NetApp Capstone Project Spring 2022 Final Report

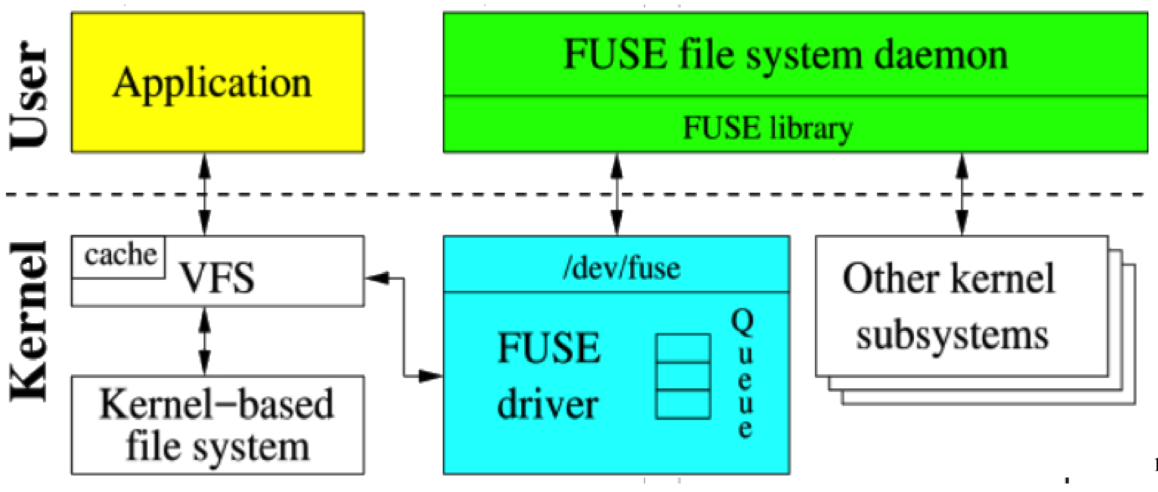
FUSE Kernel Extension

Carter S. Levinson, Danny Yu, Samuel Lasky, Qizhe Wang

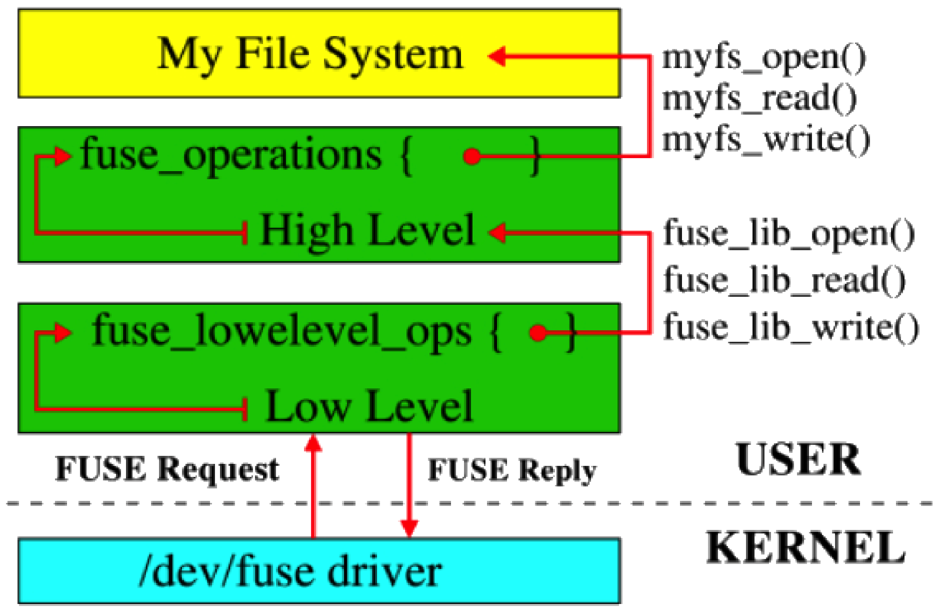
Introduction

FUSE (Filesystem in Userspace) is an interface framework for userspace applications on UNIX-like operating systems to create customized filesystems without directly modifying kernel code. It provides programmers the tools to build a modifiable UNIX filesystem at user level, largely without worrying about kernel-space. Figure (1) outlines the high-level architectural features of FUSE. In userspace, FUSE consists of a background program (i.e. a daemon program) which primarily interacts with the kernel through the /dev/fuse file. The FUSE device file maintains a queue of requests which are processed by the userspace daemon. The replies are written back to /dev/fuse which interacts with the kernel's filesystem through the Virtual Filesystem (VFS) abstraction layer. When a user space application sends a request to the VFS, the VFS reroutes this request to /dev/fuse in order to be processed by the daemon. Finally, note that the FUSE daemon is linked with libfuse which provides a framework for filesystem interaction as well as many useful helper functions, e.g. mounting and unmounting of the filesystem.

The FUSE module essentially acts as a bridge for communication between the user end and the kernel, thereby allowing users to create and manipulate filesystems via a handler program linked to the libfuse library without altering the kernel code. This can dramatically speed up development speed since the user does not have to recompile the kernel everytime they make some small change to the filesystem code, as well as giving users more flexibility and control over their filesystem.



[1]



[1] Bharath Kumar Reddy Vangoor, Prafful Agarwal, Manu Mathew, Arun Ramachandran, Swaminathan Sivaraman, Vasily Tarasov, and Erez Zadok. 2019. Performance and Resource Utilization of FUSE User-Space File Systems. *ACM Trans. Storage* 15, 2, Article 15 (May 2019), 49 pages. https://doi.org/10.1145/3310148

Figure (2) shows some of the internal organization of libfuse. The operations of a custom filesystem, here referred to as "myfs" are routed to high level fuse operations which map closely to typical filesystem operations such as chown() or chmod(). These high-level operations are mapped to low-level operations such as getattr() and setattr() by libfuse and largely let the developer program without concern for path to inode mappings.

For our capstone project this semester we worked closely with the company NetApp. NetApp is a Fortune 500 company known for their cloud and application data services[2]. We met with the project sponsors on a weekly basis, communicating through Zoom video chats, Slack messaging services, and a Trello board for project management/organization. Our primary objective was to create a filesystem that applies a disk space quota enforcement feature to the existing open-source FUSE filesystem source code.

