familiar and easy to grasp experience for the target user base.

Bibliography

[1] The cargo book. https://doc.rust-lang.org/cargo/reference/features.html. [Online; Accessed: 2023-04-28].

[2] FUSE - the linux kernel documentation. https://www.kernel.org/doc/html/next/filesystems/fuse.html. [Online; Accessed: 2023-03-07].

[3] FUSE structure - wikimedia commons. https://commons.wikimedia.org/wiki/File:FUSE_structure.svg. [Online; Accessed: 2023-03-07].

[4] fuser/fuse_pure.rs. https://github.com/cberner/fuser/blob/39d4177e809c7ee3b6757136fee8a28d5f41f040/src/mnt/fuse_pure.rs. [Online; Accessed: 2023-04-28].

[5] Memory safety - chromium.org. https://www.chromium.org/Home/chromium-security/memory-safety/. [Online; Accessed: 2023-04-07].

[6] Overview of the linux virtual file system. https://www.kernel.org/doc/html/next/filesystems/vfs.html. [Online; Accessed: 2023-03-07].

[7] The rust programming language - test organization. https://doc.rust-lang.org/book/ch11-03-test-organization.html. [Online; Accessed: 2023-03-24].

[8] The rust programming language - understanding ownership. https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html. [Online; Accessed: 2023-04-07].

[9] The rustonomicon. https://doc.rust-lang.org/nomicon/. [Online; Accessed: 2023-04-11].

[10] SQLite - most widely deployed database engine. https://sqlite.org/mostdeployed.html. [Online; Accessed: 2023-03-10].

[11] What is memory safety and why does it matter? https://www.memorysafety.org/docs/memory-safety/. [Online; Accessed: 2023-04-28].

[12] XDG base directory specification. https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html. [Online; Accessed: 2023-03-14].

[13] *inode(7) Linux man page*, 6.03 edition, February 2023.

[14] *libc(7) Linux man page*, 6.03 edition, February 2023.

[15] *link(2) Linux man page*, 6.03 edition, February 2023.

[16] *mount(2) Linux man page*, 6.03 edition, February 2023.

[17] *setuid(3p) Linux man page*, 6.03 edition, February 2023.

[18] *stat(2) Linux man page*, 6.03 edition, February 2023.

[19] *symlink(7) Linux man page*, 6.03 edition, February 2023.

[20] *syscalls(2) Linux man page*, 6.03 edition, February 2023.

[21] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), page 213–229, New York, NY, USA, 1965. Association for Computing Machinery.

[22] Pieter Omvlee. A novel idea for a new filesystem. *June*, 29:1–7, 2009.

[23] D. M. Ritchie and K. Thompson. The unix time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1929, 1978.

[24] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies FAST'17*, 2017. [Online; Accessed: 2023-04-07] https://www.usenix.org/system/files/conference/fast17/fast17-vangoor.pdf.

Appendix A

Cargo.toml

This is a listing of the manifest file for `tagfs`. All direct dependencies can be seen under the `[dependencies]` header. Their web pages can be found at `https://crates.io/crate/{name}`.

```
[package]
name = "tagfs"
description = "Tag-based virtual file system backed by SQLite."
authors = [[redacted]]
version = "0.1.0"
edition = "2021"

[lib]
name = "libtagfs"
path = "src/lib/lib.rs"

[[bin]]
name = "tagfs"
path = "src/tagfs.rs"

[dependencies]
anyhow = "1.*"
audiotags = { version = "0.4.*", optional = true }
camino = "1.*"
chrono = { version = "0.4.*", optional = true }
clap = { version = "4.*", features = ["cargo", "derive"] }
constcat = { version = "0.3.*", optional = true }
env_logger = { version = "0.*", default-features = false }
fuser = { version = "0.*", default-features = false }
indexmap = "1.*"
kamadak-exif = { version = "0.5.*", optional = true }
libc = "0.2.*"
log = "0.*"
mktemp = "0.*"
once_cell = "1.*"
regex = { version = "1.*", optional = true }
rusqlite = "0.*"
serde_json = { version = "1.*", optional = true }
ureq = { version = "2.*", features = ["json"], optional = true }
walkdir = { version = "2.*", optional = true }

[features]
default = ["autotag"]
autotag = [
    "dep:walkdir", "dep:audiotags", "dep:ureq", "dep:serde_json", "dep:regex",
    "dep:constcat", "dep:kamadak-exif", "dep:chrono",
```

```
]

[profile.release]
strip = true
lto = "fat"
codegen-units = 1
```