With respect to the division of labor, both the group members and the sponsors agreed that a round robin approach of assigning project tasks was the best way to divide up the work. This was done to ensure each and every group member gained experience and familiarity across the board with respect to the many roles required to successfully implement the project. For example, during a typical sprint there will be one group working on testing while the other works on achieving the goal for the current epic. The roles are then swapped between the two groups for the next sprint.

We envision the main users of this project to be admins and storage space owners who like to provide local or online storage systems. Our intervention is useful because it helps implement a fair and equitable approach to storage that should prevent individual users on a multiuser system from unfairly hogging system resources.

[2] https://fortune.com/company/netapp/fortune500/

Project Goals

The main goal is to utilize the FUSE filesystem to implement a kernel extension which intercepts system calls, and implement the syscalls to enforce a maximum user quota feature, analyzing those calls to check if the user has exceeded the maximum allocated disk space capacity or user quota, for example, each user gets 5GB of free storage space before needing to pay for additional storage.

For this project specifically, the additional storage feature is not going to be an implemented feature of the filesystem. However, if the individual user's maximum free storage amount has been reached, the system automatically prevents users from further exceeding it. Additionally, an in-memory database is implemented which stores user data usage. The database log entries are then used to determine whether or not the aforementioned maximum quota has been reached.

Languages and Frameworks

The primary language for the project is the C programming language. Since we are working with syscalls and something at a kernel level this is to be expected. In addition, to avoid complications for the database, we decided it is best to use a file-based database approach instead of the need to use a DBMS to manage the database. This allows us to log information about user storage usage and operation successes and failures, providing reference for ensuring the enforcement of individual user quota.

Since the project involves creating FUSE syscalls which will intercept the kernel syscalls, a good approach to testing would involve the use of shell scripts. Since the shell commands have been well-tested and invoke the filesystem syscalls under the hood, it will allow us to identify where the errors are in our implementation, since it is unlikely to result from a bug in, e.g., "ls" or “mkdir”. This was the initial plan for the project. However, due to the ease of reading and writing in python and the providing of the interfaces for working with the SQLite database library, the pytest framework is the testing framework used for testing purposes. Python subprocesses are also used to speed up testing.

The project is based on the open-source FUSE framework. The Linux kernel documentation defines FUSE as a userspace filesystem framework which “consists of a kernel module (fuse.ko), a userspace library (libfuse.*) and a mount utility (fusermount).” [3] FUSE works through a daemon program which intercepts file related syscalls and passes them down to the filesystem in a secure manner.

Main Project Features

The main project features includes the function of intercepting file related syscalls and operations, a quota based system for user storage with a certain amount of maximum storage space, and a SQLite database for managing the quotas and documenting operation logs.

Project Details

The architecture for this project is outlined by the UNIX philosophy and, specifically, the syscalls for interacting with the filesystem.

In the UNIX family of operating systems, it is well known that “everything is a file”, so the syscalls for interacting with the filesystem can be repurposed for almost any I/O one can think of. This is done by opening and closing file descriptors, which can refer to anything from space on a block device, network sockets, and physical hardware such as the CPU or audio card. Once a file descriptor is opened, client programs may use the syscall interface to perform filesystem operations from the resource. UNIX files also exist in a specific place in the filesystem, which is organized as a tree structure of directories (which are also represented with a

[3] https://www.kernel.org/doc/html/latest/filesystems/fuse.html#definitions

file descriptor). In FUSE, we add a layer of indirection between the kernel and the program so that we can create user space filesystems.

As mentioned, the project uses SQLite[4] as an in-memory database/application file format, and the Oracle VirtualBox™[5] for virtualizing the Ubuntu[6] OS on our Windows and Mac machines. The project also consists of a hand rolled logging framework, specifically the log.h header file, to track file changes and handle error reporting. Makefiles are used as our build system.

It is worth mentioning that filesystems are intrinsically tricky beasts, since there are many potential issues that we are on the lookout for. This includes concerns associated with data integrity, protecting against race conditions and hardware management, data security, etc. Furthermore, the C programming language is notorious for 'gotchas' such as segfaults and poor handling of (null-terminated) strings. The latter will be particularly concerning because our program will have to handle and parse file paths. The main challenge will be writing a memory safe and persistent program to manage the filesystem and producing a reliable program for whoever may use it.

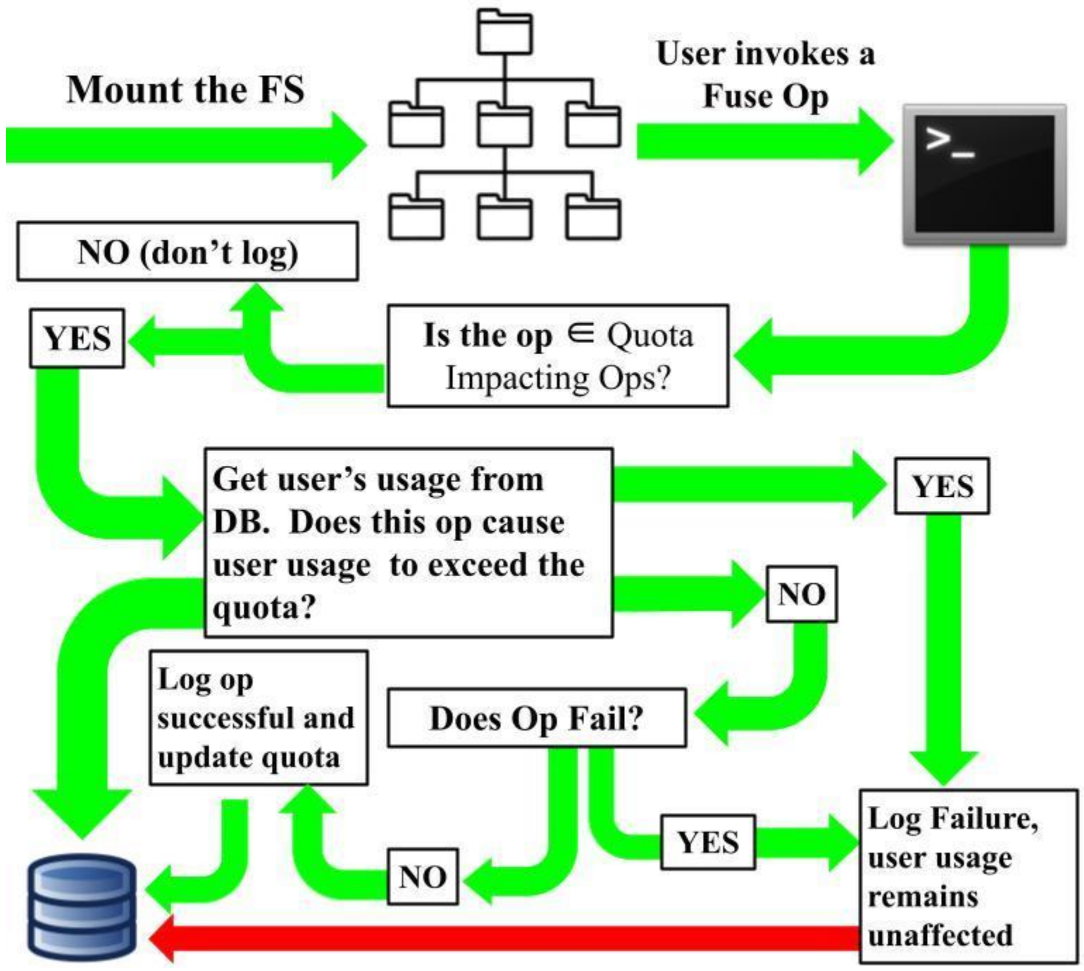
Our FUSE extension is developed to execute on Linux systems, specifically the current LTS version of Ubuntu. Other operating systems will not be supported since the specific FUSE framework used for this project is meant to work on Linux systems only.

[4] https://sqlite.org/index.html
[5] https://www.virtualbox.org/
[6] https://ubuntu.com/

Final Product

The final product of the project consists of complicated tests and operations boolean checks to ensure the logging system and quota enforcement works properly as intended. The main working environment of the product is Ubuntu, which is simulated on our Windows and Mac machines through the use of a virtual environment. This also helps to ensure the proper execution of the FUSE filesystem framework in case of the use of different machines among the individual users.

The flow chart below shows the general condition checks and how the implemented filesystem executes to properly log user quota updates and operation success or failure.


Mount the FS
User invokes a Fuse Op
>_
NO (don't log)
YES
Is the op ∈ Quota Impacting Ops?
Get user's usage from DB. Does this op cause user usage to exceed the quota?
YES
NO
Log op successful and update quota
Does Op Fail?
YES
NO
Log Failure, user usage remains unaffected

Challenges Faced

One of the immediate challenges we experienced is identifying the operations in the FUSE API that would impact user quota directly, this includes the manipulation of disk storage space and transfer or change of ownership of a file. Since there are roughly 27 filesystem operations in total, it is important to alter the correct operations to ensure the execution of the filesystem. We then point out how each operation affects the individual user quota and make the corresponding changes to the source code.

Operations implementation is both a major part and challenge of the project. The initial challenge was figuring out how to update the database through the operations, which a SQLite interface was able to help the source code to communicate with the SQLite database by providing detailed specifications for C APIs for SQLite. This goes hand in hand with the allocation of memory blocks in the main local memory to ensure the safety of storage space allocation in the system.

We also needed to consider how each operation works and if they affect the execution of other operations, that is if one operation would invoke operations or not, therefore potentially causing conflicts or issues within the database. One of these examples is the chown operation, since it unlinks the owner of the file at the provided file path and links it to another user, we have to consider possibilities of it invoking the link and unlink operations.

Last but not least, filesystems such as this are mainly meant to handle single user filesystem manipulations, and the need to alter the operations to handle possible multiple users at the same time still remains a potential issue.

Conclusion

In conclusion, our project creates a safe and consistent userspace alternative for the filesystem related syscalls. The implemented quota enforcement feature will be useful for admins and users when managing space both in the filesystem as well as on block devices. The feature uses minimal overhead so that the latency of file operations will appear nearly identical to the invocation of syscalls to handle requests during frequent reading and writing. Furthermore, the increased execution speed in the userspace is benefitted from our program since it does not (directly) need to perform context switches for every filesystem operation, which avoids the generation of a significant amount of overhead. It is our hope that users of this kernel extension will find it a fast and stable supplement to the current environment of UNIX/Linux filesystems that are currently provided.